
Table of Contents

Introduction	1.1
Folder Structure	1.2
Config	1.3
Getting Started	1.4
Data Structure	1.4.1
Import Demo Data	1.4.2
Server and Resolvers	1.5
Frontend Implementation	1.6
Auth	1.6.1
Development	1.7
Deployment	1.8

Headless GraphQL Introduction

Want to build a website like [Gumtree](#), with Headless you can easily do that.

Headless GraphQL is a Buy & Sell application where you will find everything that is needed for **a complete application**. In this application we have used **Firebase as database, Hosting, Deployment solution, GraphQL as Server/Backend Api, Next.js as Server Side Ready Client Side/Frontend**. This application is built for the user whom are looking for a whole application not just a Frontend template or a backend solution, Also the user want to learn how to built an application using **GraphQL, Nextjs, Firebase**. This Application can be your next `MVP project` you can easily use this application to create a Buy & Sell service.

Headless Application Provides The Below Feature which is needed to built a BUY & SELL service:

1. **Firebase Database Integration.**
2. **Smart Searching and Filtering.**
3. **Location Based Searching and Filtering.**
4. **Regular Authentication & Mobile Authentication.**
5. **Social Sharing**
6. **Ad Post, Publish Post, Draft Post**
7. **Nearest Post**
8. **SEO Friendly**
9. **Server Side Rendering**
10. **Favorite Post.**
11. **User Profile Page.**
12. **Category Page**
13. **Recent Post Page**
14. **Nearest Post Page**
15. **Dedicated Search Page.**
16. **Single Post Page.**
17. **User Settings Page**

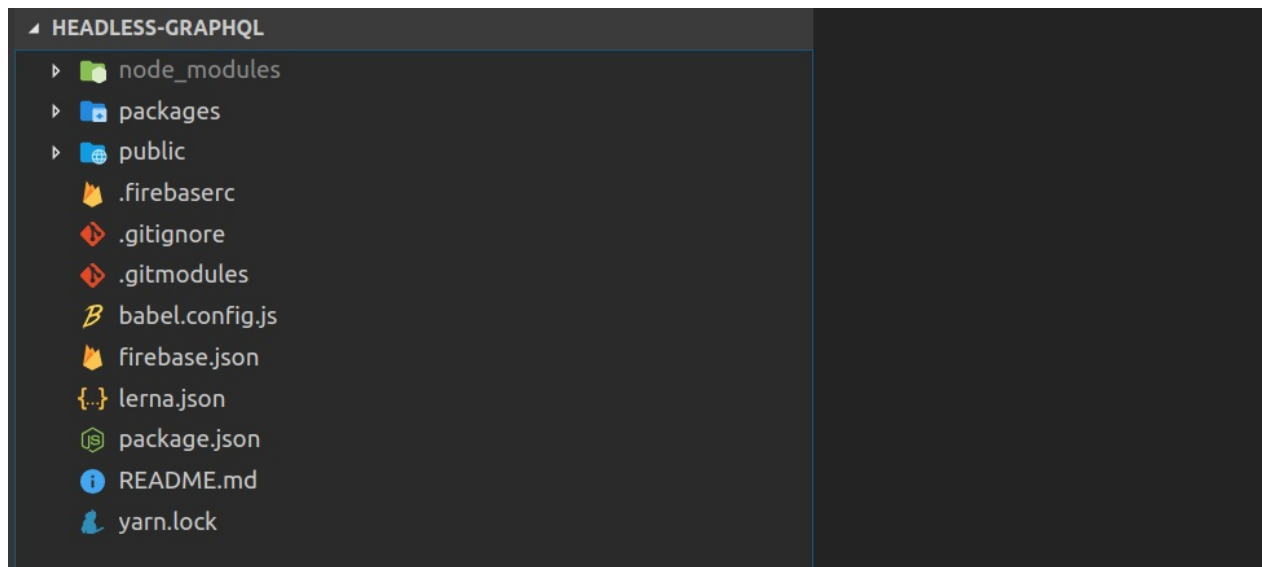
And Most importantly **Easy Deployment**, Everything (GraphQL, Node.js, Firebase Functions, Apollo, Next.js) has been deployed into Firebase, So with our deployment solution you don't have to take any headache for deploying separate servers. Check our deployment Section to know how easy it is to deploy.

