IT Solutions
ArhamSoft (Pvt) Ltd.

Coding Standards

Phase 1:

The following are the standards we need to enforce before we get into the more complex ones. All the team leads will be responsible to enforce these standards on every project. Once these are implemented, we can add some more advanced in the next phases.

Note: Any team or project found not implementing these standards, will be penalized along with the team lead.

1. **Project Structure:**
    a. Split stuff into multiple layers and tiers as per requirements.
    b. Separate presentation/UI, database & business logic, etc.
    c. For example, all the assets are in a separate directory, business logic like reducers & actions are kept in other folders, and controllers have their own place, similar to the models.
    d. An example project structure will be shared soon.
2. **README.md File:**
    a. This file should include brief and clear project details (e.g. title, purpose, build status, tech stack/framework) and other instructions like configuring and running the project on local machines.
    b. This file should also include the required software/dependencies along with the required version.
3. **Static Default Values:**
    a. All the static/constant default/hard-coded values must be stored in a single file instead of being used directly in the code.
    b. A glimpse of the sample file is attached below:

```
src > config > JS vars.js > [@] <unknown> > openAIBaseUrls
31    USDtoBNBLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=USD&convert=BNB',
32    USDtoMYNTLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=USD&convert=NRG',
33    BNBtoUSDLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=BNB&convert=USD',
34    WBNBtoUSDLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=WBNB&convert=USD',
35    MYNTtoUSDLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=NRG&convert=USD',
36    WBNBtoBNBLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=WBNB&convert=BNB',
37    MYNTtoBNBLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=NRG&convert=BNB',
38    BNBtoWBNBLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=BNB&convert=WBNB',
39    USDtoETHLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=USD&convert=ETH',
40    ETHtoUSDLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=ETH&convert=USD',
41    WETHtoUSDLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=WETH&convert=USD',
42    WETHtoETHLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=WETH&convert=ETH',
43    ETHtoWETHLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=ETH&convert=WETH',
44    MYNTtoETHLink: 'https://pro-api.coinmarketcap.com/v2/tools/price-conversion?amount=1&symbol=NRG&convert=ETH',
45    priceConversionHeaders: {
46      headers: {
47        'X-CMC_PRO_API_KEY': process.env.COIN_MARKET_CAP_API_KEY,
48      }
49    },
50
51    appName: process.env.APP_NAME,
52    myntistContractAddressBNB: process.env.NFT_CONTRACT_ADDRESS,
53    myntContractAddressBNB: process.env.MYNT_CONTRACT_ADDRESS,
54    myntistERC721BNB: process.env.BNB_CONTRACT_ADDRESS_721,
55    myntistERC1155BNB: process.env.BNB_CONTRACT_ADDRESS_1155,
56
```

4. **Package.json:**
    a. There should not be any package that is not being used in the project.
    b. Package.json should not have any unwanted details added to the file.
5. **Clean, Clear & Formatted Code:**
    a. Code should be clear, clean, readable, properly formatted, and indented.
    b. More instructions on code formatting can be seen in the document attached below with the point **Git Necessities**, explained below.
6. **Secret Keys & Environmental Variables:**
    a. If the project has any kind of credentials, these must be stored in the .env file**.**
    b. All the .env files must have an example **.env.example** file with the sample values.
    c. Do not push the .env to remote repos. More instructions in **Git Necessities** point.
7. **Commenting:**
    a. Add proper comments in the code.
    b. The comments should be meaningful.
    c. Avoid unnecessary and obvious comments.
    d. Comments should be brief, and relevant.
    e. **Bad Commenting Examples:**

```
/**
* PositiveNumbersLength function
*/
```

```
function PositiveNumbersLength(data=[]) {
  //If array's length is 0          <---- This is obvious
  if(!data.length) {
     return 0;
  }
  let positiveLength;
  //Loop for all numbers in the array          <---- This is obvious
  for(let i=0; i<data.length; i++) {
     if(data[0]>0) {
        positiveLength++;
     }
  }
  return positiveLength;
}
```

f. Good Commenting Examples:

i. Initial function definition

```
/**
* Returns the length of positive numbers from a given array @data passed as param
* Author: Umer Surkhail
* Date: 23 Jan, 2023
*/

function PositiveNumbersLength(data) {
  let positiveLength;
  for(let i=0; i<data.length; i++) {
     if(data[0]>0) {
        positiveLength++;
     }
  }
  return positiveLength;
}
```

ii. Update 1:

```
/**
* Returns the length of positive numbers from a given array @data passed as param
* Author: Umer Surkhail
* Date: 23 Jan, 2023
* Update 1 (24 Jan, 2023): Check the length of array before loop, and return 0 if the array is empty.
*/

function PositiveNumbersLength(data) {
  if(!data.length) {
     return 0;
```

```
  }
  let positiveLength;
  for(let i=0; i<data.length; i++) {
     if(data[0]>0) {
        positiveLength++;
     }
  }
  return positiveLength;
}
```

iii. Update 2:

```
/**
* Returns the length of positive numbers from a given array @data passed as param
* Author: Umer Surkhail
* Date: 23 Jan, 2023
* Update 1 (24 Jan, 2023): Check the length of array before loop, and return 0 if the array is empty.
* Update 2 (24 Jan, 2023): Assign default value to the param, an empty array, in case the argument is not
passed.
*/

function PositiveNumbersLength(data=[]) {
  if(!data.length) {
     return 0;
  }
  let positiveLength;
  for(let i=0; i<data.length; i++) {
     if(data[0]>0) {
        positiveLength++;
     }
  }
  return positiveLength;
}
```

iv. Update 3:

```
/**
* Returns the length of positive numbers from a given array @data passed as param
* Author: Umer Surkhail
* Date: 23 Jan, 2023
* Update 1 (24 Jan, 2023): Check the length of array before loop, and return 0 if the array is empty.
* Update 2 (24 Jan, 2023): Assign default value to the param, an empty array, in case the argument is not
passed.
* Update 3 (24 Jan, 2023): Modified the function to find the positive numbers count in the first half of the
array.
```

```
*/

function PositiveNumbersLength(data=[]) {
  if(!data.length) {
     return 0;
  }
  let positiveLength;
  // Looping through the first half, because we need to find the positive numbers in the 1st half of the array.
  for(let i=0; i<data.length/2; i++) {
     if(data[0]>0) {
        positiveLength++;
     }
  }
  return positiveLength;
}
```

