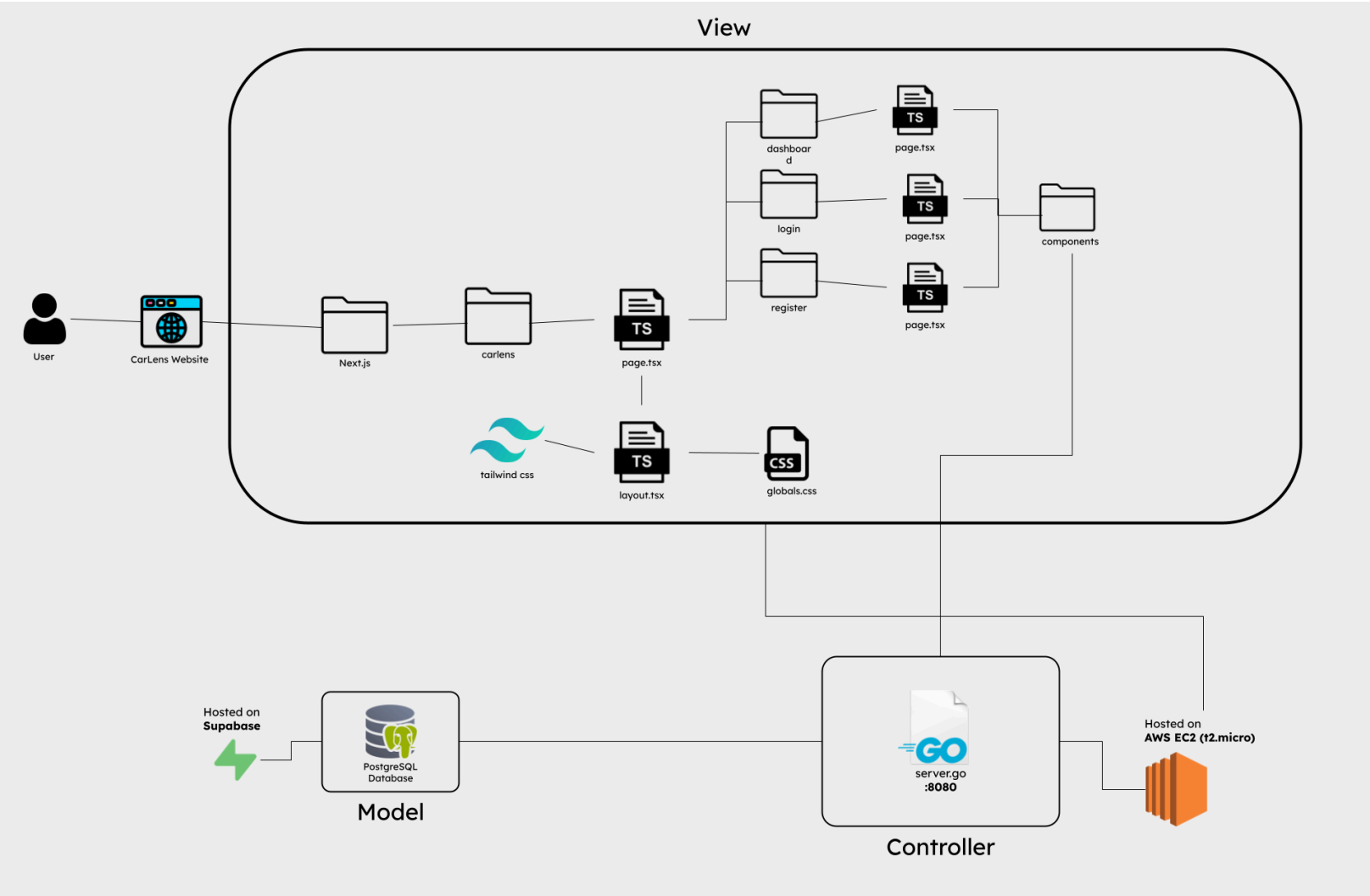
Team 7

