Coding Standards and Guidelines

@ pavan.supreeth @ Karthik Ramesh @ dhanush @ vikesh @ Raghavendra @ Mithun Ph @ Aditya Kumar @ Sneha Gangadi

NodeJS Project Guidelines

1. Divide your Solution by Components

- One of the hardest things for larger applications is to maintain a huge code base with tons of dependencies.
- This slows down production and development while adding new features.
- According to Node.js best practices, we should divide the entire codebase into smaller components so that each module gets its own folder, and certain that each module is kept simple and small.
- As a part of Node.js development services, some tried and tested best practices includes development of modular applications by dividing the whole codebase into modular components.
- In this way, we don’t have to share code with others (e.g. APIs, services, data access, test cases, etc.)

2. Use npm in it for a New Project

- Npm init will automatically generate a package.json file for your project that shows all the packages/node app of npm install has the information of your project.

```
$ mkdir demo-node app
$ cd demo-node app

$ npm init –yes
```

- Now you need to specify an engine’s key with the currently installed version of node (node -v):

```
"engines": {

  "node": "10.3.16"

}
```

3. Using Environment Aware, Secured and Hierarchical Configuration File

As security best practices, we should keep our app-level keys easily readable from file and environment variables. We should also keep secrets outside the committed code and make a config file hierarchy for easier accessibility. To meet all this, a perfect and flawless configuration setup is required. There are few node.js development project structure that are available that can help to do this like rc, nconf and config.

Also, developers should leverage the power of npmrc file in their projects, which can automatically restarts a few environment production configurations during npm init like setting up production of metadata inside project package.json config file – Author name/email/licensing details /version, setting up production of npm registry changes, log levels, log messages output level changes, installing global modules and many more.

Developers can set default values can be set through npmrc file with the below commands:

```
npm config set init.author.name "Your Name"

npm config set init.author.email "name.lastname@tatvasoft.com"

npm config set init.author.url "http://www.tatvasoft.com "

npm config set init.license "MIT"

npm config set init.version "1.0.0"
```

4. Avoiding Garbage in-app

- Node js has a default limit of 1.5 GB Single CPU core as process manager but still, it uses a greedy and lazy garbage collector. It waits until the memory usage is reached and gets recovered on its own.
- If you want to gain more control over the garbage collector then we can set the flags on V8.

```
web: node --optimize_for_size --max_old_space_size=920 --gc_interval=100 server.js
```

- You can also otherwise try to run the application using the Docker image. This is important if the app is running in an environment with less than 1.5 GB of available memory usage. For example, if you’d like to tailor a node.js to a 512 MB container, try:

```
web: node --optimize_for_size --max_old_space_size=460 --gc_interval=100 server.js
```

5. Naming Conventions

- Use lowerCamelCase for variables, properties and function names, Class names
- Use UPPERCASE for Constants

5.1 Variables

- Object / Array creation
- Use trailing commas and put short declarations on a single line. Only quote keys when your interpreter complains

```
var a = ['hello', 'world'];

var b = {

 good: 'code',

 'is generally': 'pretty',

};
```

5.2 Conditionals

- Use the === operator

```
var a = 0;

if (a !== '') {

 console.log('winning');

}
```

6. Using Async-Await or Promises

- Good development practices say to use javascript ‘synchronous function’ for multiple callbacks inside promises to handle async error this process results in a callback hell problem.
- We can take a look at the available libraries or async and await of javascript to overcome this performance issue.
- The process manager will use the promises function to catch code error. It reduces code complexity and makes code more readable.

Code Example – use promises

```
return A()

  .then((a) => B(a))

  .then((b) => C(b))

  .then((c) => D(c))

  .catch((error) => logger.error(error))

  .then(E())

Code Example - using async/await to catch errors

async function E() {

  try {

    const a= await A();

    const b= await B(a);

    const c= await C(c);

    return await D(c);

  }

  catch(error) {

    logger.error(error);

  }

}
```

7. Handling Errors Centrally

Every logic that handles errors like logging performance , sending mails regarding error should be written in such a way so that all APIs, night-jobs, unit testing can debug messages and call this method whenever any error occurs.