8. **No Commented Out Code:**
    a. Code that is disabled or excluded from execution in the app should be removed.
    b. Leaving commented-out code in source code is bad practice, as it takes up space, causes confusion, and leads to maintenance issues if it's not removed. Therefore, the commented-out code should be removed from the project.
9. **Consistent and Meaningful Naming:**
    a. The naming scheme for the methods, classes, variables, files, and folders should be consistent throughout the project. No matter how many developers are working on a project.
    b. Clear and meaningful names make code more readable and understandable and act as self-documentation within code.
    c. **Guidelines for Variables/Functions Naming:**
        i. Use descriptive names that convey the purpose, meaning, or role of the variable/function.
        ii. Follow a consistent naming convention throughout the codebase.
        iii. Choose names that are concise but still descriptive.
        iv. Use camel case (e.g., myVariableName) or underscore-separated (e.g., my_variable_name) naming conventions based on the programming language or style guide you are following. In JS, camel case is preferred so better opt for this convention.
    d. **Examples:**

i. If a variable is named as camelCase, all the variables should be camelCased in the project.
ii. If a function's name is defined as separated_by_underscores, all the function names in the whole project should be separated_by_underscores.
iii. A class/component's name is defined as InitialsCapital, all the classes/components in the project should be InitialsCapital
iv. Should be:
    1. Schemas model name, singular and Initial Capital

```
function Item(props) { // <----- Name of component, InitialsCapital
 const publishItem = async () => { // <----- Name of method/function, camelCase
    props.publishItem(props._id);
 }
 const renderLi = (type) => { // <----- Name of method/function, camelCase
     const {end_time, bidder, start_time } = props;  // <----- variable names separated_by_underscore
     // implementation
     return (
        <></>
     )
 }
 const renderLink = (type) => { // <----- Name of method/function, camelCase
  // implementation
  return (
     <></>
  )
 }
 return (
  // implementation
    <></>
 );
}
```

10. Developer Docs. (dev-instructions.md)

a. Continuing with the previous point, we can write all the instructions in a file.
b. The purpose of this file is to keep all the developers working on a project on board for the naming convention and project structure-related things.
c. Team leads will be responsible for creating this file, and all the developers will follow the instructions.

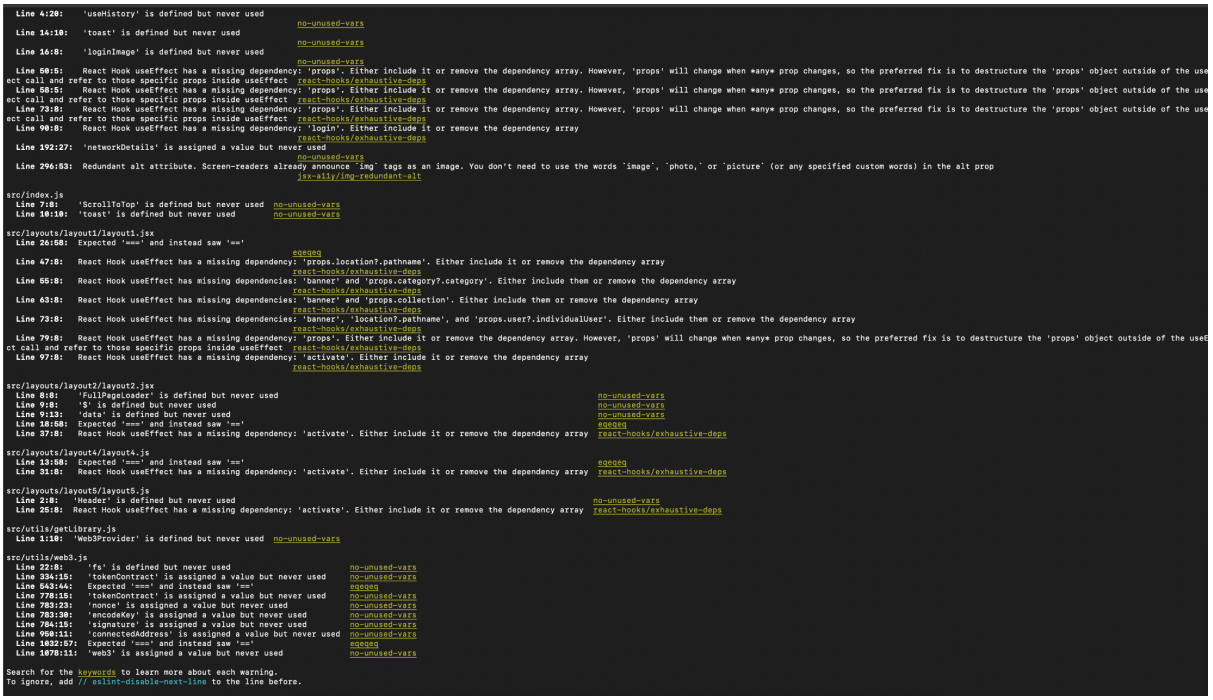
d. This file should be well-documented and must include instructions for developers like naming convention schemes and other coding techniques being used in the project.

11. Variables:

a. In JS, we can define variables using **var, let,** or **const**.
b. If the variable's value is not going to change, it is necessary to use it as const, otherwise, use it as let.
c. Avoid **var**:
    i. Use ***var* if and only if** it is really necessary.

