Code Review and Architecture

Event buds

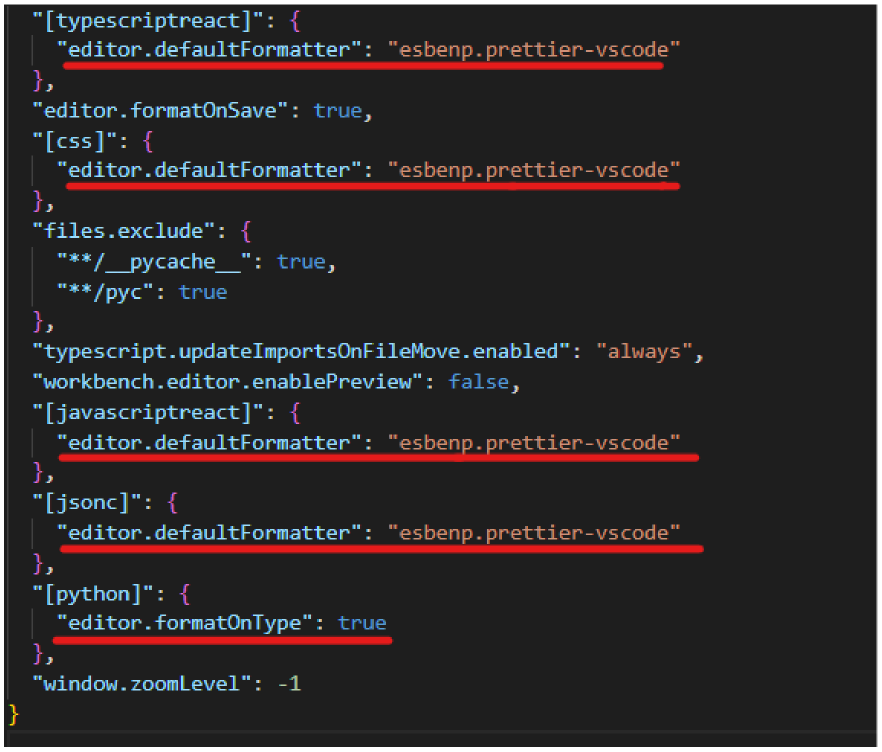
Alish Kadiwal and Ria Chevli

Table of Content

Code Formatting:

Alignment

We are using Python and React JS typescript Framework as our project stack. Since we are using Python, the code needs to be formatted and indented correctly to make python code run correctly, so we did not have to do much indentation or Alignment fixes on our Application Programming Interface (API). On our Client Side, we are using the Prettier library that auto indents based on pre adjusted settings that we select. We have selected Typescript or Javascript settings and it auto indents and align functions, HTML tags and component tags using proper format.



```
  "[typescriptreact]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "editor.formatOnSave": true,
  "[css]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "files.exclude": {
    "**/__pycache__": true,
    "**/pyc": true
  },
  "typescript.updateImportsOnFileMove.enabled": "always",
  "workbench.editor.enablePreview": false,
  "[javascriptreact]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "[jsonc]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "[python]": {
    "editor.formatOnType": true
  },
  "window.zoomLevel": -1
}
```

Spacing

Spacing after the function and length of code per line is also determined by the Prettier library. Upon save, it auto formats and excess of words per line is auto set to another line and for imports, it uses one package or component per line to make it accessible and easier to read.

Naming Convention

There is some naming inconsistency across different projects. But once we are familiar with the terms, it makes it easier to use and without much thought. We have installed the Sonar detection tool into our visual studio code and it helps us check code and inconsistency. In Python, we are using PEP 8 naming style as per sonar suggestions and python documents. For the data that are retrieved from the database are all caps. It makes it easier to recognize which variables are retrieved from Database and updated. On our front end, we are using camel case naming type which is widely used and accepted sonar format. We did not have any issues with naming convention, but later on we were thinking of adding a data transformation object (DTO) on API side which will transform all caps column names to camel case or PEP 8 style. We might also convert camelcase to PEP 8 to have consistency across all projects and types.

Clean Code

We believe in code documentation, where we do not have to add comments and additional blobs for function explanations. The code should read for itself. We have made sure that we do not have console logs and any print debug statements in our code and we did it a rule for checking for any debug statement before we push the code to the main branch. In case any statements are left by mistake, we remove them as a priority as soon as possible.

