Group 3 - Second Iteration

yAMUN, a course planning system for MUN students

Jack Harrhy, 201732922
(jaharrhy@mun.ca)
Genadi Valyeyev, 201921376
(gvalyeyev@mun.ca)
Aidan Langer, 201735677
(aplanger@mun.ca)

1 Functionalities

In our previous iteration, we stated our functionalities as thus; functions of base level importance are creating and editing personalized course schedule(s). That includes adding and deleting courses from said schedule(s), along displaying basic information that accompanies said courses within the schedules.

Secondarily, we intend to fetch detailed course descriptions, pre-requisites, and the professor's contact information from various APIs.

Thirdly, we intended to integrate searching functionalities, as well as many optional filters to assist in the searching functionality. When adding a course, if time slots conflict we intend to let the user know, and have them choose how to proceed.

Our last few functionalities listed out in the previous iteration involve an ability to add and remove bookmarks for courses to view later, and to be able to export your courses into various calendar applications.

As of the time of this writing, our base level functionalities are implemented and thoroughly tested. Courses can be added and deleted, and you can view their sections and slots, we managed to build integration with fetching courses from banner given a semester, and getting more information about a professor teaching a course, but the missing piece is the part which interfaces with the University Calendar to get course information.

Most of our time was spent building the banner and people directory scraping pieces, and ensuring all of our models and endpoints had plenty of success and fail cases. Filters on courses has been implemented, such as searching by subject, CRN, and course name, etc.. Adding, modifying, and deleting both bookmarks and schedules has been implemented, along with authorization on all of these resources.

Exporting the schedule to an ICS for calendar applications is mostly implemented, although we ran into some issues in which Google Calendar would seemingly reject any events we created, even thought the built in MacOS calendar processed them fine, so we parked that for now until we have time to return to it.

We intend to come back to the piece that bakes the University Calendar into our app, but we can still create what we intended to without it, therefore it was not deemed high priority.

2 Models

The models we have thus far are as follows;

- Banner-Cache: simple model for caching requests to banner, so we don't have to fetch the same page twice in a short period of time.
- Course: the model that defines a course that takes place in a given semester in Memorial's system, including information about itself, its sections, and their courses. Course has two basic functionalities: to find a course by CRN (used when adding course sections to a schedule or bookmarking), and the search functionalities listed earlier that users would use to find courses.
- Schedule: the model that defines a course schedule owned by a specific user, which can also be either public or not. In this model, you can add a course section (using Course's findOneByCrn() function to ensure it exists before adding) and remove a course (in which there is no check on removal in case the course was removed).
- Bookmark: defines the user's list of bookmarked courses, but in the future could be extended to more if required. Much like Schedule, you can add or remove a course with the appropriate similar methods.
- Semester: defines the year, term, and level of a semester, etc... Exists in the models directory, but is a property of Course.
- Section: defines a section of a course, with its unique CRN, who's teaching it, etc.. Exists in the models directory, but is a child of Course.
- Slot: defines a slot that takes place within a section, such as one it beigns, or what room its in, etc.. Exists in the models directory, but is a child of Section.
- User: the model that defines a user of our system. It has two properties; username and passwordHash.

There is a chance that we've missed out some methods / tidbits about these models, since there are many of them, and they have quite a few static and instance methods to cover.

We're using a mostly embedded system, in which Course is overloaded with all information about itself, but other models like Schedule and Bookmark

are separate to course, and reference their owners User entry via ObjectId references, and mention courses by simply CRN, so we have a mix of embedded and normalized.

We don't do references by ObjectId from things to courses, since we can nuke and rebuild the Course collection to our hearts desire, the schedules and bookmarks will stay valid even if the courses is removed and added back, since the CRN is assumed to be the same. If a course for some reason is removed, it is planned to simply be a 404 to the user, so our frontend can just display that they have a schedule entry / bookmark that no longer exists, and remove it, since our adding logic checks to see if things exist, but our removal does not.

3 Routes and Controllers

This is the same code from our project, all of the route definitions along with the controllers they invoke.

```
// courses - searching for courses by query param filters,
   ↪ and finding specific courses by crn
app.get("/courses", acw(coursesController.search));
app.get("/courses/:crn", acw(coursesController.courseByCrn))
   ↪ ;

// people - getting more information regarding professors
   ↪ who teach courses
app.get("/people", acw(peopleController.search));

// users - endpoints for creating users, users knowing their
   ↪  username when logged in,
// and the ability for users to log in and out
app.get("/users", acw(usersController.getInfoAboutSelf));
app.post("/users", acw(usersController.create));
app.post("/login", acw(usersController.login));
app.get("/logout", acw(usersController.logout));

// schedules - endpoints for users to create, modify, view,
   ↪ and delete their
// potentially many schedules, and to access other peoples
   ↪ schedules
```

```
app.post("/schedules", acw(schedulesController.create));
app.get("/schedules/:scheduleId", acw(schedulesController.
   ↪ getById));
app.put("/schedules/:scheduleId", acw(schedulesController.
   ↪ updateMeta));
app.put("/schedules/:scheduleId/:crn", acw(
   ↪ schedulesController.addCourse));
app.delete(
  "/schedules/:scheduleId/:crn",
  acw(schedulesController.removeCourse)
);
app.delete("/schedules/:scheduleId", acw(schedulesController
   ↪ .delete));

// bookmarks - endpoints for viewing, adding, and removing
   ↪ course bookmarks
app.get("/bookmarks/courses", acw(bookmarksController.
   ↪ getCourseBookmarks));
app.put(
  "/bookmarks/courses/:crn",
  acw(bookmarksController.addCourseBookmark)
);
app.delete(
  "/bookmarks/courses/:crn",
  acw(bookmarksController.deleteCourseBookmark)
);

// exports - currently jsut exporting to ics, work in
   ↪ progress
app.get(
  "/export/schedules/:scheduleId/ics",
  acw(exportsController.scheduleToICS)
);
```

4 Testing

We have 70 tests as of the writing of this document, and likely will add more if we find any more edge cases with our API while building the frontend. There are both controller tests for every controller, and model tests for every model. 'setup.spec.ts' defines some global before and after functions, that setup up a connection to MongoDB and populates test data that'd a smaller hard coded subset of the real data, and tears down the connection afterwards.

