NodeJS project structure and best practices [WIP]

This page will explain the standard for NodeJS as a service that being used in several microservices.

Goals

To provide developers with standard boilerplate that follows the design guideline with the infrastructure that Astro has internally and ability to scale the service/application using containerisation and support multi developers per project as a focus.

One of the main principle for the coding style is to have single responsibility, and module isolations for the codes, so multiple developers can work on the same project with loosely couple environment.

Focus

The main focus of this project structure are:

- Service type project
- Full stack type project with SSR (Server Side Rendering) support

Features

The main feature of the style is the language that being used by the coding style, is ES2017 (vanilla javascript) on the server side and babel-6 on the client side.

But, es2017 and babel-6 will work in harmony in a fullstack environment, what this means is mixing babel-6 and es2017 between client codes and server codes should be fine, without causing side effects.

Lint Configuration

To enforce the coding style, eslint must be used by default on every project.

We have created eslint configuration called eslint-config-astro which is a derivation of eslint-config-airbnb-base, that has some changes to accommodate our preferences.

Setup Development Tools

In order to use the boilerplate one should perform the following commands (assuming one has NodeJS installed):\

```
$ npm i -g yo generator-astro
```

After this, you will have astro generator installed on the development machine.

Create a new service

To create a new service, one can perform the following command:

```
$ yo astro
     _-----_
    |       |
    |--(o)--|     Welcome to Astro RESTful
   `---------´          API generator!
    ( _´U`_ )
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `
? What kind of project do you want to create? (Use arrow keys)
 Service (ExpressJS)
  Fullstack (Express JS + React + React Redux + Redux Observable)
? What is your project name? (astro)
? You want to use sequelize? (Use arrow keys)
  yes
 no
? Your API base path? (api)
? Your API version? (v1)
? Your service port? (5000)
? Version number (1.0.0)
? Description (This project generated using Astro Generator)
? Author's name (Suhendra Ahmad)
? License (MIT)
```

Once you generate the service you will have a new project in a folder called astro or any other name that you choose.

Service Structure

The following is the structure for the service only boilerplate.

```
src/
    api/
        v1/
    boot/
        startup/
    config/
    middlewares/
    services/
    utils/
    index.js
```

API

This folder will contains various versions of the api, like let say we have v1 and v2, so on and so forth. It can be maintained within its own folder. So to not mix from version to version.

For one to be able to generate an API, the following command should be performed:

```
$ yo astro:api
? What is your API endpoint name? hello
? Please give API description for documentation! Hello world
? API Method?
 get
  post
  put
  delete
```

After creating an api you will find a new folder called 'hello' inside src/api/v1 will be generated.

One thing to notice that, inside this folder everything you need for that particular api should be there. The following are what contained inside the api folder:

- `src/api/v1/hello/hello.controller.js`
- `src/api/v1/hello/index.js`
- `src/api/v1/hello/hello.validator.js`
- `src/api/v1/hello/hello.spec.js`
- `src/api/v1/hello/hello.integration.test.js`

Notice we have 2 different test files, that is spec and integration.test, why we separate between the two is because integration should not be performed as often as unit test, one should perform it in CI/CD, and it should not be on the same execution step with the unit test.

The question arises of why we put the tests in the same folder with the controller itself not in separate tests/ folder like any other boilerplate. The answer is we want it to be an isolated module, where one module is isolated from other module and the only way to expose the module is via index.js. This will enable multiple developers to work on the same project without dependency to each other. Also give easier access if one need to check or debug certain test for certain api.

Boot

This folder is used for when a service first starting up, sometimes it needs to do some actions only during the startup of the service. For example, a service might need to fetch some string from AWS parameter store to use in the api, if you call parameter store all the time, the api response time will be slower, so to make it faster, you cache the string upon startup. This is where the boot folder comes in.

Also, boot folder will be executed in sequence before server start listening.

To generate a boot, one should perform the following script:

```
$ yo astro:boot
? Boot task name? environments
```

Inside the folder `src/boot/index.js` you will see the following codes:

```
const { startupBoot } = require('./startup');
const { serverBoot } = require('./server');
const { environmentsBoot } = require('./environments');

module.exports = [
  environmentsBoot,
  startupBoot,
  serverBoot
];
```

Pay attention to the module.exports, because the order of the array define the sequence of the boot tasks. If you need the sequence to be different, you can change this array accordingly. Also, this code will be modified every-time you generate new boot code.

