Code Management

Software implementation of planned user stories:

Phase 3:

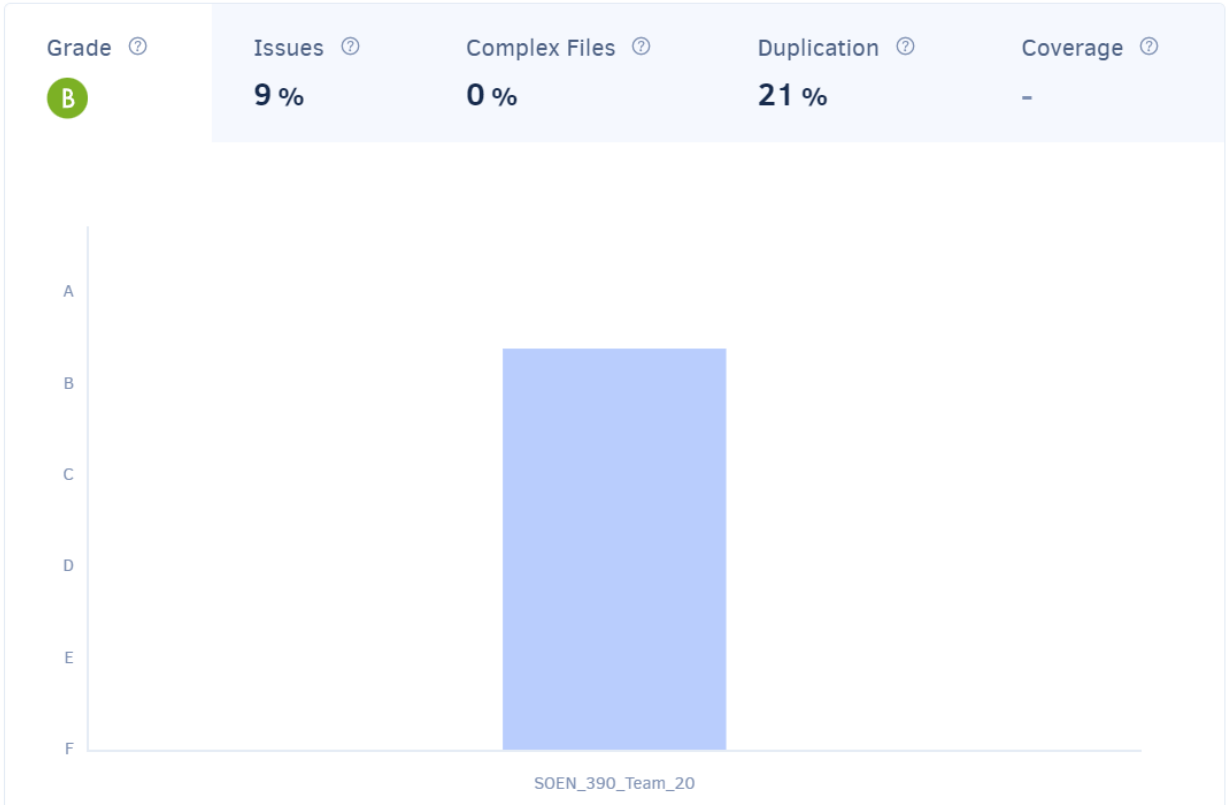
Sprint 3							
Request system	User Stories	Hassan	8	29/2/24	19/3/24	10	Complete
Requests endpoints	Sub user story	Hassan	3/8	29/2/24	1/3/24	2	Complete
Request submitter form	Sub user story	Vraj	3/8	1/3/24	3/3/24	2	Complete
Request list view (employee)	Sub user story	Vraj	2/8	1/2/24	3/3/24	2	Complete
Database web-socket connection for notification	Enhancement	Shivam		1/3/24	10/3/24	9	Complete
Add new role: Owner/Renter	Enhancement	Kaothar		1/3/24	3/3/24	2	Complete
Navigation/Routing	Enhancement	Dimitri		2/3/24	11/3/24	9	Complete
User roles management (testing)	User Stories	Shivam	7	2/3/24	10/3/24	8	Complete
Landing Page	Enhancement	Aly		19/3/24	21/3/24	3	Complete
Dashboard	Enhancement	Kaothar		19/3/24	21/3/24	3	Complete
Admin role tests	Sub user story	Omar	3/7	4/3/24	15/3/24	11	Complete
Public user tests	Sub user story	Jackson	2/7	2/3/24	15/3/24	13	Complete
Employee tests	Sub user story	Aly	2/7	2/2/24	15/3/24	13	Complete

Total effort = 71 story points
Story points completed in Iteration #1 = 4
Story points completed in Iteration #2 = 21
Story points completed in Iteration #3 = 15
Average per iteration = 13.33 story points / iteration
# of iterations = 71 / 13.33 = 5.325.