Architecture

Design Pattern:

We are using controller-service-repository pattern which is one of the famous model that was carried from Spring bot and gradually other libraries like C# projects started to implement them. Some of the organizations have also adapted this design pattern. The pattern was easy and simple to understand and it takes a very small amount of time for someone who is not well versed with different patterns. This is very closely Similar to MVC models. In MVC pattern, the functionality is separated on Model for database interaction, View on front-end for displaying of data and Controller which is mediator between those to manage the data retrieval, handling and processing. In the Controller-Service-Repository pattern, we are only dealing with API and its code segregation. Controller manages the REST API and exposes only required business functions. Service Layer handles all business logic. Lastly, Repository is similar to Model in MVC and deals with storing the data in a database and retrieving data.

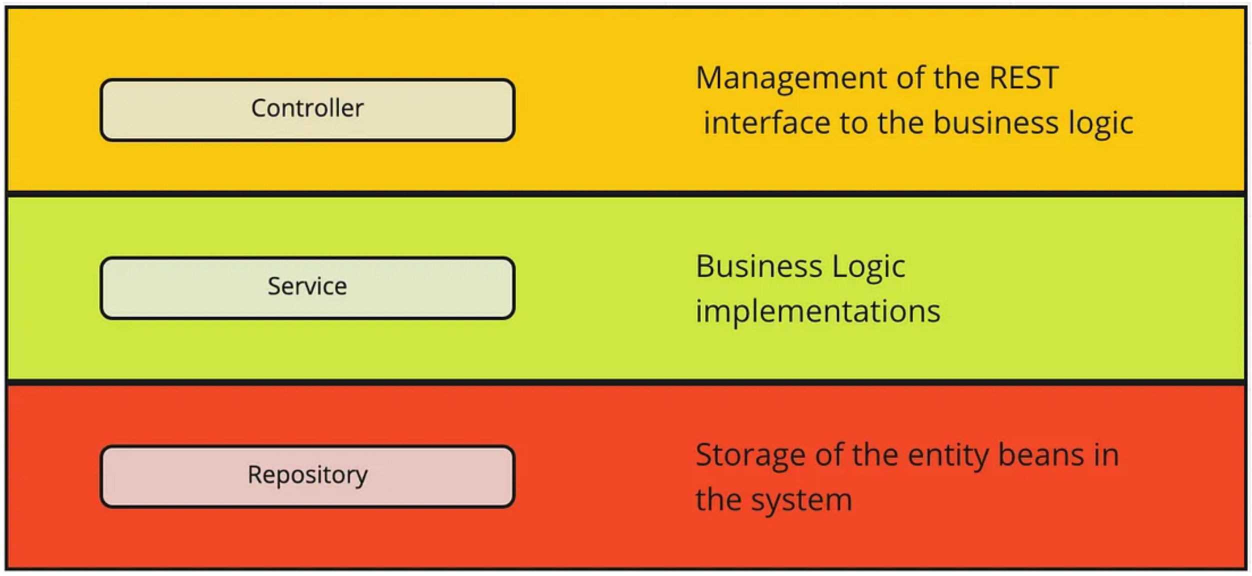

Controller
Management of the REST interface to the business logic
Service
Business Logic implementations
Repository
Storage of the entity beans in the system

Fig 1: Controller-Service-Repository Pattern (Collings, 2021)

Technology:

As mentioned before, we are using React typescript, Python Fast API for our API service and Oracle Cloud VM for hosting our API and our website. Finally, we are using SQL Autonomous database from Oracle because of its compatibility and document and community support. We chose React Typescript as we believed that strict type handling will help us quickly detect any error and be easy to debug. Although, we are not using the types to the best of its extent as there are open or any types which makes it dynamic. We have unfortunately used that in many places, but during refactoring, we will also take a look at each function and make the variable types as strict as required.

Infrastructure Setup:

We have used Oracle cloud VM for our infrastructure and set up everything from base in a linux machine. Our API is using Red hat enterprise linux which is labeled under their different operating System name. We have installed python libraries and we are currently running the script from its var/www or apache folder in the variable directory.

We are doing the same for the UI project, after we have configured the folder within var to use SSL ports and setting our domain name. In the UI project, we just have to add the build folder created by react to the project folder and the whole project will run as expected. There were additional rules required which are stored in .htacess file in the project folder

Future Infrastructure Refactors:

We are manually updating the files in the project directory for API and UI. This method is too time consuming and there might be concerns with sync consistency. To avoid this issue, we wanted to create docker containers into our projects and we can use the same images that are created on those projects. This process will guarantee consistency and will save a lot of time.