Now, we will discuss about this application in whole, how we have developed it, technologies being used, how the user can use it,

The whole application is built on top of Monorepo Architecture using Lerna(<https://github.com/lerna/lerna>) And Yarn. you will get details about the Folder Structure in the Next Section.

Folder Structure

The folder structure of Headless GraphQL is following like that.

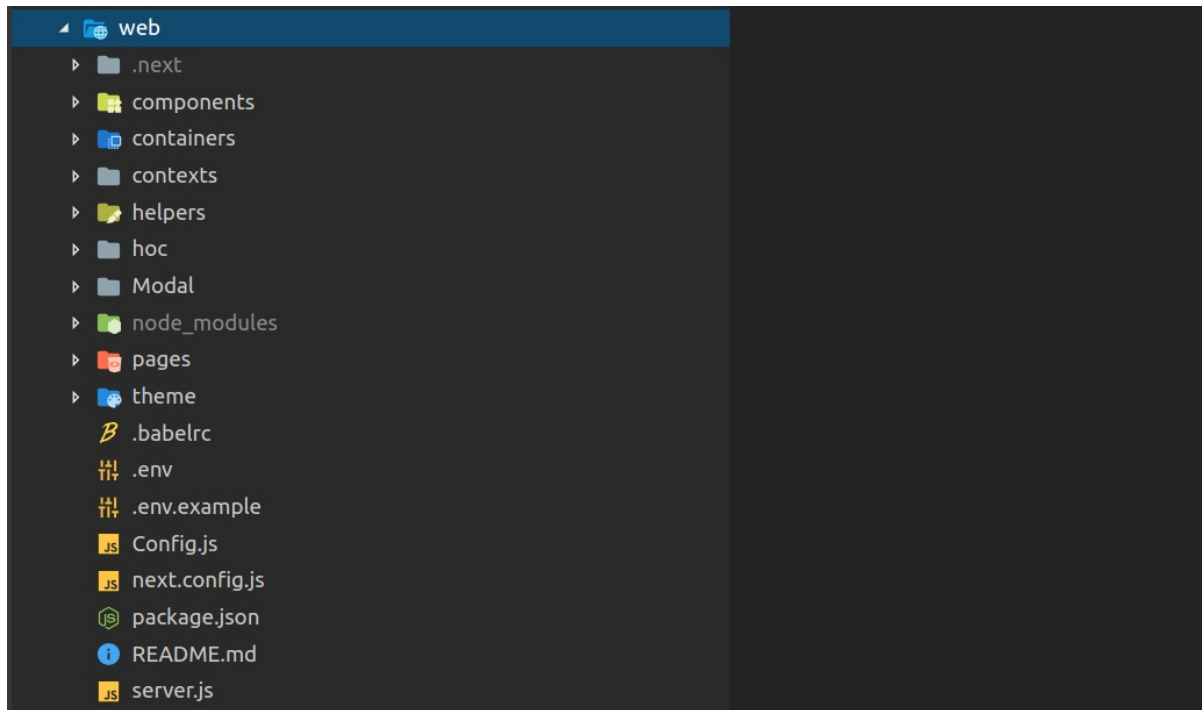


node_modules: It contains all the npm packages that is used on this projects.

packages: It's came from mono repo concepts. It's contain all kind of module which gave the share able module structure.



- **core:** Contain Project Specific components.
- **importer:** Contain demo data import functions.
- **reusecore:** Contain Reusable Components.
- **server:** Contain Server Related File and Functions.
- **web:** It's the core of the project front-end.



(web module):

- **components:** contain the compose components for this project.
- **containers:** Collections of page specific components.
- **context:** components which is build using react context API.
- **helpers:** contains all kind of helper functions like: init-apollo, urlHelpers etc .
- **hoc:** Contain's component which is build using react Higher Order Components pattern Like: AppLayout etc.
- **pages:** In next.js the file-system is the main API. Every `.js` file becomes a route that gets automatically processed and rendered. so all the route file is here.
- **next.config.js:** contain extended next configurations like css module support, font import, image import and optimization and env variables.
- **package.json:** Contains all the informations about the specific module packages, scripts etc .

public: Contains public files used on the projects like icon files.

package.json: Contains all the informations about the workspace like third party packages, scripts etc .

lerna.json: Contains lerna and yarn workspace config.

