Development Guides

- Development Workflow
- Software and Tools
- Front-end Guide
- Back-end Guide

Development Workflow

You’ve been asked to pick up a ticket, what do you do first?

Read through the ticket and confirm you understand what you need to do and what code base you need to operate on. When you’re happy to begin, set the Jira ticket status to “In Progress”.

We use GitHub to host our source code, and use GitHub Actions for our CI/CD.

We are using the Git Feature Branch Workflow - this means that the basic process is:

- Clone the repository e.g:

  ```
  git clone git@github.com:ArkoseLabs/<repo>.git
  ```

  It is recommended to set up an SSH key and clone via SSH. Guides for setting this can be found here, and once that is done you can find the “Clone with SSH” link from the repository in Github
- Create a branch from the default repository branch (eg main) - generally the branch name should be named after a Jira issue number, e.g. for ENG-10000

  ```
  git checkout -b ENG-10000
  ```

- Make changes, commit. Make sure your work will not break any systems.
- Ensure you have appropriate unit tests written to test your new functionality.
- Push up to the origin, in this case that is ArkoseLabs/<repo> e.g.

  ```
  git push -u origin ENG-10000
  ```

- Create a Pull Request
  - Your Title should contain the ticket number and ensure that it’s clear what the PR is
  - Link to the ticket in your description, pasting the URL of the ticket at the top of your description is fine
  - In your description, write some information on what your PR does and anything that your reviewers will need to know to correctly understand what you have done and why
  - Adding comments in the GitHub interface to assist your reviewers in understanding what they are looking at is good too
  - Add Reviewers - at least one. It’s a good idea to find out who are the experts on a particular part of the code and assign them, along with other people to diffuse that knowledge. Start with your squad members and you should slowly get to know experts in certain areas in the coming days / weeks.
  - Add an Assignee - For customer portal this will be @Jason Ramke
- If the PR CI tests do not pass, fix them until they do
  - If the issue seems to be unrelated to the actual issue, still attempt a fix, but ask an owner or lead engineer

Once you’ve got your PR up and you’re happy that you’ve satisfied the above, go back to your ticket and ensure you have met the Acceptance Criteria, then mark it for “**Code Review**”. Make sure you consider the work required for the Quality Engineers to test your work, and document test steps in the ticket as appropriate.

Software and Tools

- Nvm - use nvm to install node and npm
- Git
- Docker
- Docker Compose
- Direnv
- Insomnia - we recommend this for testing graphQL
- Zoom
- Slack
- Your preferred IDE

Front-end Guide

Technologies

The following is a list of frameworks and packages that we use for the front-end, further information about some of these will be provided below.

- React - React
- Material UI - Material UI: React components that implement Material Design
- Apollo Client - Introduction to Apollo Client
- Formik - Formik
- React Router - https://reactrouter.com/web/guides/quick-start
- Webpack - webpack
- React Testing Library - React Testing Library | Testing Library
- Jest - Jest
- Lodash - _ Lodash
- Direnv - direnv – unclutter your .profile

Guide

Javascript

We generally prefer using ES6 styles and functions when writing JavaScript and we have linting rules set up to help with this. We also prefer functional programming patterns and immutable data as much as possible, again our linting will help guide you with the patterns we prefer.

We would like code to be as readable and clear as possible, variable and function names should be descriptive and obvious. Comments should be left when further information is needed about a line of code; however if code is clear and readable, comments can be kept to a minimum.

GraphQL

The server side API component of the system will be a GraphQL service using Apollo (Streamlining APIs, Databases, & Microservices | Apollo GraphQL). More information will be provided about this service at a later date, at the start of this project we will mock any API calls. The front-end will use Apollo Client (Introduction to Apollo Client) to interact with the GraphQL API, more information can be found in the React section below.

React

We would like all React components to be pure functional components and not class based. An example of a functional component is as follows:

```
import React from 'react';

function FunctionalHeader() {
  const greeting = 'This is an example of a functional component';

  return <h1>{greeting}</h1>;
}

export default FunctionalHeader;
```

We would also like to use React Hooks for state and any effects. Local state should be used for a simple state that does not require to be global or passed up the component tree. Local state can however be passed down as props.

A good guide to functional components using React Hooks can be found here: https://www.robinwieruch.de/react-function-component

If a component needs to fetch data from the api we would like to use the Apollo React hooks, this will help with GraphQL implementation and make fetching data easier. More information can be found in the Apollo Client documentation: https://www.apollographql.com/docs/react/api/react/hooks/

When using Apollo React hooks, the “useQuery” or “useLazyQuery” should be used for fetching data and “useMutation” used for mutations.