Config

This folder will retain all of the configurations across different modules and folders. Inside this folder, by default exist a file called vars.js, where this file, can be used to store any variables across the modules and folders.

Middlewares

This folder contains all express middlewares that one used in the express middleware, the default middleware provided are:

`error`, this middleware will handle any error exceptions thrown by APIs without couple different type of errors, such as validation error, internal server error, etc.

`monitoring`, this middleware will monitor any traffic made to the service or outside the service and sent the result into logger.

One can generate their own middleware using the following command:

```
$ yo astro:boot
? Middleware name? auth
```

A new middleware folder will be generated in src/middlewares/auth.

Services

This part will be containing all workers or external services that needed by the api or any other part of the service code. Example for this, like when one need to call api to another external service to perform certain functionality from within the service itself.

One can generate a new service using the following command:

```
$ yo astro:service
? Service name? userService
```

The above code will generate a service to external user microservice, and create a new folder on src/services/userService. Contained inside this folder are: service code, unit test for the service.

Utils

This folder mostly used for utilities code that shared across different part of the codes, one can put for example APIError utils that will be used by middleware, and API.

One can generate a util using the following command:

```
$ yo astro:util
? Utility name? APIError
```

The above codes will generate a folder for util in src/utils

React.js and Express Fullstack structure

The above details are for service only project using Node.js + ExpressJS. But sometimes, some projects needs to have fullstack configuration for various of reasons. One of the approach that we're going to choose is the SSR (Server Side Rendering). The SSR enables us to have better SEO

implementations and faster rendering and some other benefits compared to SPA (Single Page Application).

Generator Astro supported React.js fullstack code generations using the following stacks:

- Redux
- Redux-Observable
- RxJS
- HMR (Hot Module Reload)
- Enzyme
- react-testing-library (optionals)

There are some other state management library other than redux like mobx state tree, but the reason we used redux, because it has state immutability nature, and wide array of libraries, and some other advantages.

The reason we used redux-observable is because it uses reactive programming approach on handling the state's actions, and this is good to handle or cancel side effect of an action.

Fullstack project structure

The following is the structure for fullstack project

```
src/
    client/
        elements/
        components/
        containers/
        constants/
        ducks/
        hox/
        styles/
    server/
        ...check for service structure
```

Elements

This folder will contains all UI elements. What we mean with ui element is pure stateless component without any route, reducer, epic, etc. The whole purpose of this is to be reused by components inside components folder.

One can generate element as follows:

```
$ yo astro:element
? What is your element name? (NewElement)
   create src/client/elements/new-element/index.js
   create src/client/elements/new-element/style.scss
   create src/client/elements/new-element/new-element.component.js
   create src/client/elements/new-element/__test__/component.spec.js
```

Components

Component, is a reusable building block of a view. A component comprises of multiple elements and even other components as well.

According to component driven design principle, there can be two types of components:

- Stateless(Dumb/Presentational) component
- Stateful(Smart/Container) component

Before explaining them in details, let's talk about two kinds of states that can exist in a component- data and UI.

Data State

Consider a simple component which shows a list of bank names- the bank names are fetched over some API call. Before fetching the data the list of bank names is empty; so your component should show an empty list and once the data has been fetched it should show the names. This is called a data state- where your component can hold any sort of data- preferably data fetched/brought over via some external service.

UI State

To keep your UI interaction behaviour inside your component- this kind of state is used. For example- you clicked on a button and it triggered fetching of the bank names, while the list is being fetched you should not allow user to press on this button again. This kind of behaviour can be tracked via let's say a simple boolean `isListLoading` or something like that. This is an UI state.

Now that we understood the difference between data and UI state, let's talk a bit about two different type of logic as well- business and UI.

Business Logic

Consider a refund component, if I have initiated a partial refund, it should somewhere show me the amount left which were not refunded. This domain/context aware logic is referred as business logic or in some other frameworks as controller logic.