12. Application Console Warnings:

a. The consoles of the applications should not have any errors or warnings. When we start a react-based application, it shows warnings and errors in the console. Most of the time, the warnings are ignored, but these should not be ignored as these can help a lot in optimizing the applications.
b. This should not only be restricted to the terminal console but also the browser's console.
c. **Example consoles with warnings:**



d. **For example consoles without any warning, make sure to not have any warning in the console:**

```
Compiled successfully!

You can now view e-auction in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.18.15:3000

Note that the development build is not optimized.
To create a production build, use yarn build.

webpack compiled successfully
```

13. Logs:

a. All the logs added to the code for debugging purposes should be removed before committing/pushing the code.

b. Sometimes there can be scenarios where we need to debug use cases on the live site and have to push the logs. Those logs should be removed after the error is tracked and fixed.

c. **We often see these kinds of logs in the live application, which is a bad practice and should be removed before pushing the code.**

```
4 handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
  handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
2 handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
  handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
3 handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
  handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
4 handle image …fyjR8W+EpRuAgc/3nD1vyXJQWtmWEJec97TzYfHNcZJ/D7v3c59HdNMxbJAAAAAElFTkSuQmCC
  props.settings ▸{settings: {…}, settingsAuth: true, subscribe: null, subscribeAuth: false, tokenSettings: Array(5), …}
  chainBased Cur ▸{name: 'Ethereum', label: 'ETH', symbol: 'ETH', value: 1, icon: 'assets/images/ethereum.svg', …} ▸(5) [{…}, {…}, {…}, {…}, {…}]
  props.settings ▸{settings: null, settingsAuth: false, subscribe: null, subscribeAuth: false, tokenSettings: null, …}
```

```
collection: categories, method: createIndex
Filter =  {"name":1}
Result = { unique: true, background: true }


collection: emails, method: createIndex
Filter =  {"type":1}
Result = { unique: true, background: true }


collection: collections, method: createIndex
Filter =  {"name":1}
Result = { unique: true, background: true }

EVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENT
LOG 1
EVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENT
LOG 1
EVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENTEVENT
LOG 1
```

```
otherEventsExistotherEventsExistotherEventsExistotherEventsExist
LOG 3

collection: transfersubscriptions, method: findOne
Filter =  {"transactionHash":"0x2e3d56c5e6f5fb74bd61914475a69b6b03745ec576321fd8f6b655963ca74b0b"}
Result = { projection: { transactionHash: 1 } }


collection: sellhistories, method: insertOne
Filter =  {"sold":true,"copyCheck":false,"_id":"63e0b7f5ca0f234491689944","sellerId":"634512fb9017
67947","createdAt":"2023-02-06T08:19:01.964Z","updatedAt":"2023-02-06T08:19:01.964Z","__v":0}
Result = { session: null }


collection: sellingnfts, method: findOneAndUpdate
Filter =  {"_id":"6363caa850caafc74b0ff78f"}
Result = {
  '$setOnInsert': { createdAt: 2023-02-06T08:19:01.967Z },
  '$set': {
    updatedAt: 2023-02-06T08:19:01.967Z,
    avaliableNfts: [],
    sellingNfts: []
  }
}
```

14. Switches for Conditional Rendering:

a. While working with conditional rendering on JS projects, specifically on the frontend side, we often see the rendering of components in the ternary conditions in the same render, **which makes code look very complex to see and understand**. In cases like these, we should use switch statements to render condition-based components, which makes code look more easy to understand.

b. **Example without switch (conditional rendering in the same render)**

```
function Item(props) {
  const publishItem = async () => {
    props.publishItem(props._id);
  }
  return (
    <div className="item-holder mb-4">
      <h3 className="item-name d-block mb-2">
        <Link to={`/item-detail/${props._id}`}>{props.name}</Link>
      </h3>
      <ul className="list-unstyled">
        {
          props.type === 1 ?
          <li>
            <strong>Earned By: </strong>
            <span>{props.bidder ? props.bidder : 'None (Ended without Accepting)'}</span>
          </li> :
          props.type === 2 ?
          <li>
            <strong>Ends In: </strong>
```

```
                <span>{moment(new Date(props.endTime)).format('MM/DD/YYYY h:mm A')}</span>
              </li> :
              props.type === 3 || props.type === 4 ?
              <>
                <li className="mb-2">
                  <strong className="d-block mb-1">Start Time: </strong>
                  <span>{moment(new Date(props.startTime)).format('MM/DD/YYYY h:mm A')}</span>
                </li>
                <li>
                  <strong className="d-block mb-1">End Time: </strong>
                  <span>{moment(new Date(props.endTime)).format('MM/DD/YYYY h:mm A')}</span>
                </li>
              </> : ""

          }
        </ul>
        {
          props.type === 2 ?
          <Link to={`/item-detail/${props._id}`} className="btn btn-primary">Bid</Link> :
          props.type === 3 ?
          <Button variant="primary" onClick={() => publishItem()}>Publish</Button> :
          ""
        }
      </div>
  );
}
```

c. **Example with switch (conditional rendering different functions)**
   i. This might have added up some additional lines of code, but has made the code more readable and easy to understand.

```
function Item(props: ItemData) {
  const publishItem = async () => {
    props.publishItem(props._id);
  }
  const renderLi = (type: number) => {
    switch(type) {
      case 1:
        return (
          <li>
            <strong>Earned By: </strong>
            <span>{props.bidder ? props.bidder : 'None (Ended without Accepting)'}</span>
          </li>
        )
```

```
      case 2:
        return (
          <li>
            <strong>Ends In: </strong>
            <span>{moment(new Date(props.endTime)).format('MM/DD/YYYY h:mm A')}</span>
          </li>
      )
      case 3:
      case 4:
        return (
          <>
            <li className="mb-2">
              <strong className="d-block mb-1">Start Time: </strong>
              <span>{moment(new Date(props.startTime)).format('MM/DD/YYYY h:mm A')}</span>
            </li>
            <li>
              <strong className="d-block mb-1">End Time: </strong>
              <span>{moment(new Date(props.endTime)).format('MM/DD/YYYY h:mm A')}</span>
            </li>
          </>
      )
      default:
        return "";
    }
  }
  const renderLink = (type: number) => {
    switch(type) {
      case 2:
        return (
          <Link to={`/item-detail/${props._id}`} className="btn btn-primary">Bid</Link>
      )
      case 3:
        return (
          <Button variant="primary" onClick={() => publishItem()}>Publish</Button>
      )
      default:
        return "";
    }
  }
  return (
    <div className="item-holder mb-4">
      <h3 className="item-name d-block mb-2">
        <Link to={`/item-detail/${props._id}`}>{props.name}</Link>
```

```
        </h3>
        <ul className="list-unstyled">
          {renderLi(props.type)}
        </ul>
        {renderLink(props.type)}
      </div>
  );
}
```

15. **Destructive Properties:**
    a. We frequently work with objects and arrays to perform different actions based on the data stored in these. And often we need to extract properties from objects and assign them to new variables.
    b. Destructuring is a very useful feature to extract properties from objects and assign them to variables.
    c. We should use destructive properties instead of old methods.
    d. Please see some examples below for better explanations.

```
//We need to extract some properties from the object and need to set it in a variable.
const student = {
  name: 'Umer Surkhail',
  grade: 'A',
  id: 1,
  section: 'A'
}

//One way is to access the properties one by one and assign them in the variable.
const section = student.section;
const name = student.name
const id = student.id
const grade = student.grade

//Object destructuring makes it possible to get required properties and assign them to variables in one line.
const {section, name, id, grade} = student
```

    e. You may go through this link to discover **Destructive properties** in detail: Ref:
        i. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
        ii. https://dmitripavlutin.com/javascript-object-destructuring/

16. **Arrow Functions:**

a. It is preferred to use arrow functions because of some important reasons:
   i. Arrow functions reduce the size of the code.
   ii. Make code more structured and readable.
   iii. The return statement and functional braces are optional for single-line functions.