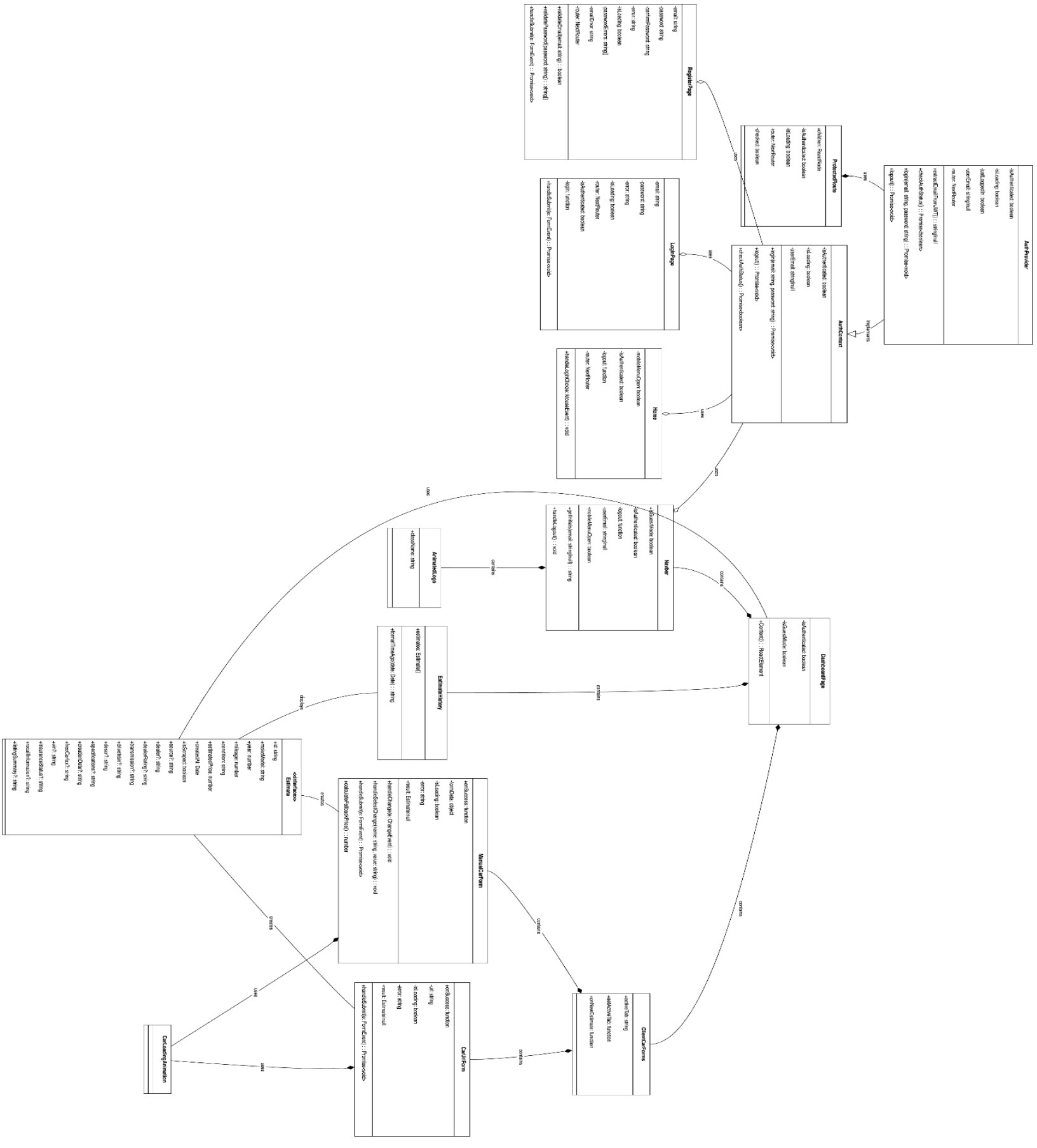
SENG 401 – Final Project - Design Document

