Table of Contents

Intro

This documentation describes Austin Kobayashi's work on RichReview.net Learning Management System during May-Aug, 2019. Austin built a full-stack system that includes a nuxt-based frontend, an express-based backend, and a legacy code written by Dongwook running as a part of the frontend codeset (denoted as *legacy* in the following).

Backend

The backend is a (mostly) restful Express NodeJs server that interfaces with the Azure redis to provide data for the Frontend. It is bound to port 3000, and the redis database that it uses is defined by the environment (production or development). The functionality is split between “DatabaseHandler”, such as “AssignmentDatabaseHandler”, which handles all functionality for assignments. These handlers are in the bin directory. The backend has a heavy emphasis on the singleton design pattern, with all DatabaseHandlers being singletons.

Note: DatabaseHandler never import a different DatabaseHandler (eg, CourseDatabaseHandler doesn’t import AssignmentDatabaseHandler), in order to avoid cyclic dependencies.

This is an extremely important point, since cyclic dependencies will cause “Not Found” errors for function calls, despite the respective function being imported. Instead, ImportHandler.js handles all of the imports. ImportHandler is imported by each script in the /routes directory, and the ImportHandler is passed as a variable to any calls that are made to a DatabaseHandler. The interface with redis is handled by AsyncRedisClient.js and RedisClient.js. These are both singletons that are used by the DatabaseHandlers to obtain a reference to a redis connection. There are two types of redis clients (RedisClient.js. and AsyncRedisClient.js.) since the backend is slowly transitioning to only use AsyncRedisClient.js.

Running the Backend

The backend is run using the command “sudo npm start” while in the Backend/ directory. Sudo is required, since the backend requires access to the root path of the hard drive to store temporary pdf files. See the get_instance() function in DocumentUploadHandler.js for more details.

Running the Django Server

The django server is used to analyze pdf documents. To run it, navigate to mupla_core/django_server and run the command “sudo python manage.py runserver 5000” with Python 2

KeyDictionary.js

This is an extremely important script that holds the prefix values for all keys in redis. KeyDictionary is, and should always be used when creating a key string (eg, KeyDictionary.key_dictionary[‘user’] + user_id), since it will make the code much more maintainable. If a key ever needs to be changed, changing it in the key dictionary allows for the changes to be reflected across the entire backend without any extra work. Any new prefixes for new database tables that are added should be added to the key dictionary

AzureHandler.js
This script handles the connection to the azure blob storage so that pdfs can be uploaded to Azure

Multicolomn.js & MuplaHandler.js
These scripts handle the layout analysis of an uploaded pdf. They are an adaptation of the legacy scripts that use to run in the legacy frontend. They have been moved and adapted to run in the backend for a better pipeline. These scripts rely on the dom to perform calculations, so the package “jsdom” is required to simulate a dom in the backend.

RedisToJSONParser.js
Data retrieved from redis isn’t in json format, so this script will convert the data retrieved from redis and convert it to json

Frontend
The frontend is a server-side rendered nuxt.js application that is hosted on port 80 & 443 for production and port 8001 for development. Using nuxt’s “serverMiddleware” property, the legacy express website (./Frontend/legacy) is run in tandem with the nuxt app. The “serverMiddleware” allows for an api to act as the handler to a route, so the legacy app is the handler of the “/” route, meaning any traffic to “/” (or it’s subdomains) are handled by the legacy express app. This means that all traffic to the website is handled first by the legacy express app, and then by the nuxt.js app. If the legacy express app has a handler for the route, then that handler is used, otherwise the nuxt.js handler is used. This means that the legacy express app can be used to filter or act upon all traffic before it is sent to the nuxt app, which can be useful for something like authentication. The nuxt.js portion of the frontend runs under the /edu domain and its sub-domains. UBC and Google users will be automatically redirected to the /edu path after logging in, or when accessing richreview.net while logged in.