Code Best Practices:

Separation Of Concern:

We have properly separated each module and front end function such that there is no other function that does a similar process and the existing function does not break or interfere with other functions. Each function's specifications are segregated into their groups. Following the above image example, we have achieved separation of API handling, Business Logic and Database interaction. That Controller-Service-Repository model clearly specifies separate duties for each function. We have also applied this similar concept to our Front end code. The codes are separated within the function and each variable that is used within that function is initialized in that function component. In our Front end, we have separated components within the pages and the components are called instead of populating pages. There is still more work required in order to split existing large functions to helper functions or components.

Minimize Code Reuse:

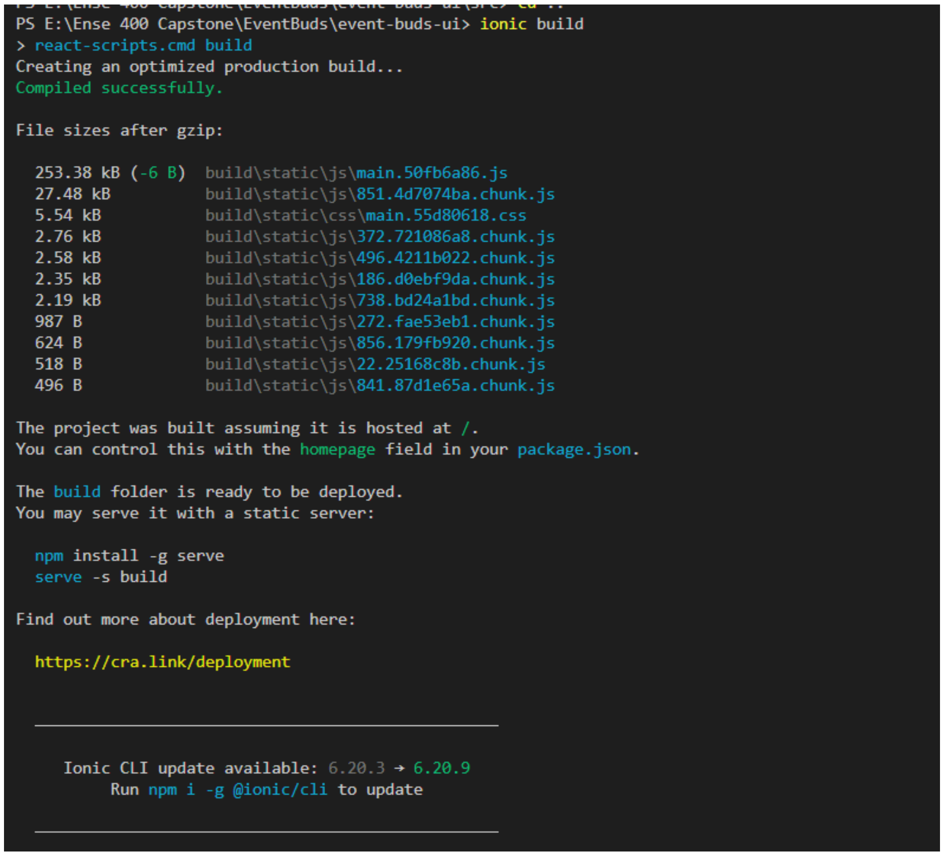
We have tried to minimize code reuse as low as minimum as possible. On our front-end, we are reusing splitted functions and trying to minimize code reuse, variable reuse and logical functionality. We have refactored the same logic to a function and used just one variable and initialized it in just one place instead of recalling the same variable that has been passed from parent component.

Removing Unnecessary Comments:

We have tried to remove unnecessary comments and there are no comments or help comments in the UI project. We believe that code should read for itself and the logic is easier to follow after going through it once. Although, we do have a couple of comments just a signifier and print error in our catch blocks in case something goes wrong and to keep note of that. We also have connections completed for the database when the API is connected to the database. This helps us with monitoring the connection and it only happens once on startup. We will be looking for alternatives and adding monitor logs to make it easier to get notified upon failures.