Chaijs (https://www.chaijs.com/) is our assertion library, and supertest (https://www.npmjs.com/package/supertest) is what we use for API tests, since it allows you to make API tests without actually making fetch requests, since it wires directly into the express app instance itself.

As see the image below, we've got quite decent coverage as well, with 90 percent coverage on statements, and 75 percent coverage on branches.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	89.3	75.12	85.37	89.23	
backend	100	100	100	100	
config.ts	100	100	100	100	
backend/api	94.34	88.89	85.71	94.06	
auth.ts	95.83	83.33	100	95	27
errors.ts	85.71	83.33	100	85	28-30
index.ts	95.56	100	66.67	95.56	98-99
utils.ts	100	100	100	100	
backend/api/controllers	92.2	78.57	95.65	92.61	
bookmarks.ts	92.86	50	100	92.86	18,43
courses.ts	100	100	100	100	
exports.ts	95.12	85.71	100	97.5	51
people.ts	93.33	75	100	93.33	25
schedules.ts	80.36	66.67	83.33	80.36	115-132,149,166
users.ts	100	100	100	100	
backend/database	89.19	37.5	66.67	87.88	
index.ts	89.19	37.5	66.67	87.88	26,41,45,63
backend/models	89.76	68.66	94.74	89.57	
banner-cache.ts	100	100	100	100	
bookmark.ts	100	100	100	100	
course.ts	65.91	52.5	75	65.91	106-111,145-176
people.ts	100	100	100	100	
schedule.ts	96.55	75	100	96.43	64
section.ts	100	100	100	100	
semester.ts	100	100	100	100	
slot.ts	100	100	100	100	
user.ts	95.45	87.5	100	95.45	38
backend/scrape/banner	83.65	78.13	33.33	84.31	
fetch-data.ts	46.67	0	0	50	16-47
index.ts	66.67	100	0	60	9-10
insert.ts	81.82	100	50	81.82	10-11
parse-data.ts	96.92	83.33	100	96.92	65,102
parse-page-data.ts	57.14	100	0	57.14	8-10
backend/scrape/banner/regular-expressions	100	100	100	100	
course.ts	100	100	100	100	
section.ts	100	100	100	100	
slot.ts	100	100	100	100	
backend/scrape/people	66.67	25	66.67	61.76	
index.ts	66.67	25	66.67	61.76	69-70,78-79,108-118
backend/utils	88	70	100	86.96	
ajv.ts	87.5	75	100	87.5	18-19
misc.ts	88.89	66.67	100	85.71	4

5 Extra

This section is for outlining things outside the scope of the requirements, but are added here for context on decisions made for the project, such as

why we're using TypeScript, and the packages we're importing / over all rational.

TypeScript was picked due for the pros of static typing for most things, rather than the usual dynamic typing of JavaScript, that can be quite error prone; any argument can be handed to any function, but within TypeScript you can't do that. You also get very nice editor integration as well, so a nice added bonus in that department as well.

As for packages:

Dependencies:

```
"ajv": "^7.0.4", // for validating entry from users, and
   ↪ converting them to TypeScript interfaces
"bcrypt": "^5.0.1", // for handling passwords, both
   ↪ hashing / salting, and verifying hashed passwords
"connect-mongo": "^4.2.2", // for handling session tokens
"cookie-parser": "^1.4.5", // parsing cookies from the
   ↪ user
"debug": "^4.3.1", // outputting verbose debug information
   ↪  during development
"dotenv": "^8.2.0", // reading a file called `.env`,
   ↪ allowing to change the defaults
"eol": "^0.9.1", // when outputting the .ics file, have
   ↪ the correct line endings, by the .ics spec
"express": "^4.17.1", // the api
"express-session": "^1.17.1", // handling user sessions in
   ↪  express
"ics": "^2.27.0", // outputting ics files
"jsdom": "^16.4.0", // reading banners website, and
   ↪ pulling elements from it
"mongodb": "^3.6.3", // the database connection
"mongoose": "^5.11.14", // what we used for managing
   ↪ models / validation / overall interfacing with
   ↪ mongodb
"node-fetch": "^2.6.1", // fetching data from MUN
"rrule": "^2.6.8" // creaiting rrule strings, repeating
   ↪ rules that are used in generating ics files
```

As for dev. dependencies, lots of TypeScript type definition imports from the `@types` repo, and utils for testing.

6 Running our app

When it comes to running the application, we've personally been testing it on the latest version of node (v15), and potentially using some features that might not even work on node versions less than 12. Even though we are using TypeScript, ‘npm install‘ followed by ‘npm start‘ should be all you need, the only expectation the app makes is that it can connect to your MongoDB instance locally without authentication, and create a database named ‘yamun‘ and ‘yamuntest‘, if these values must be changed, ‘.env.dist‘ can be copied to ‘.env‘, and the values can be changed there.