8. Using Built-in Error Handling Mechanism

- There are many ways otherwise available for developers to raise error and resolve them. They can use strings or even define custom types.
- The Built-in error object makes a uniform approach to handle errors within our source code and other open-source JSON packages.
- It is also recommended to log errors and their names and other Meta properties of errors so that it can be easily identifiable.

```
// throwing an Error from typical function, whether sync or async

if(!productToAdd)

throw new Error('How can I add new product when no value provided?');



// 'throwing' an Error from EventEmitter

const myEmitter = new MyEmitter();

myEmitter emit('error', new Error('whoops!'));



// 'throwing' an Error from a Promise

const addProduct = async (productToAdd) => {

try {

const existingProduct = await DAL.getProduct (productToAdd.id);

if (existingProduct !== null) {

throw new Error('Product already exists!');

}

} catch (err) {

// ...

}

}
```

9. Always Await Promises before Returning to Avoid a Partial Stacktrace

- When an error occurs, whether, from a synchronous or asynchronous flow, it's imperative to have a full stacktrace of the error flow. Surprisingly, if an async function returns a promise (e.g., calls another async function) without awaiting, then an error should occur that makes the caller function disappear in the stacktrace.
- the more if the error cause lies within that caller function then there is a feature v8, also called "zero-cost async stacktraces" that allow stacktraces not to be cut on the most recent await
- So, to avoid these loopholes in stacktraces for the cases when the returned promises would get rejected. So, we must always explicitly resolve these promises by waiting before returning them from the functions.

10. Name Your Functions

You can name all the functions which may include the closures and callbacks. You can restrict the use of anonymous functions. Make sure you use the Naming function. Naming will allow you to simply implement what you want and then Take a snapshot of memory usage.

11. Proper Naming Conventions for Constants, Variables, Functions, and Classes

As a standard best practice, we should use all constants, functions, variables, and class names in lowercase when we declare them. Also, we should not use any short forms instead use only full forms that are easily understandable by everyone using it. We should use underscore between two words.

Code Example :

```
// for class name we use Uppercase

class MyClassExample {}

// Use the const keyword and lowercase

const conf = {

  key: 'value'

};

// for variables and functions names use lowercase

let variableExample = 'value';

function foo() {}
```

12. Use Const Over Let, Do Not Use Var

- Const variables assigned cannot be changed, this will help you prevent the use of a single variable multiple times so that way we can keep our code clean. In some scenarios where we need to re-assign variables, we will use the let keyword. For example, in a loop, if we want to re-declare variable value we can use let.
- Apart from this, “let variables” have blocked the scope, meaning they are accessible inside of a particular block where they are declared. Variables declared using var can be used anywhere inside the function.
- The process manager is a simple command-line interface that keeps the inflow of scripts continuously in all the projects.
- Project) practices for all the Node js security vulnerabilities you come across. Please find security tips below for your web application.

13. Formatting

- 4 spaces for indentation -Use 4 spaces for indenting your code and swear an oath to never mix tabs and spaces — a special kind of hell awaits otherwise.
- No trailing white space
- Write small functions (Max 100 lines)
- Use single quotes- To avoid deep nesting of if-statements, always return a function’s value as early as possible.

```
function isPercentage(val) {

 if (val < 0) {

   return false;

 }

 if (val > 100) {

   return false;

 }

 return true;

}
```

- Opening braces go on the same line

Your opening braces go on the same line as the statement.

```
if (true) {

   console.log('winning');

}
```

- Declare one variable per var statement
- Declare one variable per var statement, it makes it easier to re-order the lines

14. Manage HTTP Headers

In order to prevent clickjacking, cross-site scripting (XSS attacks), and other malicious attacks, you can create a new impact on impactful node.js applications with secure HTTP headers. We can use plug-ins like the helmet which is easy to configure and create our own Node.js security rules.