Running the Frontend
The frontend is run using the “sudo npm run dev” for a developer mode. For production, do “sudo npm run build” then “sudo npm run start”. Sudo is required so that the frontend can bind to port 443 and port 80. Running the frontend will first build the legacy express app, then it will build and run the nuxt app.

Webapp Sync
The legacy express app has a web app function that will synchronize all public javascript files to the Azure content delivery network (Azure Blob storage that is forwarded to CDN). This function (webAppSync) is a part of /Frontend/legacy/app.js and runs whenever the frontend is started. In

the legacy system, the webapp sync and run server functions were promisifed, so that the webapp sync was run, then the server was started. In the current nuxt.js system, there is an issue where the web app sync function constantly prints the list of files being synced while the server is building. It will still properly sync the files to the CDN, but the large amount of console logs makes it difficult to debug any server build issues. This seems to be a combination of the webAppSync no longer being promisified, and nuxt.js using the legacy system as a serverMiddleware.

Components

In nuxt.js, components are reusable layouts that aid in developing a frontend. For example, the footer is present on all pages, so instead of writing the html and css for the footer for each page, it can instead be written once in a footer component, then that component can be imported for each page.

Pages

Pages define the route handlers for the nuxt js app

router.js

This is the route for the nuxt.js app. Normally, nuxt.js will automatically create a router for the project, and each time a new page is created, a route will be created for it automatically. This automatic routing does not work for the project, so the routing is manually done in this file.

Plugins

Plugins define site wide functionality. Event-bus.js is an important plugin that allows for data transfer along the event-bus. This is used to pass data from components back to the page. For example, many modals in the site are used to edit data, and these edits are passed back to the page via the event-bus when the modal is closed.

User Authentication Procedure

The nuxt.js app currently supports Google and UBC login methods for the /edu path, and the legacy demo supports the pilot login strategy. UBC and Google users will be automatically redirected to the /edu path after logging in. The nuxt.js frontend uses the legacy system as the router for the authentication flow; These functions can be found under Frontend/legacy/routes/index.js. The official authentication package for nuxt.js did not support SAML type logins (required for UBC CWL), but the “passport” express package does, which is one of the reasons behind the design choice of delegating authentication to the legacy express app. The nuxt.js app uses a “store” (Frontend/store/index.js) to store information of the current logged in user. The “authUser” property of the store is set when the user logs in, and cleared when the user logs out. Pages in the nuxt.js app check the store when they are loaded, and if there is no current authUser in the store, then the user is redirected to the login page.

UBC CWL

UBC CWL is a SAML strategy that provides user information when a user logs in. This information includes:

- • Unique Id
- • CWL
- • Email
- • Display name
- • First name
- • Last name
- • Enrolments

This information is passed to the backend where it will create a user if they do not currently exist. It will then enrol the user in all of their courses, and remove them from any dropped courses. Due to the certificates required to access CWL, this login method only works from richreview.net.

ELDAP Database

RichReview has an eldap database that holds all of the courses to be synchronized with redis. This eldap is synchronized with UBC's main eldap every 10 minutes. The RichReview eldap has two ou's:

- • ou=richreview.net,ou=Applications,ou=CPSC-UBCV,OU=CLIENTS,dc=id,dc=ubc,dc=ca
- • ou=richreview.net_src,ou=Applications,ou=CPSC-UBCV,OU=CLIENTS,dc=id,dc=ubc,dc=ca

richreview.net_src is automatically unrolled into richreview.net every 10 minutes, overwriting the groups in richreview.net that also exist in richreview.net_src. It leaves alone anything in richreview.net that is not also in richreview.net_src. Anthony Winstanley manages the RichReview eldap can add new courses or remove existing courses from the eldap if required. To add a new UBC course to RichReview, first Anthony must add the new course to the RichReview eldap, then the eldap-sync script will automatically create the course in redis. A good way to view the RichReview eldap is through Apache Studio. An account with administrator privileges is required to access the eldap; Anthony can give an account administrator privileges. To connect to the eldap through Apache Studio, create a new connection with the following configuration:

The RichReview eldap can then be viewed by using “Go to Dn” and pasting:
ou=richreview.net,ou=Applications,ou=CPSC-UBCV,OU=CLIENTS,dc=id,dc=ubc,dc=ca

SAML Parsing

The SAML string returned by UBC when a user logs in must be parsed in order to access the user data. This parsing occurs in the “login_ubc_return” post request handler of /Frontend/legacy/routes/index.js. The SAML string is returned in base64, so it must first be decoded to ascii. The “xml2json” library is then used to convert the string to json. The user data in the resulting object can be accessed via:
<jsonobject>['saml2p:Response']['saml2:Assertion']['saml2:AttributeStatement']['saml2:Attribute']

The fields of the user data do not have intuitive names, so this dictionary provides a better way to access the user information:
UBC_PERSISTANT_ID = 'urn:oid:1.3.6.1.4.1.60.1.7.1';
UBC_CWL = 'urn:oid:0.9.2342.19200300.100.1.1';
UBC_EMAIL = 'urn:oid:0.9.2342.19200300.100.1.3';
UBC_FULL_NAME = 'urn:oid:2.16.840.1.113730.3.1.241';
UBC_FIRST_NAME = 'urn:oid:2.5.4.42';
UBC_STUDENT_NUMBER = 'urn:mace:dir:attribute-def:ubcEduStudentNumber';
UBC_LAST_NAME = 'urn:oid:2.5.4.4';
UBC_COURSES = 'urn:oid:2.16.840.1.113719.1.1.4.1.25';
The user data is then sent to the backend where this dictionary is used to access each field to create or update the user profile in redis.

Google

Google login is handled in the /Frontend/legacy/routes.index.js file via the login_google and login-oauth2-return functions. After a user logs in, the user data is received by login-oauth2-return in a json format. This user data is then sent to the backend where it is used to create or update the user profile in redis

Database Schema (draw.io)

https://drive.google.com/file/d/1Y87fZiRkLdR2xATNBb3l7NUmd3ZEAzym/view?usp=sharing

Virtual Machine

The website is hosted off of an Azure virtual machine. It is accessed with:
ssh rr_admin@richreview.net
Ask Dongwook for the required password

Folders

RichReview is hosted out of the /nuxt folder on the virtual machine. It has the following folder structure:

```
/nuxt
.
+-- ecosystem.config.js
+-- package-lock.js
+-- start.sh
+-- current/
| +-- symlink to releases/production
+-- releases/
| +-- build
| +-- RichReviewXBlock
| +-- production
| +-- RichReviewXBlock
```

Logs are saved in the /logs directory of the virtual machine. There is a file for standard logs and error logs for both the frontend and backend (eg, log_edu_backend.txt & log_edu_backend_err.txt). Note that the frontend has two sets of log files (eg, log_edu_frontend_out-0.log & log_edu_frontend_out-1.log). This is because the frontend is run using the clustering mode of pm2. Clustering mode allows for multiple instances of the server to be hosted and is required for zero downtime deployment. Since there are two instances of the frontend running, there are log files for each instance.

Virtual Machine Server Scripts

restart_backend.sh

This script is used to restart the backend. The backend runs using the node package "forever", so that it can run in the background. The script will first stop all forever processes, then run the backend using forever. It does not kill all node processes when stopping the backend, since that will affect the nuxt processes. It logs to /logs/log_edu_backend.txt and /logs/log_edu_backend_err.txt

stop_django.sh

This script is used to stop the django pdf analysis engine. It will kill all processes bound to port 5000.

restart_django.sh

This script is used to restart the django pdf analysis engine. It first calls stop_django.sh, then restarts the django server. It logs to /logs/log_django.txt

stop_frontend.sh

This script will stop the frontend by killing all processes bound to port 443 and port 80, then calling “pm2 kill”. This script is rarely used and is only for manually stopping the frontend.