17. Template Literals:

a. There are many cases when we need to use a string in our project. We should use template strings because it makes the code more readable.
b. It makes it easy to concat strings and variables and provides a clear view of the code
c. Examples:
   i. Without string literals:

```
const str2 = "This is a string 2 concatenated to be with string 1";
return "This is a string 1" + str2;
```

   ii. With string literals:

```
const str2 = `This is a string 2 concatenated to be with string 1`;
return `This is a string 1 ${str2}`;
```

18. Promises and async/await:

a. Use promises or async/await along with handled rejection.

19. Minimized Logic in the Render Method:

a. Use minimum logic in the render method to make things more simple, clear, and readable.
b. When we have simple logic, think logic that is easy to read and understand, we can put it in our render function. If instead, we have logic that we would not want to be re-created each time the render function is called, or logic that is harder to read/understand, etc. we should put that logic in helper methods or in redux actions (if used).
c. **Example:**

```
render() {
    const { nft } = this.state
    const fileTypes = ENV.nftFileTypes
    const currentDate = moment(new Date(), 'YYYY-MM-DD HH:mm:ss:SSS')
    const nftStartDate = moment(
        new Date(nft?.auctionStartDate),
        'YYYY-MM-DD HH:mm:ss:SSS',
    )
    const nftEndDate = moment(
        new Date(nft?.auctionEndDate),
        'YYYY-MM-DD HH:mm:ss:SSS',
    )
    const isAuctionStarted = nftStartDate.isBefore(currentDate)
    const isAuctionPast = nftEndDate.isBefore(currentDate)
    const withCopies = nft?.copies > 1
    const multipleOwners = nft?.owners?.total > 1
    const owners = nft?.owners?.list || []
    const isOwner = nft?.owners?.isOwner  // flag to know either if viewer is owner or not
    const blockChainName = blockChains.filter((b) => b.chainId === nft?.chainId)[0]?.chainName

    let nftPriceInUSD = 0
    if (nft.currentPrice) {
        if (nft.chainId === chainIds['BNB'][0] || nft.chainId === chainIds['BNB'][1]) {
            if (nft.currency === 'BNB' && this.props.app.rateAuth)
                nftPriceInUSD = convertBnbToUsd(nft.currentPrice, this.props.app.rate)

            if (nft.currency === 'MYNT' && this.props.app.myntRateAuth)
                nftPriceInUSD = convertMyntToUsd(nft.currentPrice, this.props.app.myntRate)

        }
        else if (nft.chainId === chainIds['ETH'][0] || nft.chainId === chainIds['ETH'][1]) {
            if (nft.currency === 'ETH' && this.props.app.ethToUsdRateAuth)
                nftPriceInUSD = convertEthToUsd(nft.currentPrice, this.props.app.ethToUsdRate)

        }
    }
    ...
}
```

i. From line # 2 to 19 it's simpler logic so, leaving it as it is whereas, from line # 21 to onward things could have been made more nicer if this above code was written in some helper function such as:

```
render() {
    const { nft } = this.state
    const fileTypes = ENV.nftFileTypes
    const currentDate = moment(new Date(), 'YYYY-MM-DD HH:mm:ss:SSS')
    const nftStartDate = moment(
        new Date(nft?.auctionStartDate),
        'YYYY-MM-DD HH:mm:ss:SSS',
    )
    const nftEndDate = moment(
        new Date(nft?.auctionEndDate),
        'YYYY-MM-DD HH:mm:ss:SSS',
    )
    const isAuctionStarted = nftStartDate.isBefore(currentDate)
    const isAuctionPast = nftEndDate.isBefore(currentDate)
    const withCopies = nft?.copies > 1
    const multipleOwners = nft?.owners?.total > 1
    const owners = nft?.owners?.list || []
    const isOwner = nft?.owners?.isOwner  // flag to know either if viewer is owner or not
    const blockChainName = blockChains.filter((b) => b.chainId === nft?.chainId)[0]?.chainName

    const nftPriceInUSD = nft && currencyConversions(nft)
    ...
}
```

```
// helper function written somewhere in class declaration or in some utils
const currencyConversions = (nft) => {
    if (nft.currentPrice) {
        if (nft.chainId === chainIds['BNB'][0] || nft.chainId === chainIds['BNB'][1]) {
            if (nft.currency === 'BNB' && this.props.app.rateAuth)
                return convertBnbToUsd(nft.currentPrice, this.props.app.rate)

            if (nft.currency === 'MYNT' && this.props.app.myntRateAuth)
                return convertMyntToUsd(nft.currentPrice, this.props.app.myntRate)

        }
        else if (nft.chainId === chainIds['ETH'][0] || nft.chainId === chainIds['ETH'][1]) {
            if (nft.currency === 'ETH' && this.props.app.ethToUsdRateAuth)
                return convertEthToUsd(nft.currentPrice, this.props.app.ethToUsdRate)

        }
    }
}
```

20. Npm Packages and Versions:

a. While starting any project use the latest stable version of Node, npm, and all the packages being used in the project.
b. Keep updating the package versions, while the project is in the development state.
c. Please make sure the packages being used have enough downloads and are secure to use.
d. **Npm-check-updates** is a package that you can use to identify the upgradable packages and upgrade them.
https://www.npmjs.com/package/npm-check-updates
    i. As JS is evolving very fast and there are major changes coming up every other day. So, keep checking the upgradable packages and upgrade them.

