Native/ ├── main.py ├── config.py ├── routes/ │ ├── **init**.py │ ├── food\_routes.py │ ├── auth\_routes.py │ ├── doctor\_routes.py │ └── test\_routes.py ├── utils/ │ ├── **init**.py │ ├── auth.py │ └── error\_handler.py ├── activity.py ├── blood\_sugar.py ├── meal\_insulin.py ├── activities.py └── enhanced\_meal\_insulin.py i am making documentation of what each file contain

**config.py:**

**Constants:**

* MONGO\_URI: MongoDB connection string ("mongodb://localhost:27017/native\_new")
* SECRET\_KEY: Application secret key for security
* APP\_TIMEZONE: Application timezone setting (UTC)
* TOKEN\_EXPIRY: JWT token expiration time (24 hours)
* ALLOWED\_ORIGINS: List of allowed CORS origins (["http://localhost:3000"])

**Objects:**

* app: Flask application instance
* mongo: PyMongo instance for database operations
* logger: Application logger instance configured with DEBUG level

**Configurations:**

* CORS: Configured for <http://localhost:3000>
* MongoDB: Configured with MONGO\_URI
* Logging: Configured with DEBUG level

**Functions:**

* get\_mongo(): Returns the MongoDB instance
* get\_logger(): Returns the logger instance

**Global Attributes:**

* app.mongo: MongoDB instance made available to the app
* app.logger: Logger instance made available to the app

**main.py:**

**Imports:**

* Flask core components from config.py
* Blueprint modules for different functionalities

**Registered Blueprints:**

* food\_routes: Food-related endpoints
* auth\_routes: Authentication endpoints
* doctor\_routes: Doctor-specific endpoints
* test\_routes: Test endpoints
* meal\_insulin\_bp: Meal and insulin calculation endpoints
* activity\_bp: Activity tracking endpoints
* blood\_sugar\_bp: Blood sugar monitoring endpoints

**Error Handlers:**

* not\_found(error): Handles 404 errors
* internal\_error(error): Handles 500 errors

**Functions:**

* before\_request(): Makes the MongoDB instance available to all blueprints

**Server Configuration:**

* Host: '0.0.0.0'
* Port: 5000
* Debug: Enabled

**Error Handling:**

* 404: Returns JSON response with "Resource not found"
* 500: Returns JSON response with "Internal server error" and logs the error

**Logging:**

* Logs blueprint registration
* Logs application startup
* Logs internal server errors

**Runtime:**

* Runs in debug mode when executed directly
* Listens on all interfaces (0.0.0.0)
* Uses port 5000

