🎯 Assignment 3: Customizing the Go API Codebase to Our Application

⚙️ Environment Setup

Item	Details
Operating System	Linux
Programming Lang	Go
Git Repo	https://github.com/Muditha-Kumara/Go/tree/task2
Original Project Name	Vehicle Behavior Monitoring System
Git Repo	https://github.com/Muditha-Kumara/IntelligentDevices

Project Structure:

```
API 0.1/
  cmd/api/main.go
  internal/api/handlers/
  internal/api/middleware/
  internal/api/repository/
  internal/api/server/
  internal/api/service/
```

📝 Introduction

This report documents the process of customizing the Go backend codebase for intelligent devices for Vehicle Behavior Monitoring Project, as per Assignment 3. The main focus was to redesign the data entity, update the database schema, adjust handlers, add validation, and implement logic to handle rapid POST requests.

1. Codebase Exploration

The project uses a Handler-Service-Repository-Model pattern:

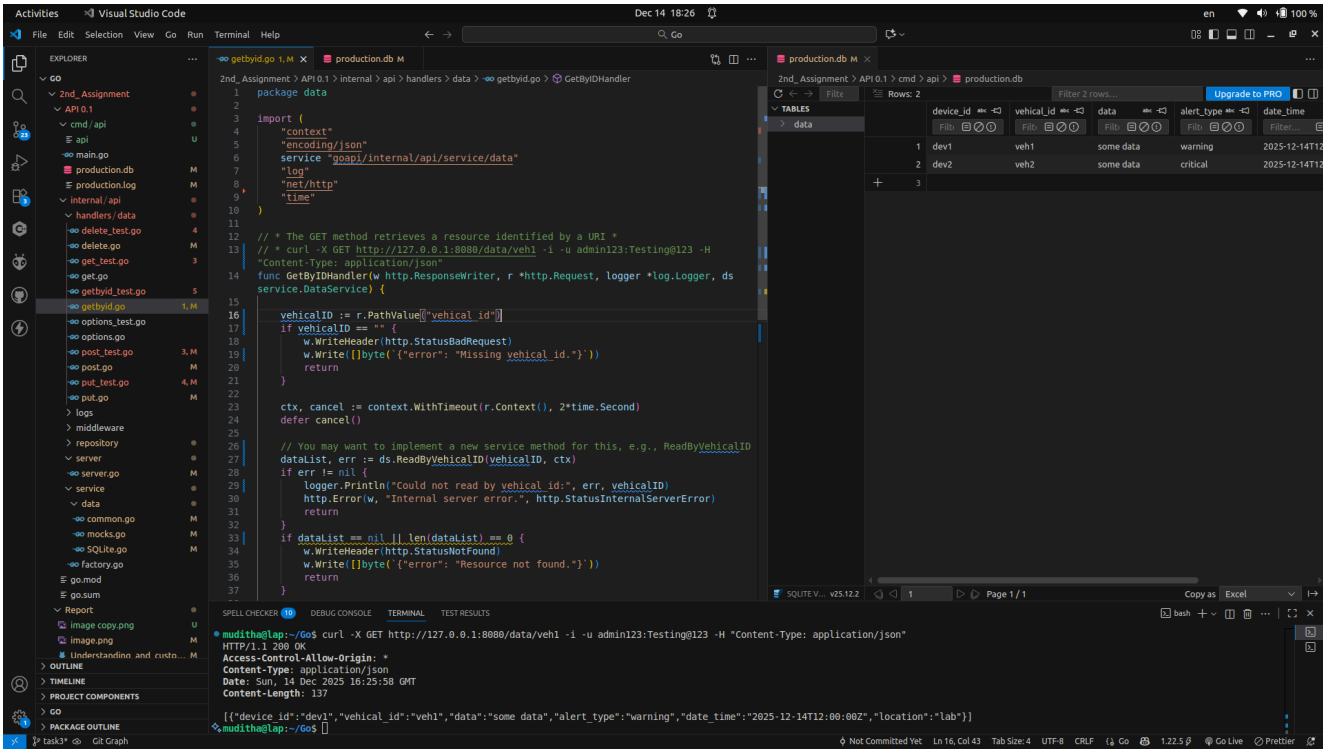
- **Handler Layer:** HTTP request/response logic (`internal/api/handlers/data/`)
- **Service Layer:** Business logic and validation (`internal/api/service/data/`)
- **Repository Layer:** Database access (`internal/api/repository/DAL/SQLite/`)
- **Model Layer:** Data structures (`internal/api/repository/models/`)