/nuxt/start.sh

This is a bash script used to start the frontend. It is automatically called by rrrun_education.sh and should only be called by the user when the frontend needs to be manually restarted. In order to achieve zero downtime deployment, there are two git repos on the virtual machine:

- • /nuxt/releases/build
- • /nuxt/releases/production

The build folder is used to build the frontend and the production folder is used to host the frontend. When this script is run, it will run “sudo npm build” in the build folder. After the frontend has finished building, it will swap the build and production folders, so that the newly built frontend resides in production. Finally, it will restart the server using pm2 for a zero downtime deployment. This script requires sudo privileges so that the frontend can bind to ports 80 and 443.

rrrun_education.sh

This is the main script used to run the server. It will call restart_backend.sh and /nuxt/start.sh in order to (re)start the frontend and backend. The functions that the script performs are broken into 2 conditional cases:

- • The github repository has not changed
- • The github repository has changed

In the case where the repository has not changed, no updates are required so it will simply call restart_backend.sh and /nuxt/start.sh
In the case where the github repository has changed, it will:

1. Update the /nuxt/releases/build/RichReviewXBlock directory to the latest git commit
2. Run “npm install” to ensure that the packages are up to date
3. Run /nuxt/start.sh to restart the frontend and to swap the /nuxt/releases/build and /nuxt/releases/production directories
4. Run restart_backend.sh to restart the backend.
5. Update the /nuxt/releases/build/RichReviewXBlock directory to the latest git commit
6. Run “npm install” to ensure that the packages are up to date.

Utility Desktop

The utility desktop is a desktop currently in X508 that runs utility scripts. This machine is used to perform any utility functions on the server, rather than having the backend perform these functions. This allows for an easier workload for the backend, as well as the flexibility of adding

new utility scripts without affecting the website. It must run on UBC internet in order to access the eldap. It can be accessed through:
ssh dwyoon@198.162.55.119
Ask Dongwook for the required password, it is the same password as the Virtual Machine

Utility Desktop Scripts

These scripts are found under RichReviewXBloc/utils/edap-sync and are run using Python 3
Note: All of these scripts can be run on either the local or the Azure hosted redis. This is defined by setting the path of the “redis_config”. “redis_config.json” is used to run on the Azure redis, “redis_config_local.json” is used to run on the local redis

eldap-sync.py

This script synchronizes the Azure Redis database with UBC’s course eldap so that UBC users are automatically enrolled in courses. It is start using the command ./run-edap-sync.sh and runs in the background, synchronizing the databases every hour. The script will prompt for a username and password. These should be the credentials for a CWL account with access to the RichReview eldap. Course information is pulled from the RichReview sub-eldap. The data in this sub-eldap is fetched every 10 minutes from UBC’s main eldap by the CS department. The eldap stores the course title, and a list of users. Courses are broken up into two entries: one for students and one for instructors. For example, the course cpsc 554y for 2019 winter session will have the entries “CPSC_554Y_201_2019W” and “CPSC_554Y_201_2019W_instructor” which hold the list of students and instructors respectively. The eldap sync script will iterate all courses in the eldap, and create a course entry in redis if the course does not exist. It will then iterate all users listed under the course in the eldap, create a user entry in redis if the user does not exist, then enrol the student to their courses in redis. It will then iterate all of the courses in redis, and check to see if the list of users is the same as in the eldap. If there are any users enrolled in a course in redis, that are not enrolled in the course in the eldap, then that user has dropped the course so they are moved to the “blocked_students” section of the course in redis, and the course is removed from their enrollments in redis.