Code Report:

We were planning to import sonarqube to be installed into our project or any open source tool for code checking and reports. We already have sonar lint which detects the issue within the projects locally and gives us warning. We are using that for detecting any potential issues for python, HTML, CSS, JS and Typescript files. We also found that React has its own package when it runs the compiler and checks for any warnings or issues within the code. We have used that and also implemented strict mode to warn us about bad practice. We have tried to minimize them, but it would still give us warning in a couple of places that we have already tested for fail safe. We have ignored them for now and they are commented using //eslint to disable warnings for that particular funciton.



```
PS E:\Ense 400 Capstone\EventBuds\event-buds-ui> ionic build
> react-scripts.cmd build
Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  253.38 kB (-6 B)  build\static\js\main.50fb6a86.js
  27.48 kB          build\static\js\851.4d7074ba.chunk.js
  5.54 kB           build\static\css\main.55d80618.css
  2.76 kB           build\static\js\372.721086a8.chunk.js
  2.58 kB           build\static\js\496.4211b022.chunk.js
  2.35 kB           build\static\js\186.d0ebf9da.chunk.js
  2.19 kB           build\static\js\738.bd24a1bd.chunk.js
  987 B             build\static\js\272.fae53eb1.chunk.js
  624 B             build\static\js\856.179fb920.chunk.js
  518 B             build\static\js\22.25168c8b.chunk.js
  496 B             build\static\js\841.87d1e65a.chunk.js

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  https://cra.link/deployment


  ------------------------------------------------


     Ionic CLI update available: 6.20.3 → 6.20.9
          Run npm i -g @ionic/cli to update

  ------------------------------------------------
```

Non Functional Requirements:

Security

We have maintained a security aspect for protecting user data wherever required. We are masking user passwords with bcrypt which uses 256 bit encryption to mask the password and the password is not logged anywhere within our log files. Our VM's are also well secured and there is no exposure of app security key or VM's passkey.

Usability

We feel like the app and functionality usability is quite high. From Developers perspective, the architecture and similar setup can be used for other projects if we intended. Also we have used the latest project tools stack. If we want to enhance it further, it will take minimal efforts to enhance it and we don't have to worry about language support or its existence.

Performance

From the performance perspective, we think that our low end machines can easily handle api from hundreds of users at a time without much lag. From a small group of users, we had 320 ms at the highest in completing one request and that time could vary from request to request. Most of the function is using the Async process. To some extent, this will ensure that user experience is not interrupted. From our tests, it does seem pretty stable for a small group of users, however extending to tens of thousands of users overnight will affect our virtual machines processing capacity. We have to increase our machine's specifications to support our backend.

Scaling

Our application is easy to expand and there should be no issues in the case of expanding to tens of thousands of users. However, according to our test, our current machines would not be able to manage them. We might need to horizontally scale the virtual machine by extending its ram and processing power. We have also researched that another library that supports FastAPI called Gunicorn can add workers which will help with vertical scaling. This would help us with extending our api and application to a number of users without impacting any performance.

Maintenance

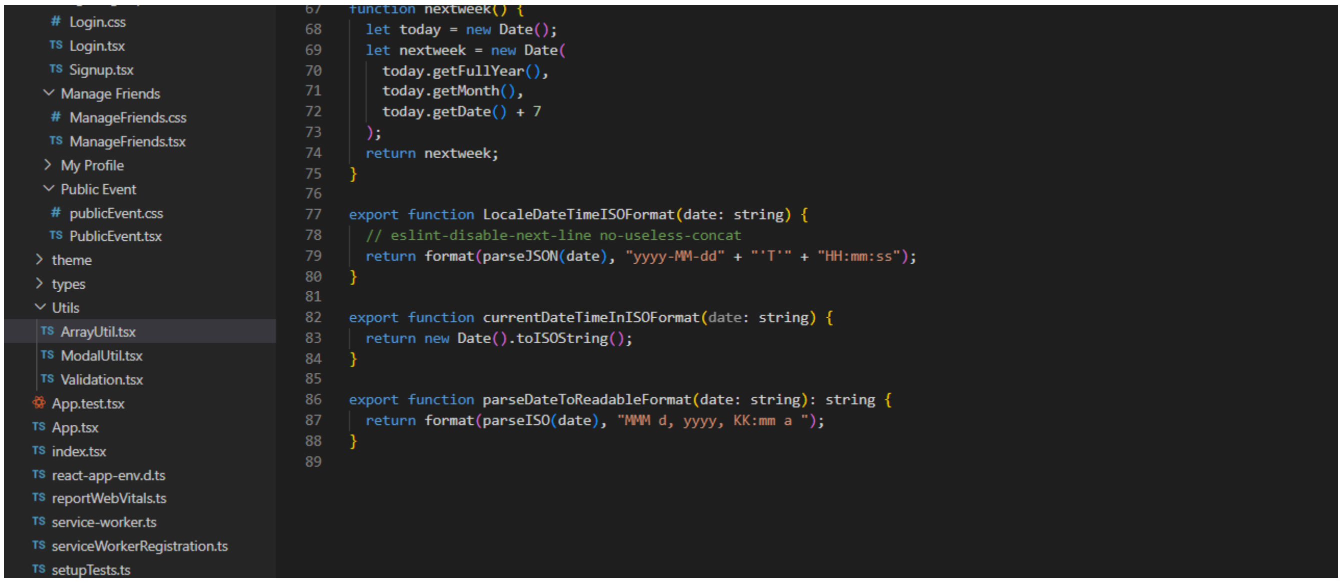
Our application should require less fix or no fixes upon extending existing functionality. We have also placed handlers for fail safe, so that if the property is modified somewhere else or if the variable doesn't exist, it won't crash the whole application. Instead, it will show default empty strings or no values if the variable value or variable doesn't exist. We have fixed many bugs along the way and fixing one bug would not break other unrelated functionality as our functions are independent from one another and focus on just one concern.