<u>Recommendation:</u>

- Use HTTP headers as per the project's requirements as shown below
- Access-Control-Allow-Origin: This shows if the response can be shared with requesting client from the given origin.
- Server: Describes the server information that generated the response.
- Strict-Transport-Security: Ensures website is accessed through HTTPS instead of HTTP.
- X-Content-Type-Options: Makes sure that MIME types mentioned in Content-type cannot be changed. In this way, you can restrict the app from MIME type sniffing.
- X-XSS-Protection: In the older versions of IE, Chrome and Safari it prevents loading of webpages when they find XSS attacks. Modern web browsers don't need this kind of production setting when sites implement a strong Content-Security-Policy as it already disables inline javascript.
- X-frame-options: This header makes sure if a page is allowed to be rendered in frame/iframe.

- Content-Security-Policy: This helps to track and stop threats such as XSS attacks (Cross-Site Scripting) and data injection. These attacks can cause data theft, site defacement, and distribution of malware.

15. Implement Automated Testing

You should plan your project deadline in such a way that all your developed functionality by developers can adhere to automated testing. It helps to test APIs without even actually calling them. We can mock database calls, and also it makes sure if the last changes done by someone else are not broken after implementing new features.

16. Structuring Test

You can use Arrange, Act & Assert (AAA) to structure your tests with 3 well-separated sections. Arrange contains all the data or parameters or expected output which will be used in subsequent calls or comparing actual and expected results, Act – calls actual implementation with all arranged parameters, Assert – compares the actual result with the expected result.

Code example:

```
describe.skip('Employee classification', () => {

    test('When employee ranks more than 3.5, classify as Good', () => {

        //Arrange

        const employee = {rank:3.5, joined: new Date(), id:1}

        const stub = sinon.stub(db, "getEmployee")

            .reply({id:1, class: 'Good'});

        //Act

        const actual_result = employeeClassifier.classifyEmployee
(employee);

        //Assert

        expect(actual_result).toMatch('Good');

    });

});
```

17. Clean-up dependencies before production release

- When an image is shipped to the production, it must be clean from any kind of development dependencies and must also be minimal. Even though Dev-dependencies are necessary during the build and test lifecycle, eventually one needs to make sure to produce clutter-free production images.
- This guarantees the number of potential attacks is minimized. Also, it is worth noting that many infamous security vulnerabilities & breaches were found in the development packages.
- While the development team decides to use the multi-stage build, achieving it can be really easy. All one needs to do is install all the dependencies and then finally run the below command:

npm ci –production

18. Clean NODE_MODULE cache

- It is good practice to remove the local cache after installing the dependencies. It makes somewhat sense to duplicate dependencies that enable faster installs for the future as there won't be any further installs because of the immutable image of Docker.
- If this is not done, then the resulting image will get shipped to production with 30% more size containing the files that are never going to be used.

19. Lint your Dockerfile

Linting is always important and linting your dockerfile is no different. It can identify issues with dockerfile that differ from best practices. With the use of a specialized Docker linter, if the expert checks the potential flaws, he can easily find our performance and security improvements which can save countless hours that were wasted or can even save time that goes behind checking security issues in production code.

20. Avoid Global Mock Data

While writing test cases we should use separate mock data for each process case rather than declaring it as global and modifying it every time.

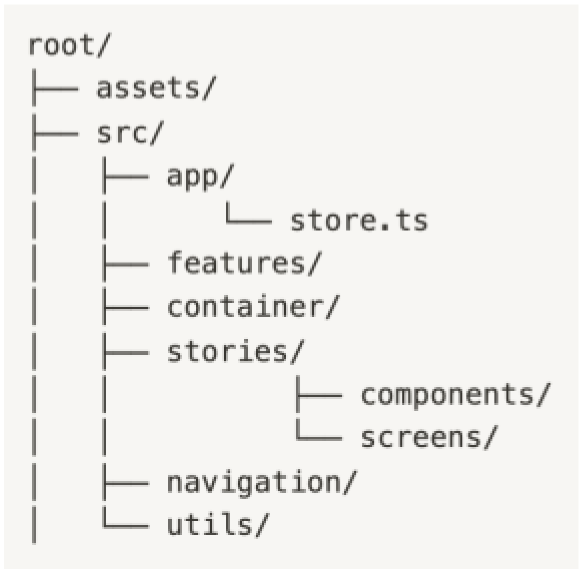
Good Code example:

```
it("When name, it should get success message", async () => {

  const site = await service.add({

    name: "oldName"

  });

  const result = await service.update(site, "changedName");

  expect(result).to.be(true);

});

Bad Code Example:

before(() => {



  await dbmock.GetDataFromJson('mock.json');

});

it("When name, it should get success message", async () => {
  const site = await service.add({

    name: "oldName"

  });

  const result = await service.update(site, "changedName");

  expect(result).to.be(true);
```