add-users.py

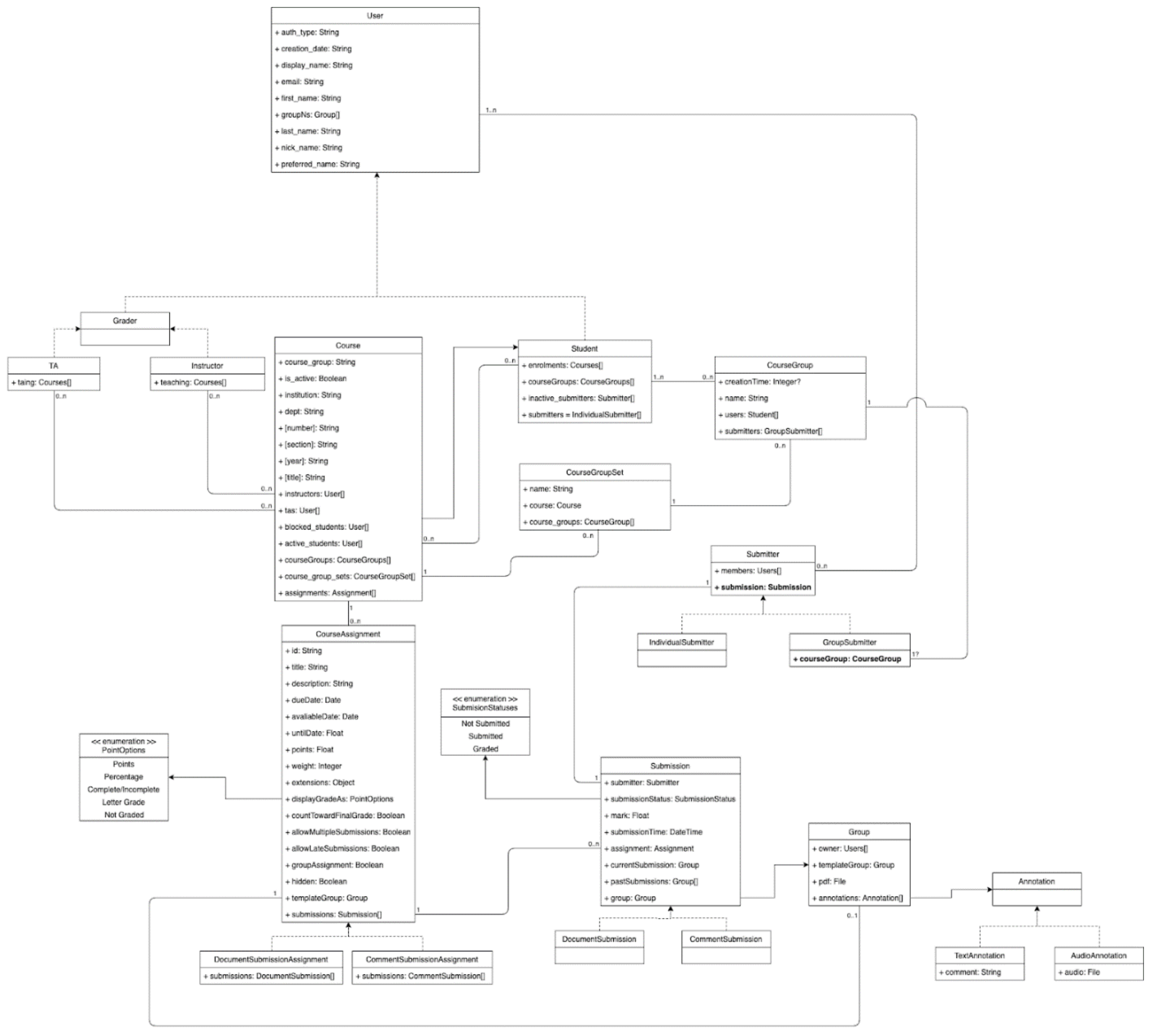
This script is used to mass add test users to a course. It was used during development to test the website performance with a large number of students.

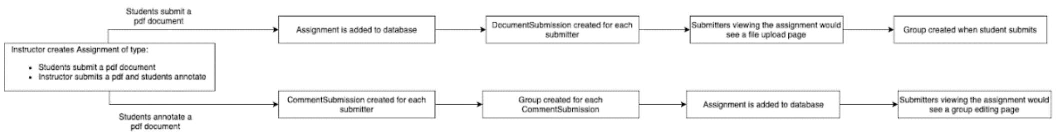
remove-users.py

This script is used to mass remove test users from a course. It was used during development to undo the changes made by add-users.py