Config

Before running the project you need to setup or config some variables in `.env` . For the configuration follow those steps:-

Step-1: Rename our provided `.env.example` to `.env`

Step-2: After renaming please provide you own `firebase` regarding configurations .

Getting Started

Settings up Configs

1. First of all navigate to `packages/web/.env` , `packages/server/.env` and `packages/importer/.env` and set your own `firebase` regarding configurations.
2. Then go to `packages/server/config.js` (this is firebase service account which uses the admin sdk you can get it from your project settings -> service account -> generate new private key -> you will get a downloaded file -> copy it's content and paste it to the `config.js` inside the `module.exports`

`GRAPHQL_URL_STAGING` and `GRAPHQL_URL_PRODUCTION` will be needed for server deployment. You will get those urls after deployment. Check deployment section for further details.

First make sure you have set all the config like Google MAP API KEY, Firebase Config.

GOOGLE_API_KEY is google map api key you will get it from here

<https://developers.google.com/maps/documentation/javascript/get-api-key>

GRAPHQL_URL_LOCAL= <http://localhost:4000>

GRAPHQL_URL_STAGING=<http://localhost:5001/headless-graphql/us-central1/api>

(optional to run in local dev)

GRAPHQL_URL_PRODUCTION= you will get it once you deploy your app into firebase

(optional to run in local dev)

APP_FRONT_END_URL=<http://localhost:3000>

CURRENCY=\$

For **Firestore** configs you need to create a firebase app with firestore database. and you can get the config from there

If you have setup these then you can run the below command in the root folder to start the project.

```
yarn
yarn start
```

Then web server will run in `localhost:3000` and graphql playground will run on `localhost:4000/playground`

Navigate to `localhost:3000` in the browser to see the site.

| For data check import demo data section

Data Structure

In our product we have used three document/table

1. posts
2. category
3. users

Posts

```
{
  id: ID
  title: String
  slug: String
  content: String
  status: String
  author: Author
  authorId: ID
  image: Image
  price: Float
  location: Location
  formattedLocation: Location
  distance: Float
  categories: [Category]
  gallery: [Image]
  isNegotiable: Boolean
  favouritedBy: [String]
  condition: Boolean
  related(limit: Int): [Post]
  contactNumber: String
  createdAt: Date
  updatedAt: Date
}
```

Category

```
{
  id: ID!
  slug: String
  name: String
  value: ID
  label: String
  image: Image
  posts(limit: Int, page: Int ): PostWithCount
  createdAt: Date
  updatedAt: Date
}
```

Users

```
{
  id: ID
  email: String
  username: String
  mobile: [Mobile]
  image: Image
  address: String
  website: String
  name: String
  posts(limit: Int, page: Int): PostWithCount
  favourite(limit: Int, page: Int): PostWithCount
  draft(limit: Int, page: Int) : PostWithCount
  createdAt: Date
  updatedAt: Date
}
```

Import Demo Data

We have provided our demo data for customer testing purpose. To import demo data you have to configure few things.

Configure the Importer

If you check our folder structure then you will see there a folder named `importer` in `packages` folder. In the root folder `packages/importer/packages/importer/headless-graphql-firebase-config.json`

file is your firebase config json. Please download your firebase json from firebase console and rename it to `headless-graphql-firebase-config.json`

Check the below image for sample config file

```
{
  "type": "",
  "project_id": "",
  "private_key_id": "",
  "private_key": "",
  "client_email": "",
  "client_id": "",
  "auth_uri": "",
  "token_uri": "",
  "auth_provider_x509_cert_url": "",
  "client_x509_cert_url": ""
}
```

NOTE:

1. Make sure that you put your own `firebase` regarding config to `.env` file.
2. Also make sure you change the code in the `packages/importer/packages.json` file. In the `sync` script you have to change `headless-graphql` with your project name. Check the below image.

