CampusLoop Backend Technical Document

1. Introduction

CampusLoop is a university-based marketplace platform designed to enable students to buy, sell, and rent items inside the university ecosystem. The backend of CampusLoop is built using **Node.js**, **Express**, **TypeScript**, and **MongoDB**, following a **Modular MVC Architecture Pattern**.

This report explains the backend implementation approach, development standards, authentication logic, wallet flow, error handling strategy, and major design decisions. As this is a lab report, the explanation focuses on concepts, code patterns, architecture decisions, and implementation details without attaching full source code.

2. Backend Architecture Overview

The backend follows a **Modular MVC** structure where each module is completely isolated and contains its own:

- Routes
- Controllers
- Services
- Models
- TypeScript interfaces
- Validation logic

Directory Structure Example

```
src/
  modules/
    auth/
      auth.route.ts
      auth.controller.ts
      auth.service.ts
      auth.model.ts
      auth.interface.ts
    item/
```

```
        item.route.ts
        item.controller.ts
        item.service.ts
        item.model.ts
        item.interface.ts
    utils/
    config/
    middlewares/
    app.ts
    server.ts
```

This structure ensures maintainability, testability, and clean separation between layers.

3. Key Design Decisions

3.1 TypeScript Enforcement

TypeScript is used across all modules for:

- Enforcing strict typing
- Improving code reliability
- Detecting errors at compile time
- Ensuring consistent data contracts between layers

All important entity definitions (Role, Status, ItemCategory, JWT payloads, Wallet structure) are stored in `.interface.ts` files.

4. Authentication & Authorization System

The platform uses role-based authentication with four defined roles:

Role	Description
BUYER	Can buy/rent items
SELLER	Can sell/rent items
ADMIN	Internal moderation

SUPER_ADMIN	Full system control

4.1 JWT Authentication Flow

1. The route calls `checkAuth(...allowedRoles)` middleware.
2. The token is extracted from either:
    - `req.headers.authorization`
    - `or req.cookies.accessToken`
3. Token is verified using JWT.
4. User status must be ACTIVE.
5. Decoded payload (email, role, id) is attached to `req.user`.
6. Controller receives a fully authenticated context.

Token Generation Example

```
export const generateToken = (payload: JwtPayload, secret:string, expiresIn:string)=>{
    const token = jwt.sign(payload, secret, {
        expiresIn
    } as SignOptions)
    return token
}

export const verifyToken = (token: string, secret: string) =>{
    const verifiedToken = jwt.verify(token, secret)
    return verifiedToken
}
```

4.2 University Email Validation

All users must register using a valid **@cse.bubt.edu.bd** email.
The system extracts a student ID from the email prefix using `extractUniversityId()` before creating the account.

```
export const extractUniversityId = (email:string):string=>{
    if(!email.endsWith("@cse.bubt.edu.bd")){
    throw new AppError(httpStatus.BAD_REQUEST, "Please Register with BUBT Edu mail");
    }
    const universityId = email.split("@")[0];

    // check if it's all digits (since student IDs are numeric)
    if (!/^\d+$/.test(universityId)) {
        throw new Error("Invalid student ID format in email.");
    }
    return universityId
}
```

5. Request Handling & Error Management

CampusLoop strictly follows a **centralized error handling mechanism**.

5.1 Controller Pattern (using catchAsync)

Every controller is wrapped with `catchAsync`:

```
import { NextFunction, Request, Response } from "express";

/* eslint-disable @typescript-eslint/no-explicit-any */
type AsyncHandler = (req: Request, res: Response, next: NextFunction) => Promise<void>

export const catchAsync = (fn: AsyncHandler) => (req: Request, res: Response, next: NextFunction) => {
    Promise.resolve(fn(req, res, next)).catch((err: any) => {
        next(err)
    })
}
```

5.2 Error Throwing Convention

Errors are thrown using `AppError`:

throw new AppError(httpStatus.BAD_REQUEST, "Invalid student email");

The global error handler receives all thrown errors and formats them consistently.

```ts
class AppError extends Error {
    public statusCode: number;

    constructor(statusCode: number, message: string, stack = '') {
        super(message) // throw new Error("Something went wrong")
        this.statusCode = statusCode

        if (stack) {
            this.stack = stack
        } else {
            Error.captureStackTrace(this, this.constructor)
        }
    }
}

export default AppError
```