Architecture



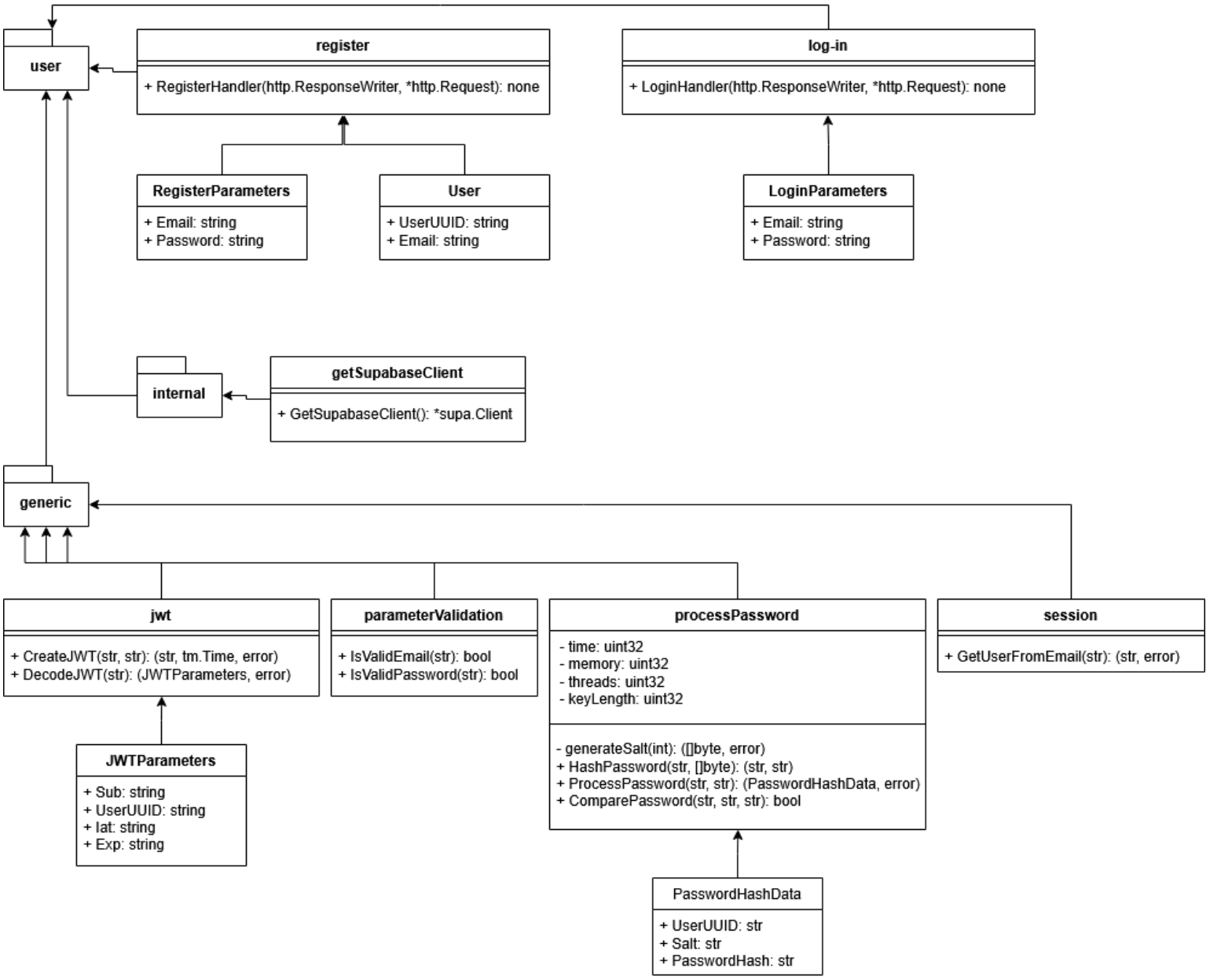
Team 7

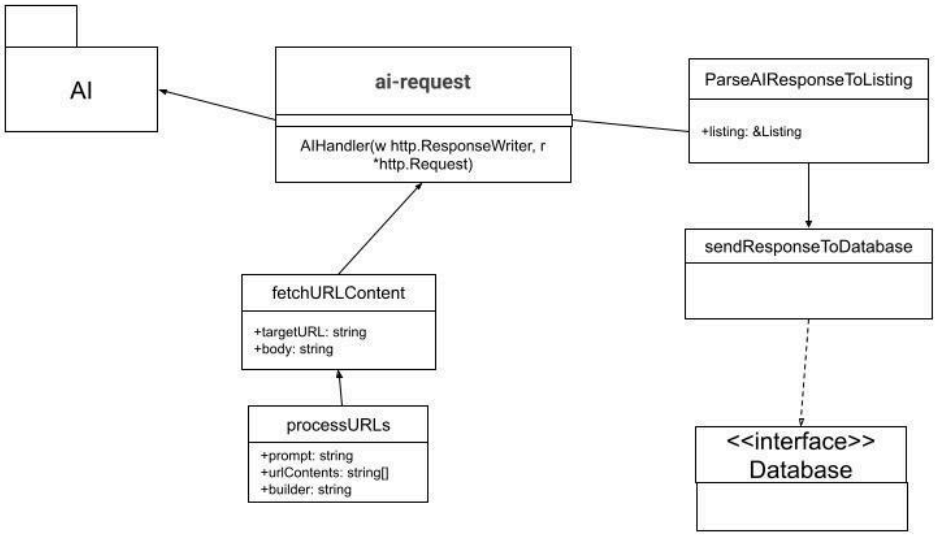
Class Diagram



Team 7

Golang Server UMLs





Design Pattern

The design pattern our project uses is MVC. We have distinct layers that handle each aspect of the MVC pattern and here is a breakdown of each.

Model

For the model, we have gone with PostgreSQL which is hosted on Supabase. We interact with the database via its API. We provide our controller (server.go) the API URL and key to allow it to interact with the tables.

View

For the view, we chose to create a frontend using Next.js, and since it is a website, it is cross-platform since both desktop and mobile can interact with it. We have made sure the website is dynamic and handles different resolutions well. The frontend interacts with the backend using HTTP requests, and for authenticated users, it provides a JWT via cookies for authorization.

Controller

For the controller, we have set up a Go API which handles register, log-in and AI generation requests. There are 3 different routes to the API (/log-in, /register and /ai). The Go API is the only entity that has access to the SQL database (model) to ensure proper MVC procedure. The frontend has to contact the API to get anything done. The API is hosted on an AWS EC2 instance (t2.micro, 750 free hours per month). To dive a little deeper, the register and log-in functionality is written from scratch, but maintains industry standards in terms of security. The passwords are securely hashed using the argon2 hashing algorithm, which has won the most secure hashing algorithm competition in 2015. These hashes are also salted with a randomly generated 128 bit sequence, which ensures 2 of the same passwords don't end up with the exact same hash. (ex. if user1 and user2 both put in Password123! as their password, without salts, they would both have the same resultant hash, but with a salt, they both with come out of the algorithm with