

Documentation for the Front-End of MathBotics

Front-End Project Structure:

The MathBotics FrontEnd is a React based frontend and was created using CRA(CreateReactApp), therefore it has a familiar folder structure.

- public: Contains index.html which will serve up the page rendering a single element by the id of "root"
- src: Where the react code lives
 - components: React Components used within the MathBotics application
 - block: Components related to the blocks
 - course: Components related to the course/courses
 - dashboard: Components related to the dashboard
 - form: Where most of the form components are stored
 - grades: Components related to Instructor Gradebook and Student Grades
 - hoc: Any Higher Order Components should go here ie. withSidebar hoc - icons: Contains icons related to MathBotics
 - lessons: Components related to lessons/lesson
 - slides: Components related to slides/slide
 - students: Components related to students/student
 - graphql: Relay configuration and mutations
 - mutations: Mutations used by components
 - pages: Contains the entry point to all the page components served up by MathBotics application
 - routes: Contains logic for specific routes ie. ProtectedRoute = logic for a protected route - types: Contains the types which aren't defined by a library or the application
 - utils: Helper functions

FrontEnd Installation Guide / Environment Setup:

- Use the docker compose in /docker file to start up the frontend + other related services

FrontEnd User Manual Docs:

- Generating gql type: If you update or add to any graphql query or mutation on the frontend you will have to generate the types, run `yarn relay` at the root of the /frontend directory

NOTE: Version Two of MathBotics should not have an infinite loading screen issue. However, in the case that you encounter this problem, remove the cookie on the browser (CMD + SHFT + C -> Application -> cookies -> clear)

Application Deployment:

We were not able to deploy the application, however the following information was left over from version one of the project.

- Since this is a react application there are tons of guides on packaging and deploying: Ref <https://blog.heroku.com/deploying-react-with-zero-configuration>

NOTE: docker-compose is only meant for developing locally, however you can deploy the application using the compose (<https://www.docker.com/blog/how-to-deploy-on-remote-docker-hosts-with-docker-compose/>). We recommend loosely coupling all the services. This means deploying frontend as static page and deploying backend using docker on herokus docker deployment (<https://devcenter.heroku.com/categories/deploying-with-docker>). Having Postgres hosted on heroku (<https://www.heroku.com/postgres>) and separate from where the backend/frontend are running. There is a deployment diagram you can check out that we included somewhere in all these files/folders

Documentation for the back-end of MathBotics

Back-End Project Structure:

The MathBotics backend is a GraphQL API with ApolloServer to expose the graphql route and handle graphql parsing,

- prisma: Contains the prisma.schema and all the migrations (explanation in the user manual).
- scripts: Contains index.html which will serve up the page rendering a single element by the id of "root"
- src:
 - data: Where the services relating to data retrieval exist.
 - graphql: Where things relating to graphql reside, this is where the "endpoints" and objects reside.
 - context: The context type resides here -- the context is passed with every single request to the resolve function of every single field.
 - mutations: These are fields that change the state of the application (they "do" things rather than "get" things).
 - objects: These are the objects, these are essentially they "types" that can be returned by mutations or queries, or be used as fields in other "types". They may also have computed fields (fields with resolve() functions).
 - queries: These fields are pure functions that simply return the applications state or something constant (these "get" things rather than "do" things).
 - provider: The services reside here, APIs for interacting with external libraries or application to simplify the code within the graphql resolvers.
 - server: The server configuration and bootstrapping occurs here.
 - middlewares: These process requests either before or after they are passed to the main application logic (the stuff in the graphql folder)
 - express: These process the request at a HTTP level (i.e. Headers).
 - passport.ts: This checks the request for an Authentication token in the cookies, these tokens contain the userId, it then

finds the correct user and attaches it to the context.viewer to be used in the resolvers of all fields.

- graphql: These process the request at a GraphQL level (after the graphql has been validated and parsed).
 - authentication.ts: This grabs the user returned from the logIn mutation and attaches its userId to the request's cookies to be fetched by the passport middleware.
 - permissions.ts: This uses GraphQL shield to set field-level permissions. At the time of writing, there are no permissions here, so the entire API may be accessed by any authenticated actor, this must be changed to only allow access to certain fields to certain actors.
- factories: Code to create database entities.
- seeders: Scripts to run code that creates database entities.
- tests: Code to test other code. Run yarn test to run the tests.
 - integration: Code to test the entire feature of the backend from top to bottom (query/mutation to response).
 - unit: Code to test specific functions of the application. Please read <https://testing.googleblog.com/2014/03/testing-on-toilet-what-makes-good-test.html> and abide by it before writing tests. Bad tests are worse than no tests. In general, a unit test should test a SINGLE path down a function, try to maintain this for integration tests as well.

Installation Guide / Environment Setup:

- Use the docker compose in /docker file to start up all services.

User Manual Docs:

- Before creating mutations or queries: We use nexus-prisma-plugin, which exposes very basic "CRUD", or Create-Read-Update-Delete, mutations and queries to reduce the amount of boilerplate code. When you find yourself needing to make a mutation or query involving some CRUD action, please first check to see that you cannot simply generate it using this plugin. (There are examples of doing this in the queries/index.ts as well as the mutations/index.ts in the extendType portion)
- Generating gql type: If you add any mutation, query, or object, make sure that it is exported at the queries/index.ts or the objects/index.ts or the mutations/index.ts. Afterwards, run yarn gen in the backend folder.
- Migrating Database (Prisma): When making changes to the schema, edit the prisma.schema in the prisma folder. Afterwards, docker exec into the running api container (via docker exec -it mathbotics_api bash) and run yarn migrate:save, fill the required info, this will create a new migration with the changes made to the schema, then run yarn migrate:up to apply that migration. Then, on your own machine, run yarn gen in the backend folder to generate a new version of the prisma client reflecting the new changes.
- Seeding, creating users: Run yarn seed src/seeder/AdminSeeder.ts to create an admin, or for other types as well.

Deployment:

We were not able to get to deploy the application, but these are the instructions that were left over from version one of the project:

- Here are some Heroku guides to deploying the application, the former may be more interesting since we can leverage the Dockerfiles we've created already
<https://devcenter.heroku.com/categories/deploying-with-docker>,
<https://devcenter.heroku.com/articles/deploying-nodejs>
- **NOTE:** docker-compose is only meant for developing locally, however you can deploy the application using the compose

(<https://www.docker.com/blog/how-to-deploy-on-remote-docker-hosts-with-docker-compose/>). We recommend loosely coupling all the services. This means deploying frontend as a static page and deploying backend using docker on heroku's docker deployment (<https://devcenter.heroku.com/categories/deploying-with-docker>). Having Postgress hosted on heroku (<https://www.heroku.com/postgres>) and separate from where the backend/frontend are running. There is a deployment diagram you can check out that is included somewhere in all these files/folders