6. Standard Response Structure

All successful responses follow a single convention through `sendResponse()`:

```ts
export const sendResponse = <T>(res: Response, data: TResponse<T>) => {
    res.status(data.statusCode).json({
        statusCode: data.statusCode,
        success: data.success,
        message: data.message,
        meta: data.meta,
        data: data.data
    })
}
```

This ensures:

- uniform API responses
- easier frontend integration
- improved debugging and testing consistency

7. Database Layer (MongoDB + Mongoose)

Mongoose Models

Each module has its own model file following typed schema definitions.

```
const itemSchema = new Schema<IItem>(
  },
  {
    timestamps: true,
    versionKey: false,
  }
);

export const Item = model<IItem>("Item", itemSchema);
```

8. Route Registration

All feature modules are combined in a single routing hub:

```
export const router = Router()

const moduleRoutes = [
    {
        path:'/user',
        route: UserRoutes,
    },{
        path: '/auth',
        route: AuthRotues
    },{
        path: '/item',
        route: ItemRoutes
    },{
        path: '/rent',
        route: RentRoutes
    },{
        path: '/admin',
        route: AdminRoutes
    }
]


moduleRoutes.forEach((route) => {
  router.use(route.path, route.route);
});
```

In `app.ts`, everything is prefixed with `/api/v1`.

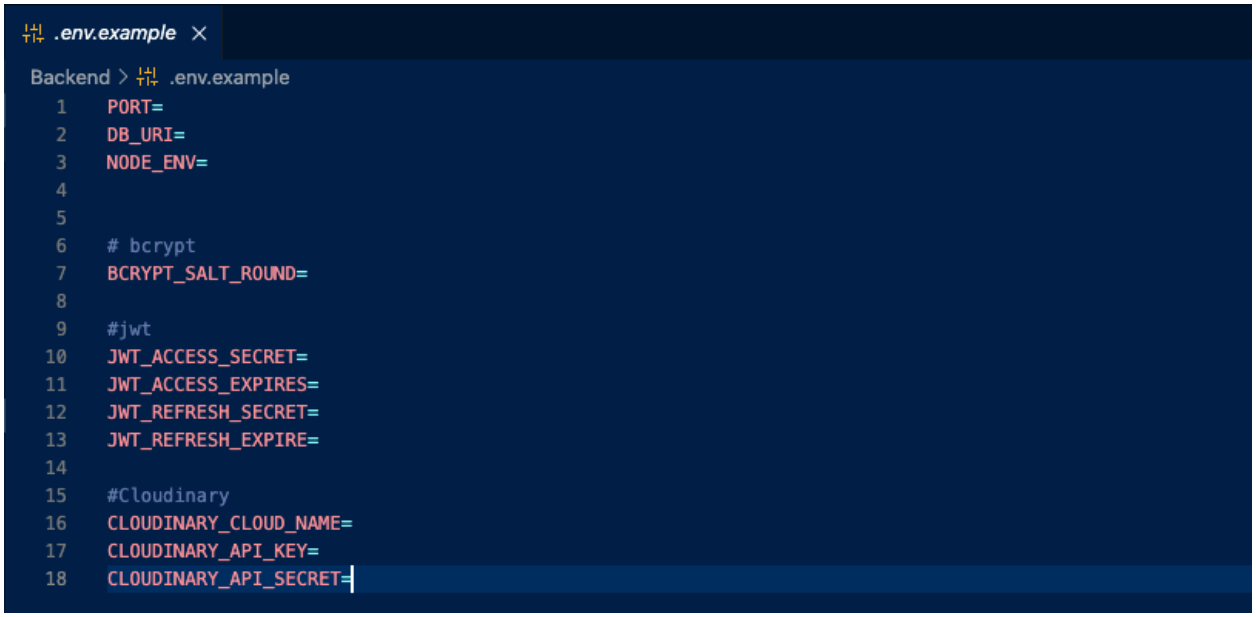
9. Environment Configuration

Environment variables are validated at startup to prevent missing keys.

Required `.env` fields:

- Database credentials
- JWT secrets & durations
- Cloudinary credentials
- Server port
- Bcrypt salt rounds

If any variable is missing, the server does not start.



10. Development Workflow

Scripts

Command	Description
`npm run dev`	Development with auto-reload
`npm run build`	Compiles TypeScript to JavaScript
`npm start`	Starts production server

Deployment (Vercel)

- Build output: `dist/`
- Entry file: `dist/server.js`
- Vercel configuration handled via `vercel.json`