**Environment:**

* Development server with debug enabled
* CORS enabled for frontend communication
* MongoDB connection maintained throughout request lifecycle
* **food\_data.py**:
  + Constants:
    - STANDARD\_PORTIONS: A dictionary of standard portion references.
    - FOOD\_DATABASE: A basic food database with nutrient information.
    - STARCH\_LIST: A dictionary of starch-based foods.
    - STARCHY\_VEGETABLES: A dictionary of starchy vegetables.
    - PULSES: A dictionary of pulse-based foods.
    - FRUITS: A dictionary of fruits.
    - MILK\_AND\_DAIRY: A dictionary of milk and dairy products.
    - SWEETS\_AND\_DESSERTS: A dictionary of sweets and desserts.
    - SNACKS: A dictionary of snack items.
    - COMMON\_SNACKS: A dictionary of common snack items.
    - HIGH\_PROTEIN\_FOODS: A dictionary of high-protein foods.
    - HIGH\_FAT\_FOODS: A dictionary of high-fat foods.
    - INDIAN\_DISHES: A dictionary of Indian dishes.
    - CHINESE\_DISHES: A dictionary of Chinese dishes.
    - ITALIAN\_DISHES: A dictionary of Italian dishes.
    - FOOD\_CATEGORIES: A mapping between food category names and their corresponding dictionaries.
* **food\_service.py**:
  + Functions:
    - get\_food\_details(food\_name: str) -> dict: Retrieves food details by name.
    - search\_food(query: str) -> list: Searches for foods based on a query.
    - calculate\_absorption\_factor(absorption\_type, meal\_timing): Calculates the absorption factor based on food type and meal timing.
    - convert\_to\_standard\_nutrients(portion\_size, measurement\_type, food\_details, converter): Converts nutrients based on portion size and measurement type.
* **init**.py (in the routes/ directory):
  + This is an empty file that serves as a marker for Python to treat the routes/ directory as a package.
* **auth\_routes.py**:
  + Routes:
    - /login (POST): Handles user login and returns a JWT token.
    - /register (POST): Handles user registration.
    - /dashboard (GET): Retrieves the user's dashboard information.
  + Functions:
    - login(): Authenticates a user and returns a JWT token.
    - register(): Registers a new user.
    - dashboard(current\_user): Retrieves the user's dashboard information.
* **doctor\_routes.py**:
  + Routes:
    - /api/doctor/patients (GET): Retrieves a list of patients for the authenticated doctor.
    - /api/doctor/patient-constants/<patient\_id> (GET): Retrieves a patient's constants (insulin-to-carb ratio, correction factor, target glucose, activity coefficients).
    - /api/doctor/update-patient-constants (POST): Updates a patient's constants.
  + Functions:
    - get\_doctor\_patients(current\_user): Retrieves a list of patients for the authenticated doctor.
    - get\_patient\_constants(current\_user, patient\_id): Retrieves a patient's constants.
    - update\_patient\_constants(current\_user): Updates a patient's constants.
* **food\_routes.py**:
  + Routes:
    - /api/food/search (GET): Searches for foods based on a query and optional category.
    - /api/food/custom (POST): Adds a custom food item.
    - /api/food/categories (GET): Retrieves the available food categories and standard portion information.
    - /api/food/nutritional-summary (POST): Provides a nutritional summary for a given meal.
    - /api/food/custom (GET): Retrieves the user's custom food items.
    - /api/food/favorite (POST): Adds a food item to the user's favorites.
    - /api/food/favorite (GET): Retrieves the user's favorite food items.
    - /api/food/measurements (GET): Retrieves the supported measurement types and standard portions.
  + Functions:
    - search\_food\_api(current\_user): Searches for foods based on a query and optional category.
    - add\_custom\_food(current\_user): Adds a custom food item.
    - get\_categories(current\_user): Retrieves the available food categories and standard portion information.
    - get\_nutritional\_summary(current\_user): Provides a nutritional summary for a given meal.
    - get\_custom\_foods(current\_user): Retrieves the user's custom food items.
    - add\_favorite\_food(current\_user): Adds a food item to the user's favorites.
    - get\_favorite\_foods(current\_user): Retrieves the user's favorite food items.
    - get\_measurements(current\_user): Retrieves the supported measurement types and standard portions.
* **test\_routes.py**:
  + Routes:
    - /test (GET): A simple test route that returns a JSON response.
  + Functions:
    - test\_route(): Handles the /test route and returns a JSON response. add the same for this files

 **auth.py** (in the utils/ directory):

* Functions:
  + token\_required(f): A decorator that checks if a valid JWT token is provided in the request headers. It extracts the user information from the token and passes it to the decorated function.

 **error\_handler.py** (in the utils/ directory):

* Functions:
  + api\_error\_handler(f): A decorator that handles exceptions raised by the decorated function and returns a JSON response with a generic error message.  
      
      
    ############################

 **activity.py**:

* Constants:
  + ACTIVITY\_LEVELS: A dictionary of activity level coefficients.
* Functions:
  + calculate\_activity\_level(activity\_level, duration): Calculates the activity level coefficient based on the activity level and duration.
  + update\_blood\_sugar(blood\_sugar, activity\_level, duration): Updates the blood sugar level based on the activity level and duration.

 **blood\_sugar.py**:

* Functions:
  + calculate\_blood\_sugar(carbs, insulin, activity\_level, duration, current\_blood\_sugar): Calculates the expected blood sugar level based on the provided inputs.
  + update\_blood\_sugar(current\_blood\_sugar, predicted\_blood\_sugar): Updates the current blood sugar level based on the predicted value.

 **meal\_insulin.py**:

* Functions:
  + calculate\_insulin\_dose(carbs, insulin\_to\_carb\_ratio, correction\_factor, target\_blood\_sugar, current\_blood\_sugar): Calculates the insulin dose required for a meal based on the provided inputs.

 **activities.py**:

* Constants:
  + ACTIVITY\_CATEGORIES: A dictionary of activity categories and their descriptions.
  + ACTIVITY\_INTENSITY\_FACTORS: A dictionary of activity intensity factors.
* Functions:
  + get\_activity\_intensity\_factor(activity\_type): Retrieves the activity intensity factor for a given activity type.
  + calculate\_activity\_impact(activity\_type, duration, current\_blood\_sugar): Calculates the impact of an activity on the blood sugar level.

 **enhanced\_meal\_insulin.py**:

* Functions:
  + calculate\_enhanced\_insulin\_dose(carbs, insulin\_to\_carb\_ratio, correction\_factor, target\_blood\_sugar, current\_blood\_sugar, activity\_level, activity\_duration): Calculates the enhanced insulin dose required for a meal based on the provided inputs, including activity level and duration.