2. Defining the Device Data Entity

New Data Entity:

```
type Data struct {
    DeviceID   string
    VehicalID  string
    Data       string
    AlertType  string
    DateTime   string
    Location   string
}
```

- `vehical_id` (note: spelling as per your structure) and `alert_type` are new fields.
- `data` is a string for device sensors data.
- `location` and `date_time` are included for context.

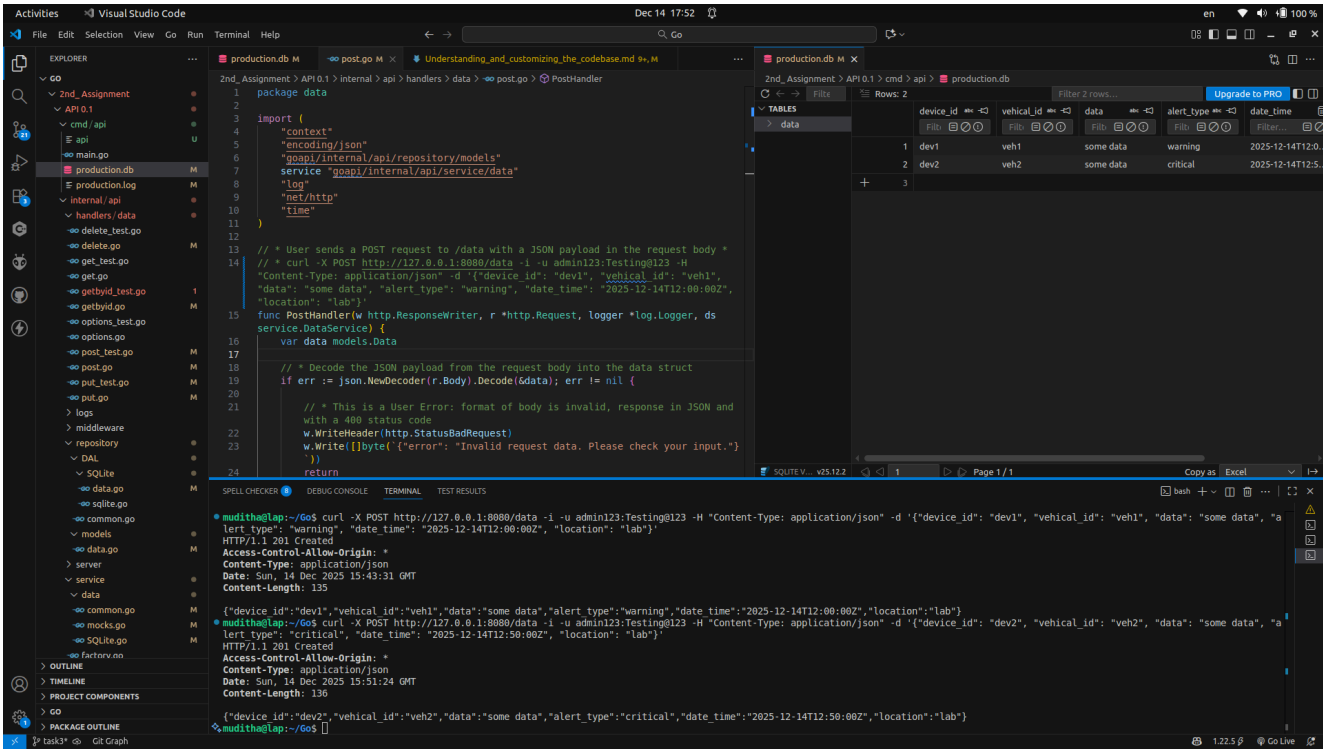


3. Implementing Custom Data Entity and Handlers

- The new struct was added to the models.
- The SQLite table schema was updated to match the new fields.
- All CRUD handlers in `internal/api/handlers/data/` were updated to use the new structure.
- The POST handler was modified: if two POST requests are received from the same device in less than 1 minute, the API responds with a "warning" message instead of "ok" message. Next the device should configure according to post reply.