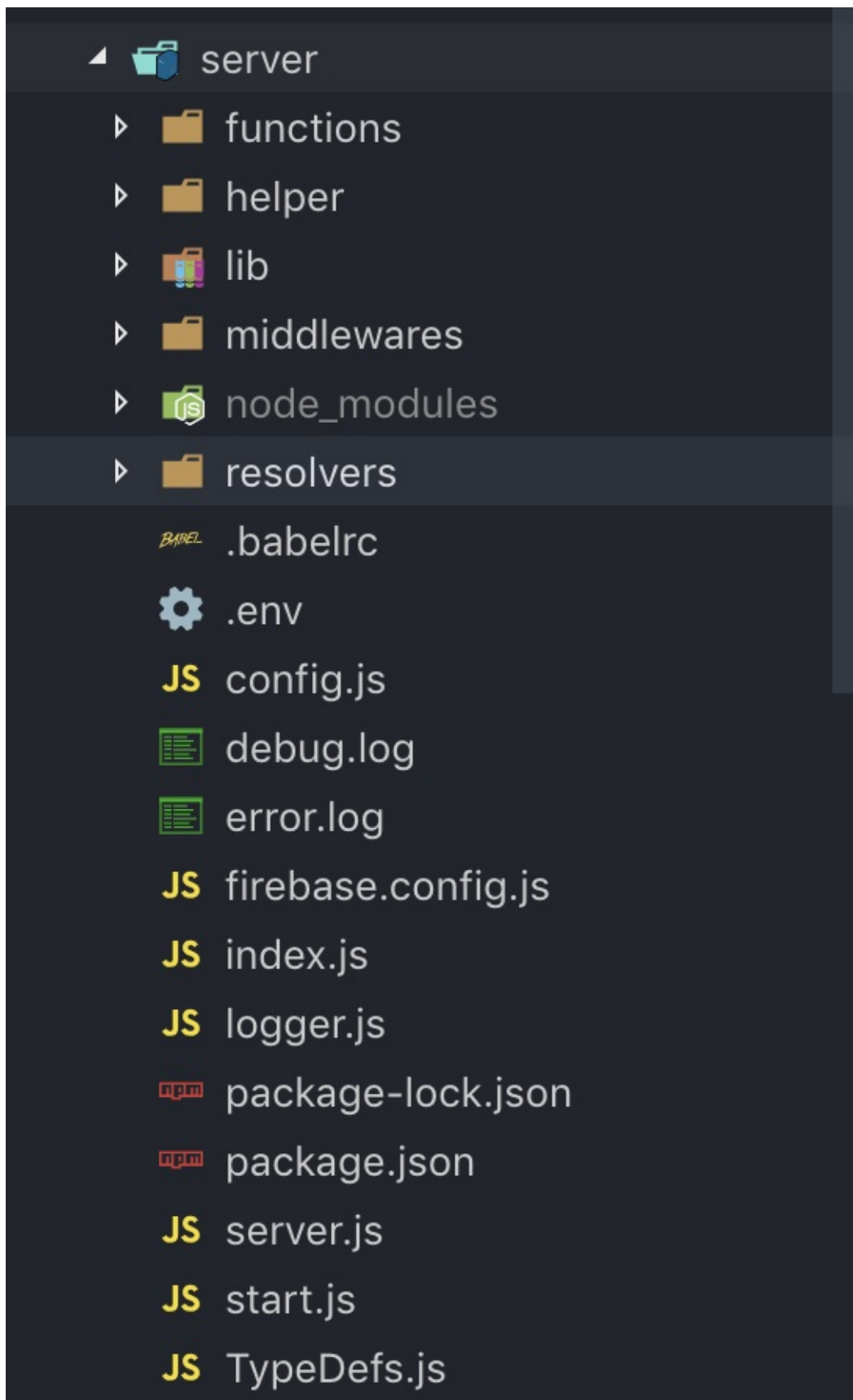
```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "sync": "firebase firestore:delete --all-collections --project headless-graphql && node import.js posts.js",
  "export": "node export.js posts && node export.js category && node export.js users"
},
```

After configuring this you can import demo data. For that you have to run the below command in the `packages/importer` folder.

yarn sync

Server & Resolvers

We have used graph-ql as graphql server. All the server code is reside in `packages/server` folder. Check the below image for server folder structure



Now let me wal you through the whole process.