Reliability

We have added and tested those functionality in the sprints, bazar day, and final capstone presentation day. We are handling and taking measures that we assign explicit data types for variables and also validating the types on API and UI. We did not observe any inconsistency with data storage and retrieval and no unexpected errors on UI and API side as well. We can say that our application is reliable and up to standard practise. We still need some sessioning on API as the database closes connection after the project and api connections are idle for a long time and needs to be rebooted.

Reusability

On our UI, we have created a util folder which holds functions logic that are common across different components. This way, we are using one function with the same logic on different components and saving us from unnecessary code smells. We are also using the same components with minimum adjustment if just the data passed within the function differs. We are using the Same Modal Function for handling guests and helpers as they behave similarly.



```
function nextweek() {
  let today = new Date();
  let nextweek = new Date(
    today.getFullYear(),
    today.getMonth(),
    today.getDate() + 7
  );
  return nextweek;
}

export function LocaleDateTimeISOFormat(date: string) {
  // eslint-disable-next-line no-useless-concat
  return format(parseJSON(date), "yyyy-MM-dd" + "'T'" + "HH:mm:ss");
}

export function currentDateTimeInISOFormat(date: string) {
  return new Date().toISOString();
}

export function parseDateToReadableFormat(date: string): string {
  return format(parseISO(date), "MMM d, yyyy, KK:mm a ");
}
```

Code Coverage:

Testing:

We did not implement any unit test or integration test for code coverage, because from the past experience we are sure that developing test cases requires the same amount of time developing features and functionality. Although we don't have a unit test cases suite, we are testing all our features by manual testing and keeping track of each test case after new code has been pushed to production.

Object Oriented Design Principle:

1. **Single Responsibility Principle (SRS):** Our architecture is well set up in order to implement the single responsibility principle. As mentioned in previous topics, we have implemented the Controller-Service-Repository model to separate and distinguish the concerns within those models. Also, we have used components within pages and also separating functions in some place to refactor or extract out common or distinguishable logic within the component which makes it better to read the code and understand it.
2. **Open Closed Principle:** Our application at this point in stage is very small in terms of scope and implementations. We believe that we have separated the functions according to our needs and created small functions. At the current point in stage, implementing a new feature or editing is easy, as we have to just expose and edit one function. However, with increasing scope, we will separate existing classes into parent children inheritance and separating bases from new derived classes.
3. **Liskov substitutability principle:** We do not have child-parent relations in our API model. Later on, we can refactor our backend project to have child parent data relationships for example friends within the child class and separating out the child entities to be base class and inheriting that property for friends, helpers and guests.

4. **Interface segregation:** We have separated interfaces and functions to be smaller in length and only sharing the data that is required. We still need to make some refactors to make our functions and classes into smaller interfaces.
5. **Dependency Inversion:** Currently, we don't have abstract interfaces. However, we have researched regarding the interface implementation and we will be calling those interfaces instead of creating an instance in every class.

References

Collings, T. (2021, August 10). *Controller-service-repository*. Medium. Retrieved April 3, 2023, from https://tom-collings.medium.com/controller-service-repository-16e29a4684e5