UI Logic

If I press button A, then dropdown B should be opened, loader C should be shown as loading and if the nav bar was opened- it should be closed. Or if a prop value is 10 then the other prop value should be shown as 100 - 10 etc. This kind of logic, which is purely UI behaviour related are referred as UI logic.

Now that, we understood the different types of state and logic a component can have, let's look at the table below to understand what is Stateless and what is Stateful component-

Properties	Stateful/Smart/Container	Stateless/Dumb/Presentational
UI State	Can have	Can't have
Data State	Can have	Can't have
Business Logic	Exists	Can exist (which can be yield without using any state)
UI Logic	Exists	Can exist (which can be yield without using any state)

Since, from the table above it's clear which attributes belong to which kind of component, now let's see some example components.

Stateless(Dumb or Presentational) Component

It's preferable to write stateless component using pure functional approach (not even using lambda functions). The given examples are just for the sake of examples, in real life even a stateless component can have good amount of UI or business logic embedded in it.

And, a component file should only consist of exactly one component function, this is to confirm with Single Responsibility Principle.

stateless-component

```
// a simple stateless component
import locales from './locales';

function Greeter({locale, name}) {
    return <h1>{locales[locale].greeting} {name}</h1>;
}
```

stateless-component

```
// a simple stateless component
import locales from './locales';

// another one
function BalanceBar({spent, total}) {
    const remaining = total - spent;
    const remainingBarStyle = {width: `((remaining / total) *
100).toFixed(2) %`};
    return (
       <div className="balance-bar">
           <div className="remaining green"
style={remainingBarStyle}></div>
           <div className="total yellow"></div>
       </div>
    );
}
```

stateless-component

```
// a simple stateless component
import locales from './locales';

// this dropdown component was extracted from a full fledged react web
application
function Dropdown({items, selectedItem, displayKey, itemType,
ItemComponent, onItemClick}) {
  return (
    <div className="dropdown">
      <button className="btn btn-rounded dropdown-toggle"
data-toggle="dropdown">
        {`All ${itemType}s` || get(selectedItem, displayKey, '')}
      </button>

      <ul className="dropdown-menu">
        {
          items.map(item => getItemComponent(
            item,
            ItemComponent,
            displayKey,
            onItemClick)
          )
        }
      </ul>
    </div>
  );
}
```

Bad practices of defining a Stateless Component

stateless-component-bad-practice

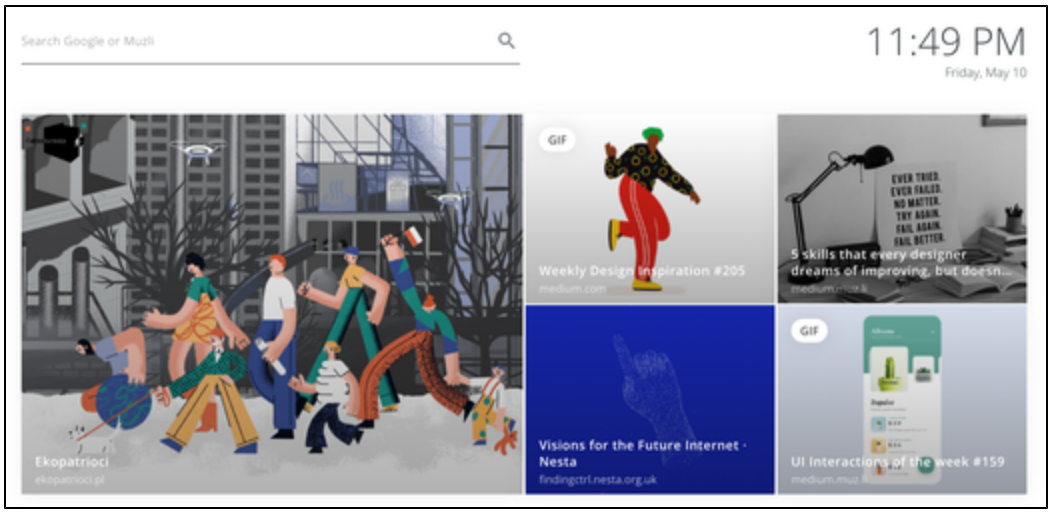
```
class Greeter extends React.Component {
 render() {
       return <h1>{locales[locale].greeting} {this.props.name}</h1>;
    }
}
const Greeter = ({locale, name}) => {
    return <h1>{locales[locale].greeting} {name}</h1>;
};
```

Stateful (Smart) Component

There are 2 different stateful component, container and pure stateful component. The difference lies with the role they are taking. Pure stateful component focus on smaller role, like when you are creating header component that can contain other stateless element. And Pure Stateful component lies in the components folder.