TypeDefs.js

It is the file where we have defined all our graphql types, query and mutations and input

Resolvers

Here is the functionality to fetch and put data to firebase.

Middlewares

Here we have restricted the routes for unauthorized access.

lib

In this folder we have written all the function to access the firestore database. In the resolvers we have used this function to fetch or put data in the firestore.

In the helper folder there are all the helper function and in functions there is a google cloud function to make a thumb of the image during upload.

Frontend Implementation

Headless Frontend Section [user view] basically build based on this procedure.

1. Pages are built according to Next JS paradigm.
2. Each pages are build on some Containers
3. Each Container hold on some components
4. Components render the data and perform their tasks
5. Datas are fetched via the GraphQL queries.

Now, Pages, Containers & Components related information you can grab via the project folder structure discussed before. Let's make a dive into the Data fetching using GraphQL queries and some very related tasks.

For data fetching we are using **useQuery** hook from `@apollo/react-hooks`

For example :

let's discuss here a query to fetch post data and display on the Home grid.

GraphQL Query :

```
import gql from 'graphql-tag';

export const GET_ALL_POST = gql`
  query getAllPost($LIMIT: Int, $page: Int) {
    posts(limit: $LIMIT, page: $page) {
      data {
        id
        slug
        title
        price
        image {
          url
          largeUrl
        }
      }
      total
    }
  }
`;
```

Then this query need to be called in the Container where you need to display & process the data.

Now, in the Home container import that query and prepared the data.


```
// pseudo code //

import React from 'react';
import { useQuery } from '@apollo/react-hooks';
import { GET_ALL_POST } from '[FROM-THE-JS-FILE-THE-QUERY-IS-WRITTEN]';

const Home = () => {

  // QUERY SECTION
  let QUERY_VARIABLES = {
    LIMIT: 8,
  };
  const { data, loading, error } = useQuery(GET_ALL_POST, {
    variables: QUERY_VARIABLES,
  });

  // Extract Post Data
  const postData = data.posts ? data.posts.data : [];

  return (
    <React.Fragment>
      <YOUR_HOME_CONTAINER />
    </React.Fragment>
  );
}

export default Home
```

Auth

This Application uses Cookie to store the token, As a complete server side rendering app you need to use cookie. as cookie can be accessible from both server and the client. Now, I will discuss how Authentication works in this application and it's flow.

Basic Login Step:

1. First, Go to the Signin or Signup page
2. provide your email, password or mobile number
3. click on the signin or signup button
4. if your credentials are ok, then firebase will return a token
5. then we have saved this token into the cookie
6. and redirect to the home page as a logged in user

so, we have seen in the above step that how we are getting the token from the firebase and using it as our initial login. to check the code first you need to go to the below file,

packages/web/containers/SignInForm OR packages/web/containers/SignUpForm

```
handleSubmit = async () => {
  setLoading(true);
  if (!Object.keys(errors).length) {
    const { email, password } = values;
    const provider = 'password';
    const { user, error } = await AuthHelper.login(provider, email, password);
    if (user) {
      token = await user.getIdToken();
      setFieldValue('token', token);
    } else if (error) { ...
    }
  } else { ...
  }
};
```

in the above file we have used `Formik` to handle form data and then used our `AuthHelper` to send this data to firebase. Look at the **handleSubmit** function, after doing the request firebase will return the **user** which have use to get the token. Then we have set the token using **setFieldValue** which will revoke the `useEffect` hook and set our token using the **setFirebaseCookie** function.

```
useEffect(() => {
  (async function() {
    try {
      if (values.token) {
        const res = await loginMutation();
        if (res && res.data && res.data.login) {
          setLoading(false);
          const user = res.data.login;
          setFirebaseCookie('id_token', token);
          setFirebaseCookie('user', { ...user });
          redirect({}, '/');
        }
      }
    } catch (error) {
      setLoading(false);
      return error;
    }
  })();
}, [values.token]);
```

so, now you have your token in your cookie and you are in the home page as a logged in user. but wait, theres more to it. what we have just done is just frontend authentication. but we need to check our protected page (profile, add post) and validate the user if they are the right user to access the page, also when adding new post or doing any changes in the account page we need to validate the user in the backend as well.