different hashes, ensuring no 2 hashes are the same, even if the stored passwords are). Secondly, the log-in endpoint returns a JSON web token as a cookie if the user was authenticated correctly. This token acts as session persistence, so even if the user refreshes the page or logs in the next day, they do not need to re-login. This token expires in 7 days.

SOLID Principles

Single Responsibility

Single Responsibility was used for functions to serve one purpose. The following functions are examples of well-defined, specific and easy-to-understand functions that improve maintainability.

```
func GetSupabaseClient() *supa.Client {

    supabaseURL := os.Getenv("SUPABASE_URL")
    supabaseKey := os.Getenv("SUPABASE_KEY")

    if supabaseURL == "" || supabaseKey == "" {
        fmt.Println("missing SUPABASE_URL or SUPABASE_KEY environment
variables")
        return nil
    }

    supabase := supa.CreateClient(supabaseURL, supabaseKey)

    return supabase
```

```
}
```

The responsibility of this code is solely to connect to the supabase database using environment variables.

```
func DecodeJWT(tokenString string) (JWTParameters, error) {

    claims := jwt.MapClaims{}

    token, err := jwt.ParseWithClaims(tokenString, claims, func(token
*jwt.Token) (interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("ERROR: Unexpected signing method: %v",
token.Header["alg"])
        }
        return []byte(secretKey), nil
    })
    if err != nil {
        return JWTParameters{}, err
    }

    if token.Valid {
        jwtParams := JWTParameters{
            Sub: fmt.Sprintf("%v", claims["sub"]),
            Iat: fmt.Sprintf("%v", claims["iat"]),
            Exp: fmt.Sprintf("%v", claims["exp"]),
        }
        return jwtParams, nil
    }

    return JWTParameters{}, fmt.Errorf("ERROR: Token is invalid")

}
```

The responsibility of this code is solely to decode the JWT code to assist in user token storage and confirmation during data requests.