As for container is a stateful component for a much bigger role, like a whole page of the web that contains other pure stateful component or elements combined together.

A stateful component will ideally be composed of multiple stateless components, different elements - will pass them necessary data via props and will have business logic and UI logic to handle interaction among all these presentational components. In component driven design philosophy a stateful component or a container component should comprise an important unit of your view. It complex UI architecture there can be stateful components which renders multiple other stateful components as well. This component is differ with container component, a pure stateful is not a container, but can contain several smaller components. As for container is assume bigger role than pure stateful component.



For example, in the above picture, the whole view can be considered as a container component itself- which is holding the dateview component, search bar component and the gridview component as well. It can be centralised hub of all the data, as in it can fetch the grid contents itself and then pass on to the gridview component, again the gridview can be a stateful/container component which holds it's state to itself and just being used inside this view component. In second case, the view component is holding multiple stateless components(e.g. searchbar component, dateview component etc.) and also a stateful component.

Here's a sample component-

stateful-component

```
// this is also an excerpt of a real life react web application

class FilterPanel extends React.Component {
  constructor() {
    super();
    this._apiService = getInstance('api');
    /**
     * List of data resolution promises for different filter types.
     * @type {{for: string, promise: *}[]}
     * @private
     */
    this._dataResolvers = [
      {
        for: 'partners',
        promise: this._apiService.getItems('partner')
      }
    ];
    /**
```

```
   * `data` contains all the options for different kind of filters.
   * `filters` is the actual state of this filter panel with different
   * selected filter options at any point of time.
   * @type {{data: {}, filters: {}}}
   */
  this.state = {
    data: {},
    filters: {}
  };
}

fetchAll() {
  Promise.all(this._dataResolvers.map(resolver => resolver.promise))
    .then((dataList) => {
      this.updateDataState(dataList);
    });
}

updateDataState(dataList) {
  let update = {};
  dataList.forEach((data, index) => {
    const resolver = this._dataResolvers[index];
    update[resolver.for] = data;
  });

  this.setState({data: update});
}

componentDidMount() {
  this.fetchAll();
}

render() {
```

```
    ...
  }
}
```

> To know more about what should be the coding style for defining stateful and stateless components, you can refer to the AirBnb stylegu ide.

Routing

Since the nature of the code generator is using SSR (Server Side Rendering), so all the requests are made to the server for every page reload before it is passed over to the client side.

For every stateful component, we can add server side route middleware, that can be used for checking certain headers or any server side related logic.

Below is the example of route generated by the code generator.

```
$ yo astro:component
? Select component type? stateful
? What is your component name? Hello
? Component route path? (eg: /astro) /hello
? Do you to generate client route middleware? yes
   create src/client/ducks/routes.js
   create src/client/ducks/reducers.js
   create src/client/ducks/epics.js
   create src/server/routes/index.js
   create src/client/components/hello/hello.route.js
   create src/client/components/hello/hello.epic.js
   create src/client/components/hello/hello.reducer.js
   create src/server/routes/hello/hello.middleware.js
   create src/server/routes/hello/index.js
   create src/server/routes/hello/hello.spec.js
```

The files for routing on the client side is `src/client/components/hello/hello.route.js`, and the file for server side routing middleware is `src/server/routes/hello/hello.middleware.js`

The `hello.route.js` will be generated using lazy loading for code splitting support. And this file reference will be injected in the file of `src/cli ent/ducks/routes.js`. You can find all your routes for all your containers are injected automatically here.

What we are interested in is the server side middleware route file.

```
const middleware = params => (req, res, next) => {  //
eslint-disable-line
  next();
};

export default middleware;
```

This file will be injected automatically into the express middlewares upon every reload. So if you need to add logic to handle the request of certain url path, you can do it here. One should call next() upon successful pass.