So, Now I will discuss this two section, How we have handled our protected page and how we have validate a user in the backend/admin section.

Protected/Restricted/Secret Page:

If a visitor comes to your site and not logged in yet, you will not want them to access any Profile, Account Settings, Add Post page as this page should only be available to your site users or members. The page is available in the menu (except the profile/account settings page) but when a visitor try to go to this page they will be redirected to the login page. We have create a HOC you will find it in the `packages/web/hoc/secretPage.js` here we have checked if the token available in our cookie using the context of `getInitializeProps`. if true then the user is logged in other wise the user can't access the protected page.

```

5  /**
6   * HOC for all the secret route
7   */
8  export default ComposedComponent =>
9    class SecretPage extends Component {
10     static async getInitialProps(context) {
11       const token = getFirebaseCookie('id_token', context);
12       let user = false;
13       user = getFirebaseCookie('user', context);
14       const isLoggedIn = token ? true : false;
15       if (!isLoggedIn) {
16         redirect(context, '/signin');
17       }
18       return {
19         isLoggedIn,
20         userId: user ? user.userId : false,
21         email: user ? user.email : false,
22         error: user ? user.error : false,
23       };
24     }
25     render() {
26       return <ComposedComponent {...this.props} />;
27     }
28   };

```

This hoc secretPage is applied to the `packages/web/pages/add-post.js`

```

const AdPostPage = props => (
  <>
    <PageMeta title="Add post" description="Add post" />
    <AddPostProvider>
      <AddPost {...props} />
      <Modal />
    </AddPostProvider>
  </>
);

export default SecretPage(withLayout(AdPostPage));

```

Backend/Admin Validation Check:

Once we have set our token into the Cookie, the server will always get access to this token using our apollo configuration.

we have followed the below docs from Apollo,

<https://www.apollographql.com/docs/apollo-server/features/authentication>

<https://www.apollographql.com/docs/react/recipes/authentication>

and here's how we have done it in the code,

first go to the `packages/web/helpers/directory` look at the **apollo.js** file.

we have send the token to the apollo graphql server, now we need to validate this token in the backend, here how we have done it,

1. first we have retrieve the token in the packages/server/server.js file (for deployment we have used the index.js)

```
// context
const context = async req => {
  const token =
    req.request && req.request.headers && req.request.headers.authorization
      ? req.request.headers.authorization
      : '';
  const user = await getUser(token);
  const isVerified = user && user.userId ? true : false;
  return {
    req,
    token,
    user,
    isVerified,
    pubsub,
    firebaseAdmin,
  };
};
```

2. the getUser function validate this token using firebase admin token verifyIdToken function. To identify a user is owner, same user, admin of a post we have used GraphQL Schield (<https://github.com/maticzav/graphql-shield>) as middleware.

That's it, This is the whole Authentication flow of this application.

Development

We have provided both the Development and Staging support for this application.

Make sure you have

Node (<https://nodejs.org/dist/latest-v8.x/>)

NPM (<https://docs.npmjs.com/>)

&

Yarn (<https://yarnpkg.com/en/docs/install>)

installed in your system.

Initially Run the below command to install the dependencies:

i) yarn

Development Mode:

Just run the below command to run this application in the development mode, most of the time you will work on the development mode to develop or customize this application.

To Run The Frontend:

```
yarn web
```

To Run GraphQL Api:

```
yarn server
```

Staging Mode:

Staging mode is used to check if everything is alright before you push it to the production server or deployment. Here are the command needed to run this application in Staging mode,

```
i) yarn web:staging  
ii) yarn server:build  
iii) yarn serve
```

This application is built on top of monorepo architecture check more on this in the Deployment section. you can find all the above command that we discussed here in the root package.json file scripts section.

Important Notes:

Make sure all the dependency you have in your `packages/web/package.json` file also need to be available in `packages/server/package.json` file. This is necessary because server package converted to cloud package which is eventually deployed to firebase so `packages/server/package.json` file is same as `packages/cloud/package.json` . In the cloud directory there is next directory which is copied from the `packages/web/.next` directory. so this is your next build folder.

