**Basic Todo App (CRUD) Feature Details**

Your Go Todo app will primarily focus on the core CRUD operations (Create, Read, Update, Delete) for Todo items.

**1. User Management (Optional but Highly Recommended for Auth)**

If you decide to implement authentication, you'll need basic user management. Even if you skip authentication initially, defining user relationships for todos is good practice.

* **User Model:**
  + ID (Primary Key, Auto-increment)
  + Username (Unique, String)
  + Email (Unique, String, Email format validation)
  + PasswordHash (String - **NEVER store plain passwords!** Use a hashing library like bcrypt).
  + CreatedAt (Timestamp)
  + UpdatedAt (Timestamp)
  + **Relationships:** A user can have many todos.
* **API Endpoints:**
  + POST /register: Create a new user.
    - Input: username, email, password
    - Response: Success message or error (e.g., username/email already exists).
  + POST /login: Authenticate a user.
    - Input: username (or email), password
    - Response: JWT token (if using token-based auth) or session ID.
  + GET /me (Protected): Get current user's profile.
    - Response: User details (excluding password hash).

**2. Todo Item Management**

This is the core of your application.

* **Todo Model:**
  + ID (Primary Key, Auto-increment)
  + UserID (Foreign Key referencing User.ID - if user management is implemented)
  + Title (String, Required, e.g., max 255 chars)
  + Description (Text, Optional)
  + Completed (Boolean, Default: false)
  + DueDate (DateTime, Optional)
  + CreatedAt (Timestamp)
  + UpdatedAt (Timestamp)
* **API Endpoints:**
  + **Create Todo:**
    - POST /todos
    - Input: title (required), description (optional), dueDate (optional)
    - Logic:
      * Validate input (e.g., title not empty).
      * Create a new Todo item in the database.
      * If authentication is enabled, assign the UserID of the authenticated user.
    - Response: Created Todo item (including its ID) and HTTP 201 Created status.
  + **Get All Todos:**
    - GET /todos
    - Logic:
      * Retrieve all Todo items from the database.
      * If authentication is enabled, retrieve only todos belonging to the authenticated user.
      * **Optional:** Add query parameters for filtering (e.g., ?completed=true, ?search=keyword), pagination (?page=1&limit=10), and sorting (?sort\_by=createdAt&order=desc).
    - Response: Array of Todo items and HTTP 200 OK status.
  + **Get Single Todo:**
    - GET /todos/:id
    - Logic:
      * Retrieve a specific Todo item by its ID.
      * If authentication is enabled, ensure the todo belongs to the authenticated user.
      * Handle case where Todo is not found (HTTP 404 Not Found).
    - Response: Single Todo item and HTTP 200 OK status.
  + **Update Todo:**
    - PUT /todos/:id or PATCH /todos/:id (Choose one, PATCH is generally preferred for partial updates)
    - Input: title, description, completed, dueDate (any combination)
    - Logic:
      * Validate input.
      * Find the Todo item by ID.
      * If authentication is enabled, ensure the todo belongs to the authenticated user.
      * Update the specified fields.
      * Handle case where Todo is not found (HTTP 404 Not Found).
    - Response: Updated Todo item and HTTP 200 OK status.
  + **Delete Todo:**
    - DELETE /todos/:id
    - Logic:
      * Find the Todo item by ID.
      * If authentication is enabled, ensure the todo belongs to the authenticated user.
      * Delete the Todo item from the database.
      * Handle case where Todo is not found (HTTP 404 Not Found).
    - Response: Empty response with HTTP 204 No Content or a success message with HTTP 200 OK.

**3. Error Handling and Validation**

Crucial for a robust API.

* **Input Validation:**
  + Validate request body fields (e.g., title is not empty, email format).
  + Return meaningful error messages and appropriate HTTP status codes (e.g., 400 Bad Request).
* **Database Errors:**
  + Handle cases where a record is not found (e.g., 404 Not Found).
  + Handle database connection errors.
* **Authentication Errors:** (If implementing authentication)
  + Invalid credentials (401 Unauthorized).
  + Missing/invalid token (401 Unauthorized).
  + Access forbidden (403 Forbidden - e.g., trying to access another user's todo).
* **Consistent Error Responses:** Define a standard JSON error response format (e.g., { "error": "Error message details" }).

**4. Database Migrations (GORM)**

Essential for managing changes to your database schema.

* **Migration Tool:** Use a dedicated migration tool or GORM's auto-migrate feature (though a dedicated tool like golang-migrate is recommended for production).
* **Initial Migration:** Create a migration to define the users (if applicable) and todos tables with their respective columns and constraints.
* **Subsequent Migrations:** Practice creating new migrations for any future schema changes (e.g., adding a new field to Todo).
* **Rollback Capability:** Ensure your migrations can be rolled back.

**5. Basic Authentication (Optional but Highly Recommended)**

This adds a significant layer of complexity and learning.

* **Mechanism:**
  + **JWT (JSON Web Tokens):** A popular choice for REST APIs.
    - User logs in, server issues a JWT.
    - Client stores the JWT and sends it with every subsequent request in the Authorization header (Bearer <token>).
    - Server validates the JWT on each protected route.
  + **Session-based:** Less common for pure REST APIs, but an alternative.
* **Features:**
  + **Password Hashing:** Use bcrypt or similar for secure password storage.
  + **Token Generation and Validation:** Create middleware to generate and validate JWTs.
  + **Protected Routes:** Implement middleware that requires a valid token for specific API endpoints (e.g., creating, updating, or deleting todos).
  + **User Context:** After authentication, make the UserID available to subsequent handlers (e.g., via Gin's c.Set).

**6. Swagger for API Documentation**

Crucial for self-documenting your API and making it easy for others (or your future self) to understand.

* **Tool:** swaggo/gin-swagger in conjunction with swaggo/swag.
* **Integration:**
  + Annotate your Go code (structs and handler functions) with Swagger comments (e.g., @Summary, @Param, @Success, @Failure).
  + Generate Swagger documentation (swagger init, swagger generate).
  + Serve the Swagger UI through a Gin route (e.g., GET /swagger/\*any).
* **Documentation Coverage:**
  + Document every endpoint: purpose, parameters, request body, successful responses, and error responses.
  + Document your models (data structures).
  + Include authentication requirements in the documentation.

**7. Project Structure**

A well-organized project is easier to maintain and scale.

* main.go: Entry point for your application.
* config/: Database connection details, environment variables.
* models/: GORM models (User, Todo).
* handlers/: Functions that handle HTTP requests (CRUD logic).
* routes/: Define your API routes and associate them with handlers.
* middlewares/: Authentication middleware, logging middleware.
* utils/: Helper functions (e.g., password hashing, JWT generation).
* migrations/: SQL migration files (if using a separate migration tool).
* docs/: Generated Swagger documentation.

**8. Environment Variables**

* **Database Connection String:** Keep sensitive information out of your code.
* **JWT Secret Key:** For signing and verifying JWTs.
* **Port Number:** For your application to listen on.

**9. Logging**

* Implement basic logging for incoming requests, errors, and significant events.
* Gin has built-in logging, but you can also integrate a more structured logger like logrus or zap.

**10. Graceful Shutdown**

* Handle SIGINT (Ctrl+C) and SIGTERM signals to gracefully shut down your server, allowing ongoing requests to complete before exiting.

By implementing all these features, you'll gain a comprehensive understanding of building a robust Go REST API with Gin, GORM, MySQL, and incorporating best practices for authentication and documentation. Good luck!