Since we were allotted 5 iterations to complete the project and most of the work done in the first sprint was focused on documentation, we believe we are on schedule to complete the project within the time provided.

Code Review:

Using Codacy, we ran some tests and received a grade of B. For sprint #4 we can work on reducing our code duplication since it is slightly elevated at 21%



Design Pattern:

For our database we are using a factory pattern implementing IDBController interface.

```
// DBControllerFactory.js
import DBController from "../controllers/DBController";
import { IDBController } from "../types/DBTypes";

/* The `DBControllerFactory` class in TypeScript provides a static method to create
instances of
`DBController`. */
Comment Code | Improve Code
class DBControllerFactory {
  /**
   * The function `createInstance` returns a new instance of `DBController` implementing
   the
   * `IDBController` interface.                                    • CMD+L for Code
   * @returns An instance of the `DBController` class is being returned.
   */
  static createInstance(): IDBController {
    return new DBController();
  }
}

export default DBControllerFactory;
```

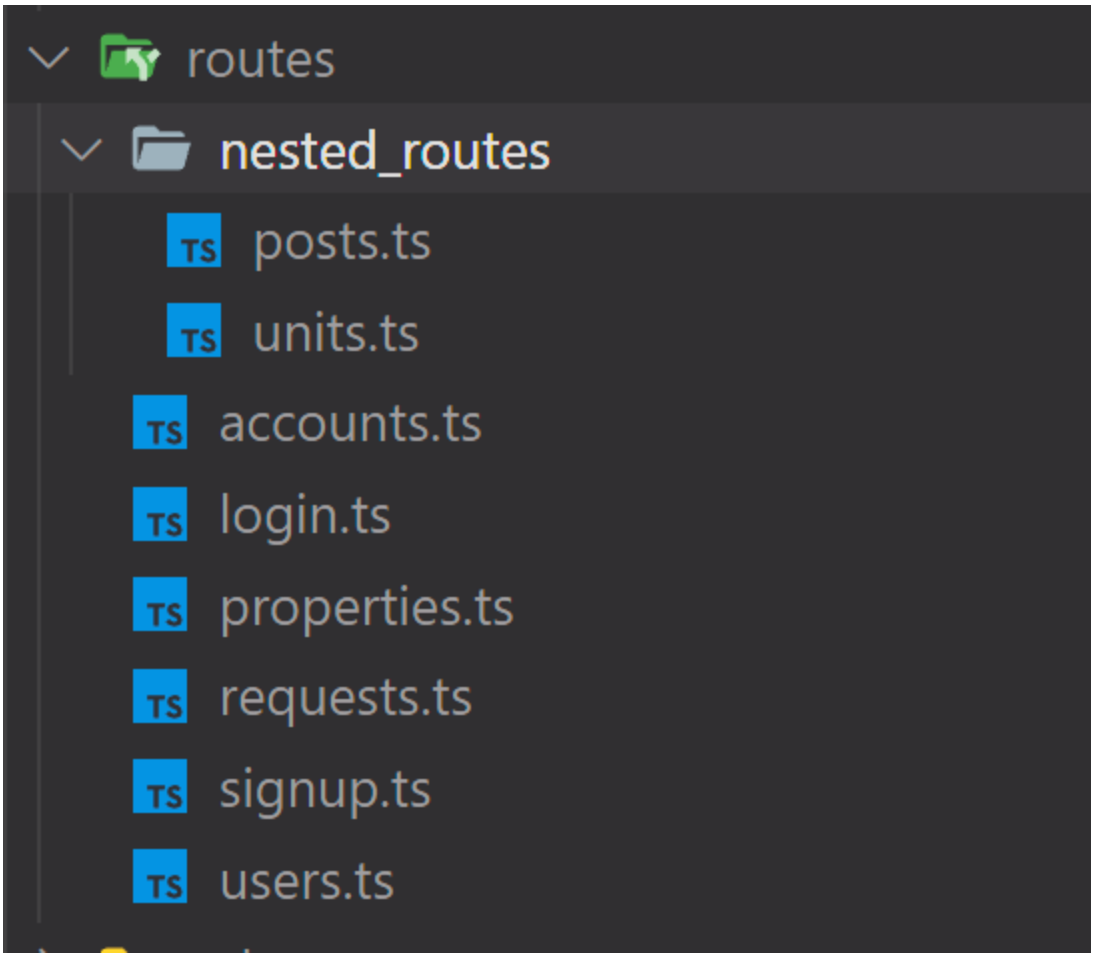
We then use Master classes so access the database with the dbController and return data to be routed:

```
accountsMaster.ts
postsMaster.ts
propertyMaster.ts
requestsMaster.ts
unitMaster.ts
```

```
class AccountsMaster {
  readonly dbController: IDBController; // You might want to re
  actual type of dbController

  constructor() {
    this.dbController = DBControllerFactory.createInstance();
  }
```

```
async getUserDetails(
  email: string,
  password: string
): Promise<{ status: number; data: PublicUserData } | Error> {
  let result = await this.dbController.getPublicUser(email, password);
  if (result.message) return new Error(result.message);

  return result as { status: number; data: PublicUserData };
}
```