Example POST Payload:

```
{
  "device_id": "dev3",
  "vehical_id": "veh3",
  "data": "some data",
  "alert_type": "warning",
  "date_time": "2025-12-14T12:00:00Z",
  "location": "lab"
}
```



4. Setting Up and Customizing Validators

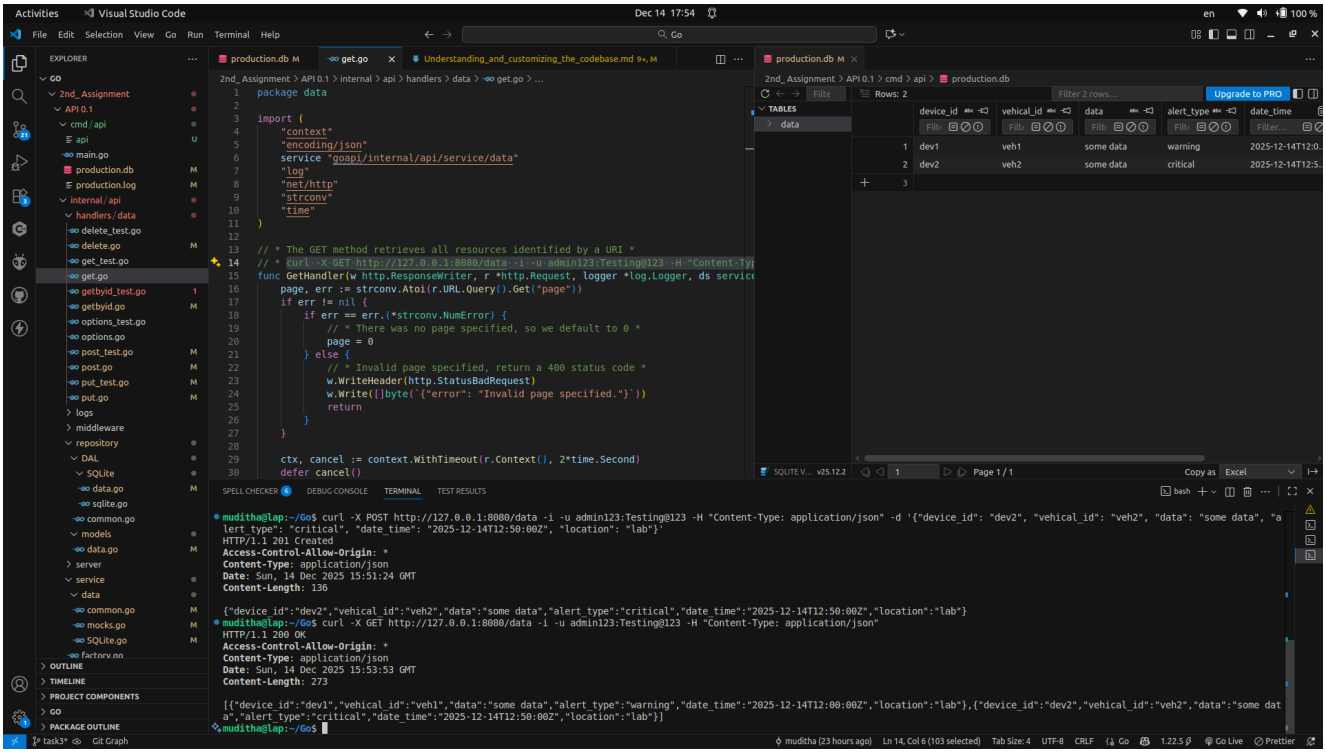
- Validators were implemented to check required fields (`device_id` , `vehical_id` , `data` , `date_time`).
- Additional checks: `alert_type` must be one of the allowed values (e.g., "info", "warning", "critical").
- Validation is performed before any database operation in the service layer.

5. Integrating Device with Backend

- Devices send data to the backend using the new POST endpoint.
- The backend responds with a message after process.