React Native Project Guidelines

1. Structure the Files and Folder Properly

- No matter what your application is all about, one basic principle popular in the react ecosystem is keeping all the code-related files and folders into the src folder (source folder) and images or any resource file into the separate assets folder.
- This src and assets folders are exposed in the root folder as depicted in the below layout (React-Native app)



```
root/
├── assets/
├── src/
│   ├── app/
│   │       └── store.ts
│   ├── features/
│   ├── container/
│   ├── stories/
│   │           ├── components/
│   │           └── screens/
│   ├── navigation/
│   └── utils/
```

2. Keep the Component and Screen Clean

- The next crucial point to note is keeping the component clean which means for all the API fetch or DB operations we need to dispatch an action from the component and finally, the reducer should change the state.
- One should also keep all the long-coded logic wrapped inside a helper function and call that function from the component.

3. Keep the App Responsive and Avoid Inline Styling

Ensure your app does not break when rendered on different-sized devices. Define styles at the proper breakpoint and use responsive elements such as flex and provide relative values instead of absolute ones while styling. Inline stylings should be avoided as it distracts the reader from the main code and confuses them with a lot of unnecessary lines. Keep them into a separate style.css file.

- It is good to use <SafeAreaView> when using react native to render the component. It ensures the component remains positioned correctly in all iPhone devices.

4. Naming Conventions

Folder and sub folder name start with small letters and the files belongs the folders is in pascal case

1. When the file is in a folder with the same name, we don't need to repeat the name. That means, components/user/form/Form.js, would be named as User Form and not as User Form.
2. Include all the controls in a single import belonging to the same module end with a semicolon. There should be no space between two imports.

```
import {ScrollView, View, TouchableOpacity, KeyboardAvoidingView,
ListView, AsyncStorage, Alert} from 'react-native';
```

- Variable declaration in camel case statement

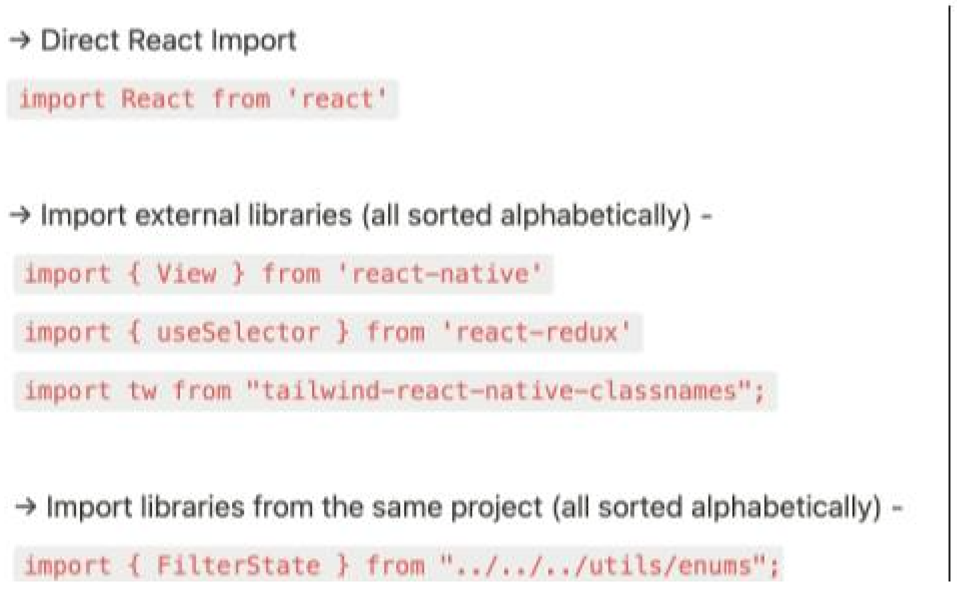
let textExample = "Hello World"

4.1 Utils Folder

- Ensure your utils folder contains helper, constant, and enum files only. For config files, keep them in a separate folder.