For access to the data we used a RESTful API design that creates routes where data can be accessed in a 3 layer system. Here is our routes:


routes
nested_routes
posts.ts
units.ts
accounts.ts
login.ts
properties.ts
requests.ts
signup.ts
users.ts

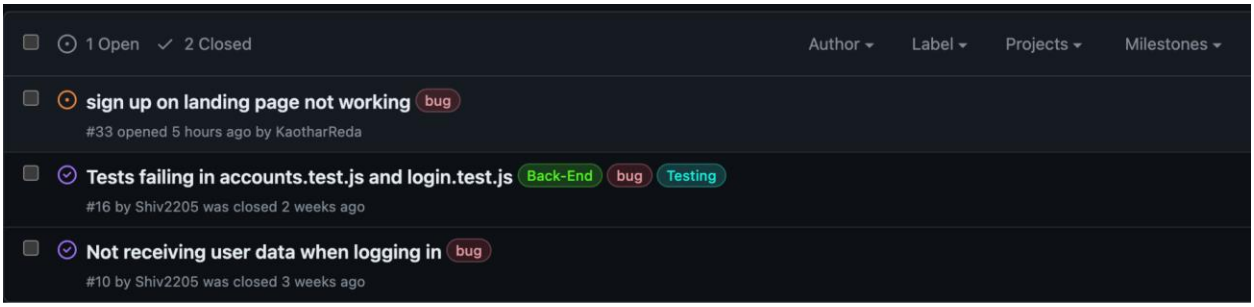
Example of a route “/requests/unit” that expects a unit_id in the body and returns the requests related to that unit:

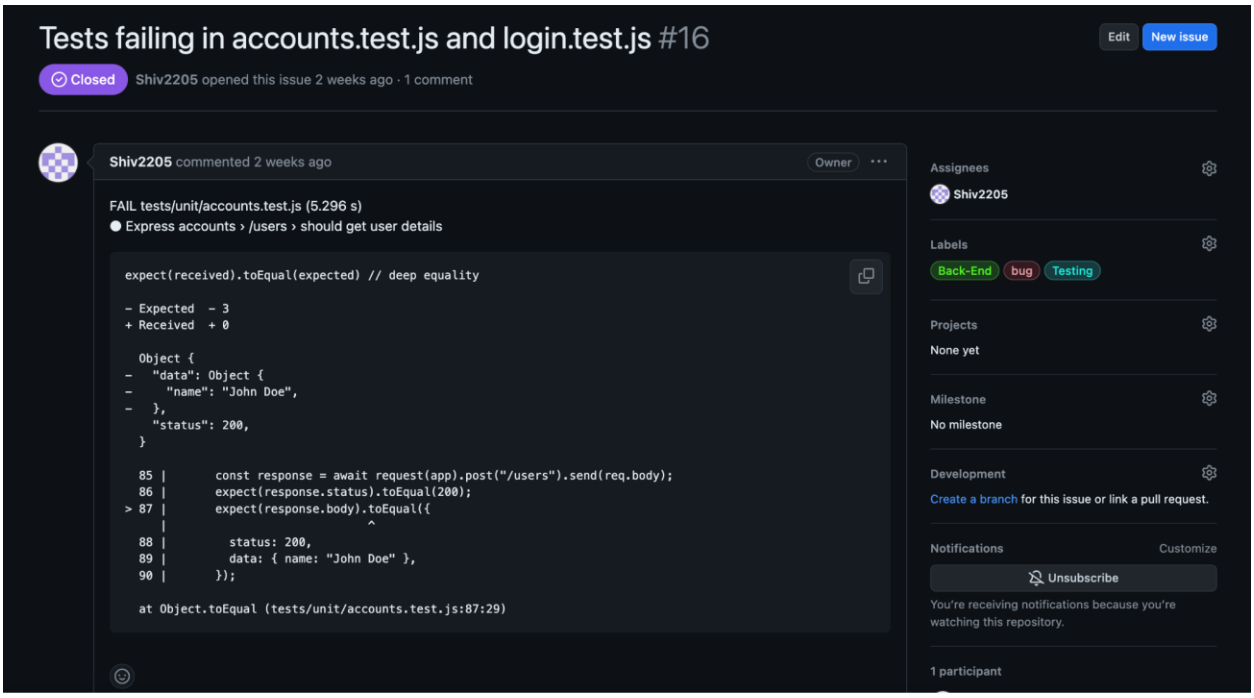
```
router.post(
    "/unit",
    Comment Code | Improve Code
    async function (
        req: Request<{}, {}, { unit_id: string }>,
        res: Response<{ status: number; data?: RequestDetails[] } | { response: string }>,
        next: NextFunction
    ) {
        const { unit_id } = req.body;

        try {
            const result = await requestsMaster.getAllUnitRequests(unit_id);
            if (result instanceof (Error)) {
                throw result as Error;
            }
            res.status(result.status).send(result);
        } catch (error) {
            res.status(500).send({ response: (error as Error).message });
        }
    }
);
```

These can be used by the frontend as API calls and have the data returned.

Bug Reports:

We are using github to document, label, categorize and assign our bugs.





Feature Branches:

Since phase 1 of this project, the team has used the idea of feature branches. For every new feature, enhancement or bug we are working on, we create a separate branch from main and work on it. Once we add our new code, we check whether it conflicts with the original code in any way causing errors, bugs or breaks. If nothing is triggered, we are safe to merge back with the original main branch.

Code Coverage:

We tested coverage for our controller classes as well for some other components. For our controller class we maintain roughly 75 - 80 % coverage which is very good. This is a great indicator that our tests extensively cover our code base.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	84.14	62.58	86.2	89.35	
Factory	100	100	100	100	
DBControllerFactory.ts	100	100	100	100	
controllers	76.47	66.27	79.76	79.88	
DBController.ts	76.47	66.27	79.76	79.88	418-442,582-697
repo	76.62	6.25	95	95.16	
accountsMaster.ts	82.14	0	100	100	36-108
postsMaster.ts	80	25	100	100	34-67
propertyMaster.ts	80	0	100	100	22-52
unitMaster.ts	63.15	0	80	80	53-55
routes	97.45	70	100	99.09	
accounts.ts	100	85.71	100	100	21
login.ts	94.11	66.66	100	94.11	35
properties.ts	94.28	50	100	100	27-59
signup.ts	100	75	100	100	25
routes/nested_routes	89.74	84.61	87.5	90.62	
posts.ts	100	80	100	100	10
units.ts	81.81	87.5	77.77	83.33	46-52
tests/unit/utils	100	100	100	100	
recordExistsTest.js	100	100	100	100	
types	100	100	100	100	
DBTypes.ts	100	100	100	100	

Coding Guides:

We recently switched from javascript to typescript and within that transition adopted the google typescript coding style. Reference for this coding style is found here : https://google.github.io/styleguide/tsguide.html