So, if something is wrong, there might be some package missing in your `server/package.json` file which is available on your web `package.json` file.

Also, every time you change some package or update something or deployed the project it's better to remove the `packages/web/.next` directory and build the web again.

Deployment

Everything (GraphQL, Apollo, Next.js, Node.js, Firebase Functions) has been deployed into Firebase without the need to setup separate servers.

In this section we will discuss about how to deploy this template in Firebase. Just run the below command in your terminal in the root `headless-graphql` directory step by step and you are good to go,

NB: before you run the below command make sure you have installed firebase cli and login into the terminal using `firebase login` command, check this link for details <https://firebase.google.com/docs/cli>. Also make sure you have Node 8.x.x installed in your system otherwise it will not work (<https://nodejs.org/dist/latest-v8.x/>).

Firebase Deploy:

```
i) yarn (if you are running it for the first time)
ii) yarn web:build
iii) yarn server:build
iv) yarn deploy
```

Here I will discuss little bit about the above command, how they are helping our template to be deployed in the firebase.

i) yarn:

As we all know by now `headless-graphql` is built on top of monorepo architecture using lerna (<https://github.com/lerna/lerna>) and yarn workspace. so if you just run `yarn` command in the root directory of this project it will install all the required packages and libraries in your project.

ii) yarn web:build:

we know, In the frontend of this application we are using Next.js (<https://nextjs.org>). So basically what this command do is, prepare the frontend for the production using `yarn build` command in the `packages/web` directory. if this command run successfully you will see a `.next` folder created in the `packages/web` directory.

iii) yarn server:build:

This is another important command to deploy our app in the firebase. Basically this command complete two task,

- first, it will build our api directory for the production mode which is available in the `packages/server` and move it to the `packages/cloud` directory.
- then, it will copy the build `.next` folder from the `packages/web` directory to the `packages/cloud` directory and paste it as `next` folder.

the above task is important for our firebase deployment, as we are deploying both the graphql api and next js in the firebase. and to do this we need to tell firebase that deploy both our graphql and next js as firebase functions. you can see exactly what we have done this into `packages/server/index.js` file.

iv) yarn deploy:

finally, **yarn deploy** command will deploy the cloud directory into the firebase.