Open-Close

Open-Close was used in parts of the code where new functionality could be added without needing to modify the original code. The following function is based on the interface above it and can be edited to add anything to the form without modifying the original interface or function.

```
export interface CarManualFormProps {
  onSuccess: (estimate: Estimate) => void;
}

export function CarManualForm({ onSuccess }: CarManualFormProps) {
  const [formData, setFormData] = useState({
    make: "",
    model: "",
    year: new Date().getFullYear().toString(),
    mileage: "",
    condition: "good",
    features: "",
  });
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState("");
  const [result, setResult] = useState<Estimate | null>(null);

  const handleChange = (e: React.ChangeEvent<HTMLInputElement |
HTMLTextAreaElement>) => {
    const { name, value } = e.target;
    setFormData((prev) => ({ ...prev, [name]: value }));
  };

  const handleSelectChange = (name: string, value: string) => {
    setFormData((prev) => ({ ...prev, [name]: value }));
  };

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    setIsLoading(true);
    setError("");
    setResult(null);
```

```
    try {
      // Validate input
      if (!formData.make || !formData.model || !formData.mileage) {
        throw new Error("Please fill out all required fields");
      }

      // Simulate API call delay
      await new Promise(resolve => setTimeout(resolve, 1500));

      // Calculate a mock price based on inputs
      // In a real app, this would come from your DeepSeek AI integration
      const year = parseInt(formData.year);
      const mileage = parseInt(formData.mileage);
      const currentYear = new Date().getFullYear();

      // Very simple mock algorithm for demo purposes
      const basePrice = 30000;
      const yearFactor = (currentYear - year) * 1000;
      const mileageFactor = mileage * 0.05;
      const conditionFactor =
        formData.condition === "excellent" ? 2000 :
          formData.condition === "good" ? 0 :
            formData.condition === "fair" ? -2000 : -4000;

      const mockPrice = Math.max(5000, basePrice - yearFactor -
mileageFactor + conditionFactor);

      const mockEstimate: Estimate = {
        id: generateId(),
        makeModel: `${formData.make} ${formData.model}`,
        year: parseInt(formData.year),
        mileage: parseInt(formData.mileage),
        condition: formData.condition,
        features: formData.features,
        estimatedPrice: Math.round(mockPrice),
        createdAt: new Date(),
        isScraped: false
      };

      setResult(mockEstimate);
```

```
      onSuccess(mockEstimate);
    } catch (err) {
      // Use type assertion with unknown as a safer alternative to any
      const error = err as Error;
      setError(error.message || "Something went wrong. Please try
again.");
    } finally {
      setIsLoading(false);
    }
  };
```

Liskov Substitution

Liskov Substitution was not presented as this project has the responsibilities of a micro services system. Golang as the backend relied on multiple runs throughout the code, as one function activates and ends, and so on. With these frameworks, it was not viable to use liskov substitution.

Interface Segregation

An example of interface segregation is our server.go file, rather than implementing the entire routes path, it is able to call the specific ai file, ai-request.go, in order to fulfill its dependencies.

```
package server

import (
    "fmt"
    "net/http"

    ai "github.com/MaiTra10/CarLens/api/routes/AI"
    "github.com/MaiTra10/CarLens/api/routes/user"
    "github.com/rs/cors" // Import CORS package
)

func StartServer() {

    // Define our routes
    http.HandleFunc("/login", user.LoginHandler)
    http.HandleFunc("/logout", user.LogoutHandler)
    http.HandleFunc("/register", user.RegisterHandler)
```

```
    http.HandleFunc("/ai", ai.AIHandler)

    // Enable CORS
    c := cors.New(cors.Options{
        AllowedOrigins:   []string{"http://localhost:3000"}, // Allow
frontend (your React app)
        AllowedMethods:   []string{"GET", "POST", "PUT", "DELETE",
"OPTIONS"},
        AllowedHeaders:   []string{"Content-Type", "Authorization"},
        AllowCredentials: true,
    })

    // Wrap your router with the CORS handler
    handler := c.Handler(http.DefaultServeMux)

    // Start the server with CORS-enabled handler
    fmt.Println("Server is running on Port 8080")
    err := http.ListenAndServe(":8080", handler)
    if err != nil {
        fmt.Println("Error starting server:", err)
    }

}
```

Dependency Inversion

Dependency Inversion is when the high-level and low-level modules depend on abstractions (interfaces). In golang, no abstractions were used as all functions were individually dependent. In React, the modules were simple and had one abstraction each and followed dependency inversion when inheriting/calling interfaces.

AI Usage:

Generative AI was leveraged in order to address certain aspects of the frontend development. The first of which was to make the navbar more mobile friendly by implementing a hamburger menu. Another use of it was to help fix the logout logic and understand JWT token handling. Lastly it was used for debugging by giving console.log() messages in order to ensure reliable backend and frontend connection.