For global state we will use React context, further information including code examples and helpers will be provided at a future date.

We would like React components to be relatively small and reusable when appropriate, this will allow us to test components more easily and make reuse easy.

UI

Material UI (Material UI: React components that implement Material Design) will be used as our UI framework, if a widget or UI element needs to be used and a material ui version exists we want to use it. A base theme will be set up when designs are available.

For custom styling any UI we will use the material UI hooks api to construct styles, for more information see @mui/styles (LEGACY) - MUI System

Linting

We use prettier and eslint for linting, the repos will have linting rules setup that you must follow. I advise you to set up your IDE to enforce these rules. A lint npm command will also be set up that can be used for checking your code.

We will also have a github action that will run on all pull requests that will check the linting rules against any new code. The pull request will fail if any rules aren’t followed and won’t be able to be merged until changes are made.

Unit Tests

We would like all code that’s committed to also have unit tests. We will have Jest set up and React Testing Library which can be used to easily test React components. We won’t enforce 100% code coverage but we would like all code logic and branching to be tested as much as possible.

For writing tests on React components that use Context we will include some test helpers that mock Context and help with these tests. Apollo React also includes test helpers that can be used (see Testing React components) and api calls can easily be mocked using the Apollo test helper “MockedProvider”.

A “npm test” command will be set up for running tests and tests will also be run by a github action when pull requests are created, if tests fail it will be the responsibility of the developer to fix the tests.

Forms

We will use Formik (Overview | Formik) for handling any forms on the front-end. This will allow us to easily handle form state and validation. When required we should write common reusable form components using Formik that can make form handling and development easier.

Environment Variables

If an environment variable is needed it can be added to the .envrc file in the repo. We use Direnv for setting environment variables in our local development environment. We recommend setting up Direnv and then creating a “.envrc-local” file at the base of the repo which you can use for setting environment variables locally.

If a new environment variable is added to the app it should be documented in the app's README document under environment variables with a default value if required and a brief description.

Directory Structure

A basic directory structure will be set up in the repo however we would like the directory structure to be similar to the following:

- .github
    - workflows
- node_modules
- src
    - {Section_name}Section
        - components
            - {ComponentName}.js
            - {ComponentName}.test.js
        - context
        - utils
            - {utilityFunction}.js
            - {utilityFunction}.test.js
    - common
        - components
            - {ComponentName}
                - index.js
                - index.test.js
        - context
        - services
            - {serviceDescription}.js
        - utils
            - {utilityFunction}.js
            - {utilityFunction}.test.js
- tests
    - helpers
    - mocks

Back-end Guide

Technologies

The following is a list of frameworks and packages that we will use for the back-end.

- GraphQL using Apollo Server - https://www.apollographql.com/docs/apollo-server/
- Knex.js - SQL Query Builder for Javascript | Knex.js
- Objection.js - Objection.js
- Jest - https://jestjs.io/
- Lodash - https://lodash.com/
- Direnv - https://direnv.net/

Database

We'll be using Postgres as the database for the customer portal project, the customer-portal-api repo will contain a docker-compose for a Postgres db instance that can be used for local development. To run this local db just run

```
docker-compose up -d
```

and update your .envrc-local file to include the connection details for that database, by default the following details will be correct:

```
export PG_DATABASE="customer-portal"
export PG_HOST="localhost"
export PG_PASSWORD="password"
export PG_PORT="54324"
export PG_USER="root"
```

To handle the database connectivity and query building from within the customer portal api we'll also be using Knex.js and Objection.js. Objection.js will be used to help us setup relationships between data and help when querying against relations. For any data queries or mutations within the customer-portal we should use the functions and helpers provided by Knex and Objection.

IDs

We would like all database Ids to be setup as UUIDs, we have added a database migration script for installing the required UUID extension for Postgres. An example of how to set the Id as a UUID in a migration script can be seen below, or seen in the migration scripts in the repository.

```
table.uuid('id').primary().defaultTo(knex.raw('uuid_generate_v4()'));
```

Migrations

Knex.js will be used to create and run database migrations and if any database tables need to be added or updated a migration script should be created in the `migrations` directory. It's recommended to read and understand the Knex migration documentation. A migrate command has also been setup to quickly run the latest migrations.

```
npm run migrate
```

Database migrations are also automatically run when using `npm start` and `npm run start:dev`.