5. Ordering Imports In The File

Divide all imports into segments logically and provide one line gap between each segment as it helps during code review. Ensure you should order all the imports added in the component/screen file alphabetically and sequentially in the below order.


→ Direct React Import

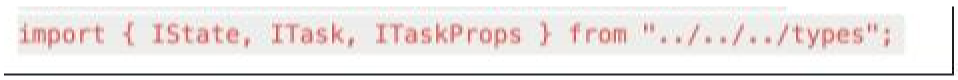
```
import React from 'react'
```

→ Import external libraries (all sorted alphabetically) -

```
import { View } from 'react-native'
import { useSelector } from 'react-redux'
import tw from "tailwind-react-native-classnames";
```

→ Import libraries from the same project (all sorted alphabetically) -

```
import { FilterState } from "../../../utils/enums";
```



```
import { IState, ITask, ITaskProps } from "../../../types";
```

6.Use Es6 Concepts and Get Rid Of Legacy Code

It is always advisable to use the latest version of libraries present and use ES6 concepts if starting from scratch.

For instance: Usage of let or const instead of var.

7. Document Everything Neatly

Ensure to add meaningful JSDoc comments to the function created by you. It will guide new developers on what this function is all about.

7.1 Commenting

- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text.
- Attach comments to code only where necessary. This is not only in keeping with React

8. Remove Logs from The Final App Build

Remove all the console.log() statements from the final build as too many log statements may impact the performance of the app.

9. Add Test Cases in Your App

As we know no app is deployment ready until it is tested thoroughly, so testing plays a vital role in app development. One can do unit testing, snapshot testing using JEST and Enzyme to test the component being rendered as per expectation.

High Impact Modules (Unit Testing)

10. Bug Avoidance

- Use optional chaining if things can be null
- Use the guard **pattern/prop types/typescript** to ensure your passed in parameters are valid
- Create PURE functions and avoid side-effects
- Avoid mutating state when working with arrays
- Remove all console.log()
- Treat props as read-only. Do not try to modify them

10.1 Exception Handling

- We use the try and catch blocks to handle exceptions within the React Native app. Mobile application that crashes with errors that aren't handled gracefully. So, exception handling plays an important role in making your app run smoothly.

```
try {

throw new Error("Error");
} catch (error) {

// handle Error

}
```

11. Architecture & Clean Code

- No DRY violations. Create utility files to avoid duplicate code
- Follow the **component/presentation** pattern where appropriate. Components should follow the single responsibility principle
- Use Higher Order Components where appropriate
- Split code into respective files, JavaScript, test, and CSS
- Create a index.js within each folder for exporting. This will reduce repeating names on the imports

```
import {Nav} from './Nav.js';

import {CookieBanner} from './CookieBanner.js';

export {Nav, CookieBanner}
```

- Only include one React component per file
- Favour functionless components
- Do not use mixins
- No unneeded comments
- Methods that are longer than the screen should be refactored into smaller units
- Commented out code should be deleted, not committed

12. CSS

- Avoid Inline CSS
- A naming convention is defined and followed (BEM, SUIT, etc..

13. Loaders

- Loading spinners While Fetching The Data Or Waiting For an API Response

14. Testing

- Write tests
- Define a quality gate using coveralls
- Do not test more than one thing in a test
- No logic should exist within your test code
- Test classes only test one class
- Code that needs to talk to a network, or, database is mocked

So, these were some of the best coding practices which you can follow while building your project. It will definitely reduce the number of bugs and crashes in your application.