update-redis.py

This is a convenience script used to mass alter redis. It is useful when new features are added that require new fields of data. The backend will expect those fields, except existing data in redis will not have those fields, so this script can add the fields to all existing redis data. It will perform either an add/modify operation, or a delete operation on all keys in a table in redis. A table would refer to all keys under a given prefix, such as all keys under “crs:” or “usr:”. An add/modify operation will add a field, or overwrite an existing field with a value. A delete operation will remove a field. The script will first prompt for which type of operation to perform, then it will prompt for which table to perform the operation on. It will then prompt for the field name, such as “auth_type” or “email”. Finally, it will prompt for the default value. For an empty string default value, simply press enter without typing a value.


User
+ auth_type: String
+ creation_date: String
+ display_name: String
+ email: String
+ first_name: String
+ groupNs: Group[]
+ last_name: String
+ nick_name: String
+ preferred_name: String
1..n
Grader
TA
+ taing: Courses[]
0..n
Instructor
+ teaching: Courses[]
0..n
Course
+ course_group: String
+ is_active: Boolean
+ institution: String
+ dept: String
+ [number]: String
+ [section]: String
+ [year]: String
+ [title]: String
+ instructors: User[]
+ tas: User[]
+ blocked_students: User[]
+ active_students: User[]
+ courseGroups: CourseGroups[]
+ course_group_sets: CourseGroupSet[]
+ assignments: Assignment[]
0..n
0..n
0..n
1
1
0..n
Student
+ enrolments: Courses[]
+ courseGroups: CourseGroups[]
+ inactive_submitters: Submitter[]
+ submitters = IndividualSubmitter[]
0..n
1..n
0..n
CourseGroup
+ creationTime: Integer?
+ name: String
+ users: Student[]
+ submitters: GroupSubmitter[]
1
0..n
CourseGroupSet
+ name: String
+ course: Course
+ course_groups: CourseGroup[]
1
0..n
Submitter
+ members: Users[]
+ submission: Submission
0..n
1
IndividualSubmitter
GroupSubmitter
+ courseGroup: CourseGroup
1?
CourseAssignment
+ id: String
+ title: String
+ description: String
+ dueDate: Date
+ avaliableDate: Date
+ untilDate: Float
+ points: Float
+ weight: Integer
+ extensions: Object
+ displayGradeAs: PointOptions
+ countTowardFinalGrade: Boolean
+ allowMultipleSubmissions: Boolean
+ allowLateSubmissions: Boolean
+ groupAssignment: Boolean
+ hidden: Boolean
+ templateGroup: Group
+ submissions: Submission[]
1
1
<< enumeration >>
PointOptions
Points
Percentage
Complete/Incomplete
Letter Grade
Not Graded
<< enumeration >>
SubmisionStatuses
Not Submitted
Submitted
Graded
DocumentSubmissionAssignment
+ submissions: DocumentSubmission[]
CommentSubmissionAssignment
+ submissions: CommentSubmission[]
Submission
+ submitter: Submitter
+ submissionStatus: SubmissionStatus
+ mark: Float
+ submissionTime: DateTime
+ assignment: Assignment
+ currentSubmission: Group
+ pastSubmissions: Group[]
+ group: Group
1
0..n
DocumentSubmission
CommentSubmission
Group
+ owner: Users[]
+ templateGroup: Group
+ pdf: File
+ annotations: Annotation[]
0..1
Annotation
TextAnnotation
+ comment: String
AudioAnnotation
+ audio: File


Instructor creates Assignment of type:
• Students submit a pdf document
• Instructor submits a pdf and students annotate
Students submit a pdf document
Assignment is added to database
DocumentSubmission created for each submitter
Submitters viewing the assignment would see a file upload page
Group created when student submits
Students annotate a pdf document
CommentSubmission created for each submitter
Group created for each CommentSubmission
Assignment is added to database
Submitters viewing the assignment would see a group editing page

Moving Forward

There is a notion board that currently holds all of the outstanding comments and issues of the system. Please email austinkobayashi@gmail.com or ask Dongwook to request access to the notion page