ii. Once you have installed the npm-check-updates package, you can check the upgradable packages in any project by running the command **ncu**. It will list upgradable packages with the currently installed and latest available version.

```
Checking /opt/homebrew/var/www/html/Biiview/biiview-frontend/package.json
[====================] 55/55 100%

 @stripe/react-stripe-js  ^1.16.3  →  ^1.16.4
 axios                     ^1.2.2  →   ^1.3.2
 bootstrap                 ^5.0.1  →   ^5.2.3
 chart.js                  ^4.1.1  →   ^4.2.0
 gsap                     ^3.11.3  →  ^3.11.4
 npm                       ^9.2.0  →   ^9.4.1
 react-chartjs-2           ^5.1.0  →   ^5.2.0
 react-datepicker          ^4.8.0  →  ^4.10.0
 react-moment              ^1.1.2  →   ^1.1.3
 react-router-dom          ^6.4.5  →   ^6.8.0
 recharts                  ^2.2.0  →   ^2.3.2
 redux                     ^4.2.0  →   ^4.2.1
 sweetalert2              ^11.7.0  →  ^11.7.1
 web-vitals                ^3.1.0  →   ^3.1.1

Run ncu -u to upgrade package.json
```

iii. Running **ncu -u** will update the packages with the latest versions in the package.json file. So, it will require **npm install** to update all the latest versions listed in the package.json.

```
Upgrading /opt/homebrew/var/www/html/Biiview/biiview-frontend/package.json
[====================] 55/55 100%

 @stripe/react-stripe-js  ^1.16.3  →  ^1.16.4
 axios                     ^1.2.2  →   ^1.3.2
 bootstrap                 ^5.0.1  →   ^5.2.3
 chart.js                  ^4.1.1  →   ^4.2.0
 gsap                     ^3.11.3  →  ^3.11.4
 npm                       ^9.2.0  →   ^9.4.1
 react-chartjs-2           ^5.1.0  →   ^5.2.0
 react-datepicker          ^4.8.0  →  ^4.10.0
 react-moment              ^1.1.2  →   ^1.1.3
 react-router-dom          ^6.4.5  →   ^6.8.0
 recharts                  ^2.2.0  →   ^2.3.2
 redux                     ^4.2.0  →   ^4.2.1
 sweetalert2              ^11.7.0  →  ^11.7.1
 web-vitals                ^3.1.0  →   ^3.1.1

Run npm install to install new versions.
```

21. Dependencies Vs DevDependencies:

a. There are a lot of packages we install in our project, which do not directly relate to the flow or functioning of the project, but provide different options to run or optimize the projects. For example, the package **nodemon** is used to run the Node.js in watch mode, **eslint** package is used to integrate the linting rules, **prettier** is used to define the rules for formatting. These packages are not dependencies of the project, but dependencies of the development. npm provides a

completely separate directive for these development-related dependencies which is **devDependencies.** All the packages used for development purposes should be installed in the **devDependencies,** but not in the **dependencies.** Installing **devDependencies** in the **dependencies** have some major drawbacks, and **greater-sized/un-optimized applications** are one of them. When we install the development-related packages separately, it benefits us. For example, while creating a build on the ReactJS side, the **devDependencies** will be ignored by the builder and will not be included in the build, thus saving us a lot of files and size. On the other hand, if **devDependencies** are installed in the **dependencies** section, all the code of these packages will be included in the build, thus greater size with useless code.

```
{
"name": "typescript-express",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "dev": "nodemon src/index.ts",
  "prod": "node dist/index.js",
  "build": "tsc"
},
"author": "",
"license": "ISC",
"dependencies": {
  "bcrypt": "^5.1.0",
  "bluebird": "^3.7.2",
  "body-parser": "^1.20.1",
  "cors": "^2.8.5",
  "dotenv": "^16.0.3",
  "express": "^4.17.1",
  "jsonwebtoken": "^9.0.0",
  "moment": "^2.29.4",
  "mongoose": "5.11.15",
  "passport": "^0.6.0",
  "passport-local": "^1.0.0",
  "socket.io": "^4.5.4"
},
"devDependencies": {
  "@types/bcrypt": "^5.0.0",
  "@types/bluebird": "^3.5.38",
  "@types/body-parser": "^1.19.2",
```

```
  "@types/cors": "^2.8.13",
  "@types/express": "^4.17.11",
  "@types/jsonwebtoken": "^9.0.0",
  "@types/moment": "^2.13.0",
  "@types/mongoose": "^5.11.97",
  "@types/node": "^18.11.18",
  "@types/passport": "^1.0.11",
  "@types/passport-local": "^1.0.34",
  "@types/socket.io": "^3.0.2",
  "nodemon": "^2.0.7",
  "ts-node": "^10.9.1",
  "typescript": "^4.5.4"
}
}
```

22. **Common Functions in Helper Folder:**
    a. Common functions should be in utils or some helper folder
    b. The screenshot attached below shows that inside the **utils** folder, there is a file named **functions.js** which contains all the common functions **e.g. axiosSyncPost** is the common function used many times in the project but its definition is written once only.

23. **No Duplicate Code:**
    a. Continuing with the previous point, there should not be any duplicated code. For example, the developer can define the same function in 2 different files, and use it in the respective

file. But, there should not be any such case, and all the repeating codes should be defined once.
***Note**: We encourage you to practice this point for both Node.js & ReactJS apps but to make things easier, for now, we've added this clause for Node.js only.*

24. **Proper HTTP Methods:**
    a. In Express, routes are used to handle different types of requests to the server, such as GET, POST, PUT, DELETE, etc. Each route is associated with a specific HTTP method, and it's important to ensure that the correct method is being used for each route.
    b. For example, a route that is used to retrieve data from the server should be associated with the GET method, while a route that is used to create new data should be associated with the POST method.
    c. Using the wrong HTTP method for a route can lead to unexpected behavior and security issues. For example, if a route associated with the GET method is used to delete data, it could be possible for an attacker to craft a malicious request that could delete data from the server without proper authorization.