After, successful deployment you will get the related url in your terminal, firebase deployment take some time to deploy, the terminal will look something like below

```
✓ hosting[headless-graphql]: file upload complete
i functions: updating Node.js 8 function generateThumbnail(us-central1)...
i functions: updating Node.js 8 function api(us-central1)...
i functions: updating Node.js 8 function next(us-central1)...
✓ functions[api(us-central1)]: Successful update operation.
✓ functions[next(us-central1)]: Successful update operation.
✓ functions[generateThumbnail(us-central1)]: Successful update operation.
i hosting[headless-graphql]: finalizing version..
.
✓ hosting[headless-graphql]: version finalized
i hosting[headless-graphql]: releasing new version...
✓ hosting[headless-graphql]: release complete

✓ Deploy complete!

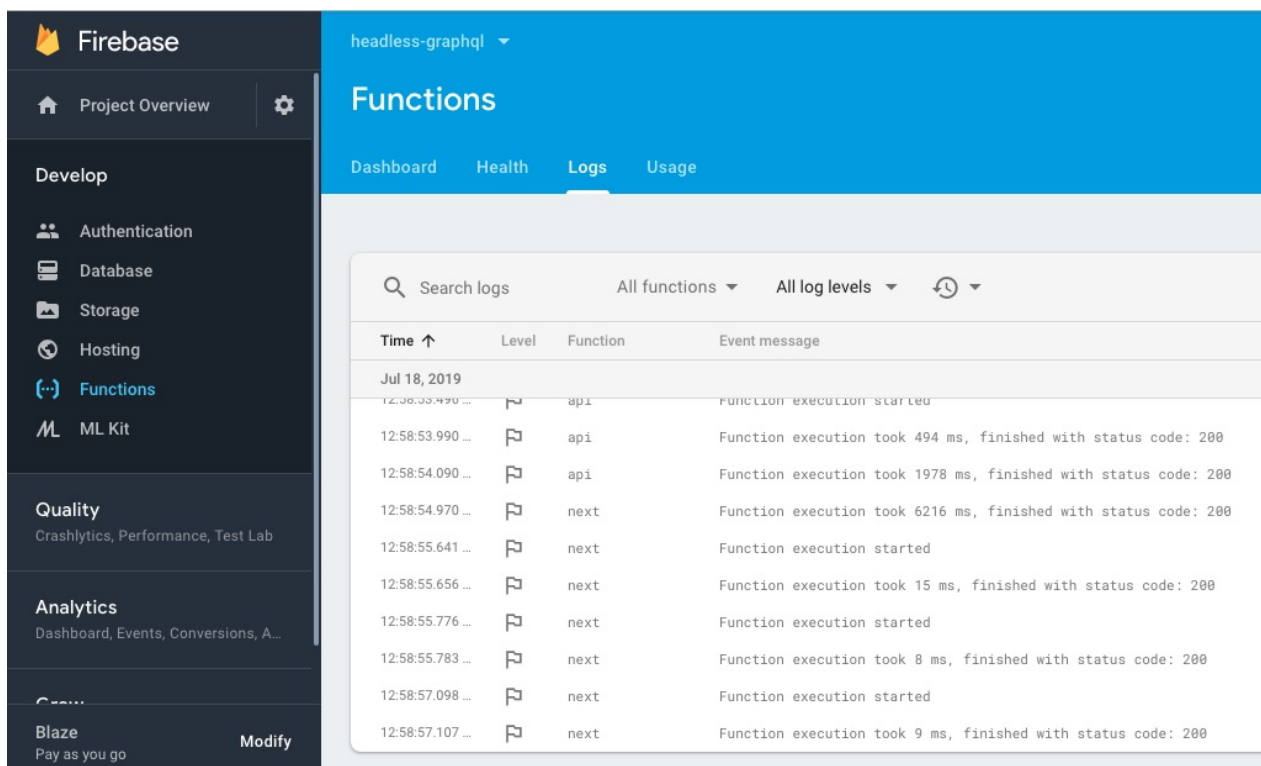
Project Console: https://console.firebase.google.com/project/headless-graphql/overview
Hosting URL: https://headless-graphql.firebaseio.com
🌟 Done in 114.82s.
```

NB: after deployment if you start the development again you may run into an error in your terminal which basically says you have multiple package.json file with same name inside packages/server and packages/cloud directory.

the reason of this issue is, there are two package.json file inside server directory and cloud directory with same name inside the package.json file, so the solution should be change the name inside the package.json file of the cloud directory. you can give it cloud or whatever you want but server name already exist on server/package.json file. as we are converting es6 code into cloud directory so most of the files get copied. you won't have issues with other files just this one.

Debugging:

if you got any error after deploying you need to check the Firebase -> Your project -> Functions -> Logs. you will find necessary information about your error here.



The screenshot shows the Firebase console interface. On the left is a dark sidebar with the 'Firebase' logo and a navigation menu. The 'Functions' option is highlighted in blue. The main content area has a blue header with 'Functions' and tabs for 'Dashboard', 'Health', 'Logs' (which is active), and 'Usage'. Below the header is a search bar and filters. A table displays the logs for the 'api' and 'next' functions.

Time ↑	Level	Function	Event message
Jul 18, 2019			
12:58:53.490 ...	INFO	api	Function execution started
12:58:53.990 ...	INFO	api	Function execution took 494 ms, finished with status code: 200
12:58:54.090 ...	INFO	api	Function execution took 1978 ms, finished with status code: 200
12:58:54.970 ...	INFO	next	Function execution took 6216 ms, finished with status code: 200
12:58:55.641 ...	INFO	next	Function execution started
12:58:55.656 ...	INFO	next	Function execution took 15 ms, finished with status code: 200
12:58:55.776 ...	INFO	next	Function execution started
12:58:55.783 ...	INFO	next	Function execution took 8 ms, finished with status code: 200
12:58:57.098 ...	INFO	next	Function execution started
12:58:57.107 ...	INFO	next	Function execution took 9 ms, finished with status code: 200