React Project Guidelines

1. Code structure:

- Put all your imports in order at the top of every file. Prioritize import as follows:
  - React Imports (Ex: import * as React from ‘react’)
  - Library Imports (Ex: import Chart from 'react-apexcharts';)
  - File Imports (Ex: import * from “./Somefile”)
- Do not create multiple components in a single file. Create a separate file for every component.
- Dividing the whole app into components and then working on it on a separate file is a good practice to maintain clean code.
- Do not use inline CSS. Always create classNames and change styles of the component
  - Ex:
    - <div style={{color: ‘red’, fontSize: 16px}}>Some Component</div>
    - <div className=”text”>Some Component</div>
- Create global CSS structure and (avoid / allow minimal) component level CSS as styles can be used across the application which will avoid duplicate code.

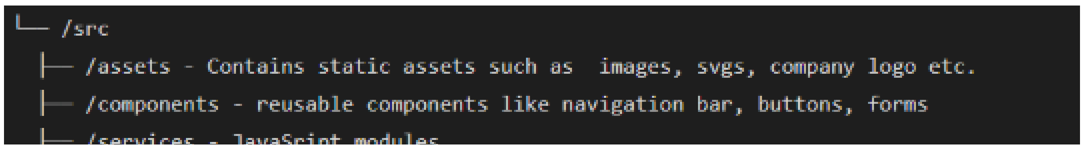
2. Naming Conventions:

- Use PascalCase for naming the files. (PascalCase --> is a programming naming convention where the first letter of each compound word in a variable is capitalized)
- Name of the file and the component should be same.
  - Ex: if the React component is LoginForm then the file should be named as LoginForm.js
- Use camelCase for naming variables, constants, functions etc. (camelCase --> camelCase is a way to separate the words in a phrase by making the first letter of each word capitalized and not using spaces)
- Use well-descriptive names for variables/functions/modules/Actions, keeping in mind that it is application-specific so that even a third party or new developer can easily understand the code.

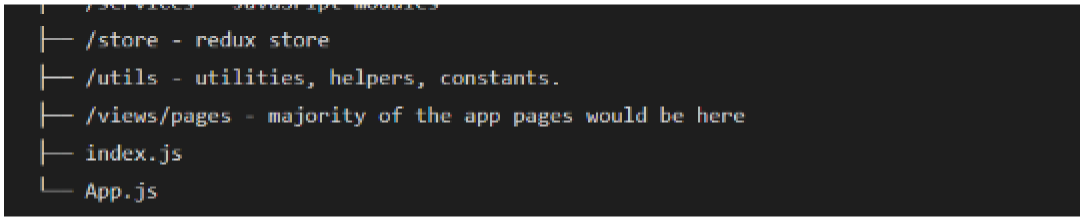
3. Code Practices:

- Remove all console statements. Have them during the process of development as they are required in the process of debugging.
- But make sure the console statements are removed before the code is deployed.
- Avoid multiple ***if-else*** blocks instead use ***ternary*** operator.
- Remove all commented-out code that is not useful anymore.
- Use React Hooks which will ease the application development.
- Write useful comments that explains important and complex logics. Also avoid writing comments for easily understandable code.
- Use const instead of var or let. const lets you check that a variable should always stay constant. let indicates that the values of the variables can be changed. Avoid using var
- Always prefer to use arrow functions which will help in accessing this keyword and helps writing smaller functions in a single line.
- Always ensure the application is responsive.
- Maintain ***README.md*** files for all components that will help in writing the code documentation and explain the code.
- Always use third party libraries with an ***MIT*** license.

4. App's Directory Structure Shall Be as Follows:



```
└── /src
  ├── /assets - Contains static assets such as  images, svgs, company logo etc.
  ├── /components - reusable components like navigation bar, buttons, forms
  ├── /services - JavaScript modules
```



```
├── /services - javascript modules
├── /store - redux store
├── /utils - utilities, helpers, constants.
├── /views/pages - majority of the app pages would be here
├── index.js
└── App.js
```

5. Component Structure

To keep all the component files consistent,

- Use PropsWithChildren
- Component controllers as higher order functions
- Helper function — utils/wrap.ts