25. **Input Validation:**
    a. Input validation is the process of checking that the data received by the server is valid and meets certain criteria before it is processed. This is an important step in ensuring the security and stability of the application, as well as preventing malicious actors from injecting unwanted or harmful data.
    b. It's also important to check for the presence of required fields, for example, if a form requires a name, email, and password fields, it's important to check that these fields are present and not empty in the request body.
    c. You must use input validation to check that the required fields are present.
26. **User Authentication and Authorization:**
    a. In the Node.js app, the authentication process should use secure methods such as hashing and salting passwords. The authorization process should use role-based access control (RBAC) or other similar methods to ensure that a user can only perform actions that they are authorized to do.
    b. Create a middleware to enforce authentication and authorization.
    c. **Example:**

```
exports.userValidation = async (req, res, next) => {
 let flag = true;
 req.user = 0;
 if (req.headers["x-access-token"]) {
   await jwt.verify(
     req.headers["x-access-token"],
     pwEncryptionKey,
     async (err, authorizedData) => {
       if (err) {
         flag = false;
         const message = "session_expired_front_error";
         return res.send({ success: false, userDisabled:
true, message, err });
       } else {
         const reqPlatform =
Number(req.headers["user-platform"]);
         req.user = authorizedData.sub;

         if (reqPlatform === 2) {
           let admin = await Admin.findById({ _id: req.user
}).lean();
           if (!admin) {
             flag = false;
             return res.send({
               success: false,
               user404: true,
               message: "There is no account linked to that
address",
               notExist: 1,
             });
           }
         } else {
           let user = await User.findById({ _id: req.user
}).lean();
           if (!user) {
             flag = false;
             return res.send({
               success: false,
               user404: true,
               message: "There is no account linked to that
address",
               notExist: 1,
```

```
            });
          }
        }
      }
    }
  );
} else if (req.method.toLocaleLowerCase() !== "options") {
  req.user = 0;
}

if (flag) next();
};
```

27. **Database Models/Collections:**
    a. **Model/Collection Structure:**
        i. The **collection/model name** must be singular and Initial Capital.
        ii. It must have **proper data types** defined for all fields.
            1. **For example:** If a field is some flag then its data type must be a **boolean**.
        iii. **Document validation** should be applied where necessary.
            1. **For example:** If a field is required then must add **required: true**.
        iv. A field should have a **default value** where it can be defined.
            1. **For example:** If a field is some flag then its default value must be **true** or **false** (as per requirement).
        v. **Proper data references/relationships** should be defined.
        vi. **Proper developer-defined values:**
            1. If there is a field inside the model like some *status* having different values then add a comment for all values in the model to let other developers know the values.
            2. **For example**, we have a field named **status** with **type Number**.

```
status: { type: Number, default: 1 }, // 1 =
Pending, 2 = Transferred
```

        vii. **Example:**

```
const mongoose = require('mongoose')
```

```
/**
* Earning Schema - This schema is to get owner earnings etc.
* @private
*/
const EarningSchema = new mongoose.Schema({
   userId: { type: mongoose.Schema.Types.ObjectId, ref:
'User', required: true },
   nftId: { type: mongoose.Schema.Types.ObjectId, ref:
'NFT', required: true },
   collectionId: { type: mongoose.Schema.Types.ObjectId,
ref: 'Collection', required: true },
   price: {  // fee earned by owner
       type: Object,
       default: {
           currency: {
               type: String, default: ''
           },
           amount: {
               type: Number, default: 0
           }
       },
       required: true
   },
   status: { type: Number, default: 1 }, // 1 = Pending, 2 =
Transferred
}, { timestamps: true }
);

/**
* @typedef Earning
*/

module.exports = mongoose.model('Earning', EarningSchema);
```

28. Optimized Queries:

a. It is seen that for performing different actions on the backend, multiple queries are being used, which also involve looping through data and applying queries in the loops. This is considered a very bad practice which will work with only small databases, and will eventually hang up the application even with a medium-sized database.

i. **Bad Example Code:**

```
for (const eachUser of users) {
    User.findByIdAndUpdate({ _id: eachUser._id }, {
        $set: {
            isBlocked: true,
            blockDate: new Date()
        }
    }).exec()
}
```

ii. **Good Example Code:**

```
const userIds = [
    ObjectId('6212.........8d'),
    ObjectId('39a6.........0z'),
    ObjectId('90s2.........2f'),
]

await User.update({ _id: { $in: userIds }}, {
    $set: {
        isBlocked: true,
        blockDate: new Date()
    }
})
```

29. Aggregation Pipelines:

a. MongoDB provides a very powerful tool to manage complex queries. You should learn how to write complex queries using MongoDB aggregate.
b. An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.
c. An aggregation pipeline prevents you from writing queries/logic in loops.
d. **Example:**
    i. This section shows how aggregation pipelines can help in writing optimized queries.
    ii. Consider a simple scenario that we have the following collection named ***orders***:

```
db.orders.insertMany( [
   { _id: 0, name: "Pepperoni", size: "small", price: 19,
     quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },
   { _id: 1, name: "Pepperoni", size: "medium", price: 20,
     quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" ) },
   { _id: 2, name: "Pepperoni", size: "large", price: 21,
     quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" ) },
   { _id: 3, name: "Cheese", size: "small", price: 12,
     quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z" ) },
   { _id: 4, name: "Cheese", size: "medium", price: 13,
     quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z" ) },
   { _id: 5, name: "Cheese", size: "large", price: 14,
     quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" ) },
   { _id: 6, name: "Vegan", size: "small", price: 17,
     quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" ) },
   { _id: 7, name: "Vegan", size: "medium", price: 18,
     quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }
] )
```

iii. Now we need to calculate the ***total quantity of medium-sized pizzas*** only.

iv. See the following query comparisons made for the same example and notice the difference in how aggregation pipelines enhance the query performance.