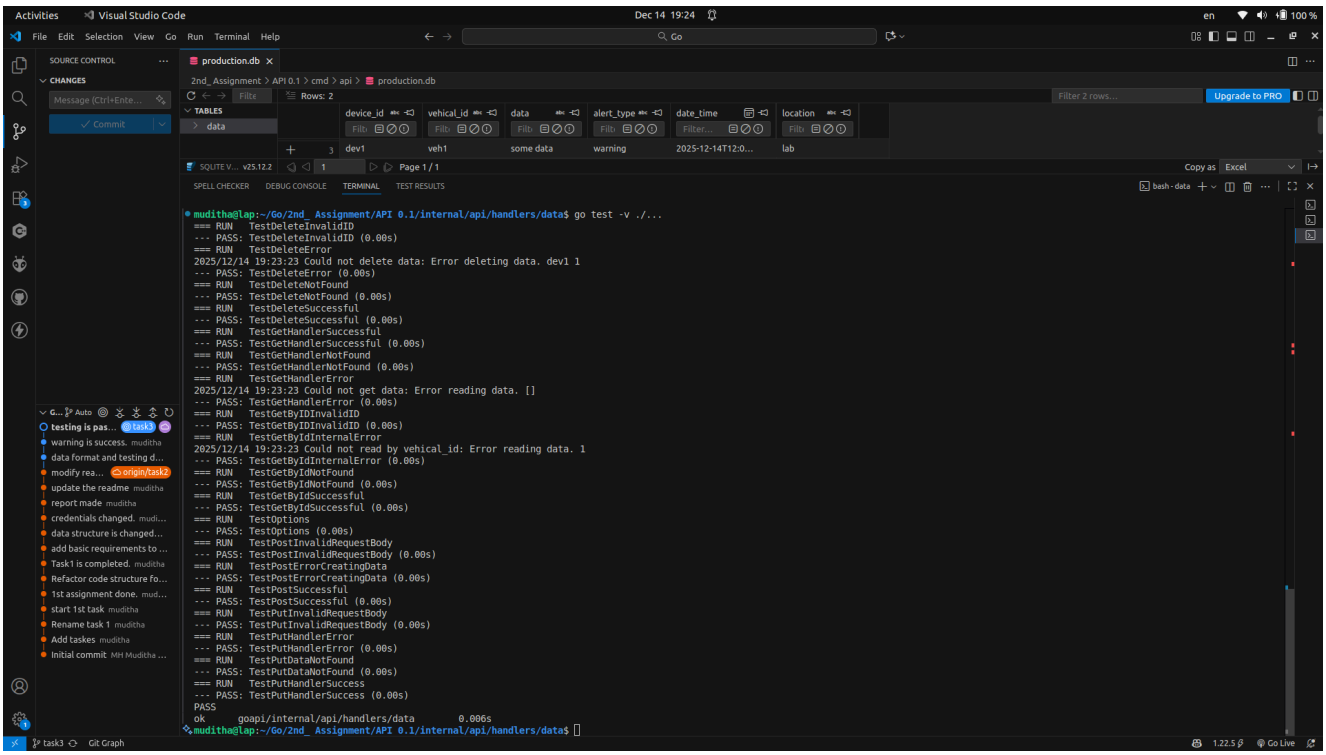
6. Advanced: Adding Intelligence

- The POST handler logic prevents rapid submissions (less than 1 minute apart) from the same device, send warn massage to device and inform to driver.



7. Testing and Debugging

- Unit tests were updated for the new data structure and logic.
- Manual API testing was performed using Thunder Client/Postman.
- The rapid POST logic was tested by sending two requests within 1 minute and confirming the correct response.



8. Documentation and Reflection

Customization Steps

- Redesigned the data entity and updated all layers (model, handler, service, repository).
- Updated the database schema and migration logic.
- Implemented new validation and anti-flooding logic in the POST handler.
- Updated and ran all tests.

Reflection

This assignment deepened my understanding of Go backend architecture, especially in customizing data flows and enforcing business rules. The main challenge was ensuring the rapid POST detection logic worked reliably and updating all layers to match the new data structure. I improved my skills in RESTful API design, validation, and test-driven development. For future improvements, I would consider adding rate-limiting middleware and more granular error handling.

✅ Conclusion

- The Data entity and all related layers were successfully customized.
- The database schema and handlers were updated.
- Validators and Warn logic were implemented.
- All changes were tested and verified.