1. **Bad Practice:**

```
const calculateTotalQty = async (size = "medium") => {
 try {
   // Stage 1: Filter pizza order documents by pizza size
   const orders = await db.orders.find({ size: size })
   // Stage 2: Calculate total quantity
   let totalQuantity = 0;
   for (let i = 0; i < orders.length; i++)
     totalQuantity += orders[i].quantity

   return totalQuantity
 } catch (error) {
   return false
 }
}
```

2. Good Practice:

```
const calculateTotalQty = async (size = "medium") => {
 try {
   const orders = await db.orders.aggregate([
     // Stage 1: Filter pizza order documents by pizza size
     {
       $match: { size: size }
     },
     // Stage 2: Calculate total quantity
     {
       $group: { _id: null, totalQuantity: { $sum: "$quantity" } }
     }
   ])

   if (orders?.length)
     return orders[0].totalQuantity

   return 0
 } catch (error) {
   return false
 }
}
```

v. Both examples return the same result but using an aggregation pipeline made things better performance-wise.
vi. For more details check ref. https://www.mongodb.com/docs/manual/aggregation/

30. Project Required Fields:

a. The results of the Database queries must only have required and desired fields.
b. Exclude all the unwanted or unneeded fields from the result of the Database queries.
c. This practice should be followed for all types of Databases.
d. Manual on how to use projection in MongoDB pipelines: https://www.mongodb.com/docs/manual/reference/operator/aggregation/project/

31. Indexes in Databases:

a. Indexes support the efficient execution of queries in any Database.

b. Therefore, we must create indexes for the fields which are frequently used to access the records.
c. **For Example:**
   i. **Single Field Indexes:**

```
const database = client.db("sample_mflix");
const movies = database.collection("movies");

// Create an ascending index on the "title" field in the
// "movies" collection.
const result = await movies.createIndex({ title: 1 });
console.log(`Index created: ${result}`);
```

   ii. **Compound Indexes:**

```
const database = client.db("sample_mflix");
const movies = database.collection("movies");

// Create an ascending index on the "type" and "genre" fields
// in the "movies" collection.
const result = await movies.createIndex({ type: 1, genre: 1 });
console.log(`Index created: ${result}`);
```

d. For more details check ref.
   https://www.mongodb.com/docs/manual/indexes/

32. Auto Compress Images:

a. In most of the projects, we integrate the feature for users to upload images (profile images, product images, etc). We should create a method to automatically compress the images before storing them on the server. One way is to integrate some packages and take the images through that method for compressing them. An example package that can be integrated with Node.js is:
   https://www.npmjs.com/package/tinypng
b. Compressing the images will help reduce the size of the images, and hence optimize the applications.

~~33. Serving Images / Static Files through CDN:~~

~~a. For all the static images and files being included in the project, we should use CDNs for delivering the resources. CDNs help in optimizing the applications.~~
~~b. As an example, Cloudinary or S3 can be used as CDN:~~

c. ~~https://cloudinary.com/~~
d. ~~https://www.npmjs.com/package/cloudinary~~
e. ~~https://aws.amazon.com/s3/~~
f. ~~https://www.npmjs.com/package/@aws-sdk/client-s3~~

34. Time/Space Complexity:

a. Writing optimized code is the most important thing for any application. Reducing the time and space complexity will make the application more optimized.
b. On a very basic level, writing more loops increases the time complexity.
c. Writing more lines of code increases space complexity.
d. We should try to reduce the time and space complexity wherever possible.
e. An example is explained below to better understand this thing.
f. **Program Requirements:**
    i. https://bitbucket.org/!api/2.0/snippets/tawkto/aA8zqE/4f62624a75da6d1b8dd7f70e53af8d36a1603910/files/webstats.json
    ii. The endpoint above returns the chats stats collected against unique website ids.
    iii. Each entry contains:
        1. Website Id
        2. Date
        3. Total number of chats
        4. Total number of missed chats
        5. There will only be one entry for a website per day.
    iv. For example, if there are a total of 5 websites for a period of 14 days, there will be 70 entries in the file.
    v. We need to write a program that will retrieve data from the provided URL and output the sum of chats and missed chats per website Id for a given date range. Date filtering is optional. If date filters are not provided, the entire dataset needs to be aggregated. Partial date filtering should also be supported. I.e. If only a start date is provided - data is filtered from a given start date without end date filtering.
    vi. **Example:** Without date range function call
        1. processStatistics()
    vii. **Example:** With date range function call
        1. processStatistics('2019-04-05', '2019-04-12')
        2. processStatistics('2019-04-05')
        3. processStatistics('', '2019-04-12')
    viii. Below is the code that is completely working fine for the requirements:

```
const axios = require('axios');
```

```
const processStatistics = async function (startDate, endDate) {
    let response = await fetchAllRecords()
   if (response && response.data) {
       const websiteStats = response.data;
       let processedStats;
       // Fetch data without date filters
       if(!startDate && !endDate) {
           const aggregate = {};
           websiteStats.forEach((website) => {
               if (aggregate[website.websiteId]) {
                   aggregate[website.websiteId].chats += website.chats;
                   aggregate[website.websiteId].missedChats += website.missedChats;
                   delete aggregate[website.websiteId].date
               } else {
                   aggregate[website.websiteId] = website;
                   delete aggregate[website.websiteId].date
               }
           });
           processedStats = Object.values(aggregate);
           console.log(processedStats);
           return processedStats;
       }
       // Fetch data with date filters
       let start = startDate ? new Date(startDate) : ''
       let end = endDate ? new Date(endDate) : ''
       let filterWebsites = {}
       let filteredData;
       if(start && end) {
           filteredData = websiteStats.filter(item => new Date(item.date) >= start
&& new Date(item.date) <= end)
       }
       else if(start && !end) {
           filteredData = websiteStats.filter(item => new Date(item.date) >= start)
       }
       else if(!start && end) {
           filteredData = websiteStats.filter(item => new Date(item.date) <= end)
       }
       filteredData.forEach((website) => {
           if (filterWebsites[website.websiteId]) {
               filterWebsites[website.websiteId].chats += website.chats;
               filterWebsites[website.websiteId].missedChats +=
website.missedChats;

           } else {
```

```
                filterWebsites[website.websiteId] = website;
                delete filterWebsites[website.websiteId].date
            }
        });
        processedStats = Object.values(filterWebsites);
        console.log(processedStats);
         return processedStats;
    }
    else {
        return [];
    }
}
const fetchAllRecords = async() => {
 try{
 const response = await
axios.get('https://bitbucket.org/!api/2.0/snippets/tawkto/aA8zqE/4f62624a75da6d1b8d
d7f70e53af8d36a1603910/files/webstats.json');
 return response;
 }
 catch(e) {
   return false;
 }
}
processStatistics();
```

g. This code gives the desired results, but in terms of optimizations, this is not a good code.
h. The code manages the conditions for with dates and without dates separately. Which is adding up additional lines of code, thus increasing the utilized space.
i. **The part where it manages the stats for the condition without date ranges.**

```
// Fetch data without date filters
        if(!startDate && !endDate) {
            const aggregate = {};
            websiteStats.forEach((website) => {
                if (aggregate[website.websiteId]) {
                    aggregate[website.websiteId].chats += website.chats;
                    aggregate[website.websiteId].missedChats += website.missedChats;
                    delete aggregate[website.websiteId].date
                } else {
                    aggregate[website.websiteId] = website;
                    delete aggregate[website.websiteId].date
                }
```

```
            });
            processedStats = Object.values(aggregate);
            return processedStats;
        }
```

j. **The part where it is managing the stats for the condition with date ranges.**

```
// Fetch data with date filters
        let start = startDate ? new Date(startDate) : ''
        let end = endDate ? new Date(endDate) : ''
        let filterWebsites = {}
        let filteredData;

        if(start && end) {
            filteredData = websiteStats.filter(item => new Date(item.date) >= start
&& new Date(item.date) <= end)
        }
        else if(start && !end) {
            filteredData = websiteStats.filter(item => new Date(item.date) >= start)
        }
        else if(!start && end) {
            filteredData = websiteStats.filter(item => new Date(item.date) <= end)
        }

        filteredData.forEach((website) => {
            if (filterWebsites[website.websiteId]) {
                filterWebsites[website.websiteId].chats += website.chats;
                filterWebsites[website.websiteId].missedChats +=
website.missedChats;

            } else {
                filterWebsites[website.websiteId] = website;
                delete filterWebsites[website.websiteId].date
            }
        });
        processedStats = Object.values(filterWebsites);
         console.log(processedStats);
         return processedStats;
        }
        else {
          return [];
        }
```

k. Every character takes 1 byte, and if there are let's say 200 characters that are additional, they are adding up 200 bytes of additional space to the program. Which is definitely not the optimized way in terms of space utilized.

l. **Let’s optimize this solution to reduce the space complexity.**

```
const axios = require("axios");
const fetchAllRecords = async () => {
   try {
     const response = await axios.get(

"https://bitbucket.org/!api/2.0/snippets/tawkto/aA8zqE/4f62624a75da6d1b8dd7f70e53af
8d36a1603910/files/webstats.json"
     );
     return response;
   } catch (e) {
     return false;
   }
};
const filterData = (start="", end="", data)=> {
 start = start ? new Date(start) : ""
 end = end ? new Date(end) : "";
 if (start && end) {
 return data.filter(
   (item) => new Date(item.date) >= start && new Date(item.date) <= end
 );
 } else if (start && !end) {
   return data.filter(
     (item) => new Date(item.date) >= start
   );
 } else if (!start && end) {
   return data.filter((item) => new Date(item.date) <= end);
 }
 else
   return data;
}
const processStatistics = async function (startDate, endDate) {
 let response = await fetchAllRecords();
 if (response && response.data) {
   let filteredData = filterData(startDate, endDate, response.data);
   const aggregate = {};
   filteredData.forEach((website) => {
       if (aggregate[website.websiteId]) {
         aggregate[website.websiteId].chats += website.chats;
         aggregate[website.websiteId].missedChats += website.missedChats;
```

```
        delete aggregate[website.websiteId].date;
      } else {
        aggregate[website.websiteId] = website;
        delete aggregate[website.websiteId].date;
      }
    });
    console.log(aggregate);
    return Object.values(aggregate);
 } else {
   return [];
 }
};
processStatistics();
```

m. Initially, this program was taking around 65 lines and now reduced to 50. This has optimized the program for the space complexity.
n. Now we need to optimize the program for time complexity. This program currently has 2 loops. We need to reduce the loops to improve the performance of the code.

```
return data.filter(
   (item) => new Date(item.date) >= start && new Date(item.date) <= end
 );
```

```
filteredData.forEach((website) => {
```

o. In the code below we have optimized it more enough to remove one loop. It has also reduced a little more space as well.

```
const axios = require("axios");
const fetchAllRecords = async () => {
   try {
     const response = await axios.get(

"https://bitbucket.org/!api/2.0/snippets/tawkto/aA8zqE/4f62624a75da6d1b8dd7f70e53af
8d36a1603910/files/webstats.json"
     );
     return response;
   } catch (e) {
     return false;
   }
};
function isWithinFilterRange(start, end, websiteDate)  {
 if (start && end)
   return websiteDate >= start && websiteDate <= end;
 if (start && !end)
```

```
    return websiteDate >= start;
  if (!start && end)
    return websiteDate <= end;
  return false;
}
const processStatistics = async function (startDate, endDate) {
  let response = await fetchAllRecords();
  if (response && response.data) {
    const websiteStats = response.data;
    startDate = startDate ? new Date(startDate) : ""
    endDate = endDate? new Date(endDate) : "";
    const aggregate = {};
    websiteStats.forEach((website) => {
      if(isWithinFilterRange(startDate, endDate, new Date(website.date))) {
        if (aggregate[website.websiteId]) {
          aggregate[website.websiteId].chats += website.chats;
          aggregate[website.websiteId].missedChats += website.missedChats;
          delete aggregate[website.websiteId].date;
        } else {
          aggregate[website.websiteId] = website;
          delete aggregate[website.websiteId].date;
        }
      }
      });
      return Object.values(aggregate);
  } else {
    return [];
  }
};
processStatistics('2019-04-05', '2019-04-12');
```

p. The program is now 45 lines, with 1 loop.
q. **We should try to reduce the space and time wherever it is possible.**

35. Try-catch Statements:

a. Try-catch statements are used in JS to handle unwanted errors during the execution of a program.
b. **Try:**
    i. The code block to be tested for errors while the program is being executed is written in the try block.
c. **Catch:**
    i. The code block that is executed when an error occurs in the try block is written in the catch block

```
try {
   // write code here
} catch (error) {
   // write code here
}
```

36. **Error & Exception Handling:**
    a. In any application, there will be situations where something goes wrong, such as an external API call returning an error, a database query failing, or input validation failing. It's important to properly handle these errors to ensure that the application remains stable and the user is informed of the problem.
    b. Handle exceptions carefully.
    c. In case of exceptions, return appropriate error messages to the user along with the error so that it can be fixed or corrected later.
37. **Git Necessities:**
    a. We should configure and use git properly with our project. Required actions are listed in the document below: https://docs.google.com/document/d/13uWJ5c0iSiU-9YdTcO64JyacHN3cmqWfIp4pLIPQSXM/edit