That is a fantastic project for an interview! It allows you to showcase core skills in Go, API design (REST), modularity, and database interaction. Since the goal is to demonstrate **fundamental skills**, leveraging the robust, modular architecture shown in the source material is the perfect approach.

How familiar are you currently with Go and the Gin Gonic framework, and specifically, what experience do you have with PostgreSQL database access in Go?

1. Architecture and Key Concepts for a Go REST Service

The approach outlined in the source video—building a loosely coupled, distributed system—provides a highly professional foundation for your subscription service.

A. Architectural Components

The recommended structure emphasizes **Separation of Concerns**:

Component	Purpose (Goal: DRY & Modularity)	Relevance to Your Task	Source Citation
main.go	Application entry point (func main) and primary configuration (e.g., database connection setup, logging initialization, calling route setup functions).	Handles service startup and calls the database.Connect function (or equivalent for PostgreSQL).	
models	Defines Go **structs** that represent data resources (like your Subscription object) and includes serialization/validation tags.	Your Subscription and AggregationQuery request bodies will be defined here.	
controllers	Contains the **Handler Functions** (the logic for HTTP endpoints). These functions receive requests via the Gin context (C) and prepare responses.	Implements the CRUDL and aggregation business logic (e.g., calling database functions).	
routes	Separates endpoint configuration from handler logic, defining paths and mapping them to controllers.	You will define paths like /subscriptions (POST, GET List) and /subscriptions/:id (GET One, PATCH, DELETE)	

		here.
database	Centralizes code for interacting with the database (e.g., establishing the connection, handling the MongoDB Go Driver in the source, or a PostgreSQL library in your case).	Manages your PostgreSQL connection and houses reusable functions for transactions/queries.
utils	Stores reusable helper or utility functions (e.g., reading environment variables, custom validators).	Essential for reading your configuration files (.env/YAML) and logging setup.

B. Key Go and Gin Gonic Concepts

1. **Gin Gonic:** A high-performance HTTP web framework chosen for its speed and minimalistic power. It allows easy definition of routes, middleware, and request/response handling.
2. **Packages:** Go uses packages to group related code. The entry point is always `package main`. To be accessible outside its package, a function or variable must be **exported** (start with a Capital Letter).
3. **Variable Declaration (`:=`):** Go uses the short variable declaration operator (`:=`) inside functions to declare and initialize a variable simultaneously, with the compiler automatically inferring the type.
4. **Context Management:** Essential for robust API services. Using `context.WithTimeout` and `defer cancel` ensures that database operations have deadlines (e.g., 100 seconds) and frees up resources, preventing leaks. **Best Practice:** Pass the Gin context (`C`) to your resource cleanup calls rather than `context.Background`, as `C` respects client request cancellations.
5. **Data Structuring:** Models are defined using `type [Name] struct`. You will use **struct tags** (e.g., `json:"service_name"`) for marshaling/unmarshaling JSON data.

2. Step-by-Step Plan for the Subscription Service

This plan adapts the modular approach of the source material to your specific PostgreSQL CRUDL/Aggregation task, focusing on demonstrating fundamental skills.

Phase 1: Project Setup and Configuration (Requirements 5, 7)

Step	Goal	Source/Notes
1. Initialize Project Structure	Create the standard folders: cmd/server, internal/controllers, internal/models, internal/database, internal/routes, and config.	The structure based on the source ensures modularity.
2. Go Module Init	Initialize the Go module inside your project root.	Use go mod init [your-project-name].
3. Install	Install Gin Gonic and a	Use go get

Dependencies	PostgreSQL driver (e.g., lib/pq or pgx).	github.com/gin-gonic/gin.
4. Define Configuration	Create a config.yaml or .env file to hold database credentials, server port, and logging levels.	Use a library (like viper for YAML or jo/go.env from the source) to read these values into a Go struct at startup.
5. Docker Setup	Create Dockerfile (for the Go service) and docker-compose.yml (to link the Go service, PostgreSQL container, and optionally the migration tool).	Essential for Requirement 7.

Phase 2: Data Model and Database Layer (Requirements 1, 3)

Step	Goal	Source/Notes
6. Define Models (Structs)	Create the Subscription struct in internal/models, mapping fields to PostgreSQL columns (e.g., service_name, price, user_id, start_date, end_date).	Use JSON tags for HTTP serialization (json:"service_name") and add validation tags (validate:"required,min=1").
7. Database Connection	Implement the PostgreSQL connection logic within internal/database.	This mirrors the DBInstance or connect function in the source. This function should return the connected database client object.
8. Migrations	Implement database migration files (SQL scripts or a Go migration tool) to create the necessary subscriptions table.	Required by the task. This table must enforce UUID format for user_id and handle integer prices [T: 3, 1].

Phase 3: CRUDL Endpoints (Requirement 1, 4)

Step	Goal	Source/Notes
9. Logging Setup	Configure logging (e.g., using Go's built-in log package or a structured logger like zap) to capture requests and errors.	Logging should be included in every controller function to track the execution path and any errors.
10. Controller	Create SubscriptionController in	Each function must accept

Setup	internal/controllers with placeholder functions: CreateSubscription, ListSubscriptions, GetSubscription, UpdateSubscription, DeleteSubscription.	the Gin context C.
11. Implement Create (C)	Bind the incoming JSON request body to the Subscription struct. Validate the data. Insert the new record into PostgreSQL.	Use C.ShouldBindJSON(). Return HTTP Status Created (201) upon success.
12. Implement Read/List (R, L)	Query PostgreSQL to retrieve a single subscription (by ID) or all subscriptions (List).	The List endpoint should allow optional filtering by user_id or other parameters (if provided in the task notes) [T: 1].
13. Implement Update/Delete (U, D)	Implement handlers for PATCH and DELETE operations, targeting the record via a URL parameter (e.g., /subscriptions/:id).	Ensure appropriate status codes are returned (e.g., 200 OK for success, 404 Not Found if the ID is missing).

Phase 4: Aggregation Endpoint (Requirement 2)

Step	Goal	Source/Notes
14. Define Aggregation Model	Create a request struct for the aggregation query (containing user_id, service_name filter, and start_period, end_period).	This will dictate how the client requests the aggregated cost.
15. Implement Aggregation Handler	Write the controller function to receive the query parameters or body, construct a complex SQL query (using SUM() and WHERE clauses for filtering by date/user/service), and return the total cost.	This is the core business logic. Use Gin's C.Query() for URL parameters or C.ShouldBindJSON() for a body.

Phase 5: Finalization (Requirement 6)

Step	Goal	Source/Notes
16. Swagger Documentation	Generate and/or manually provide documentation for all implemented API endpoints (CRUDL and Aggregation).	While the source did not cover Swagger, using annotations and a tool like swaggo is standard practice

		for Go APIs.
17. Final Code Review	Ensure code adheres to Go idioms (naming conventions, error handling) and every significant part of the code is covered by clear logs.	Double-check that := is used correctly and that all functions use the standard Go error handling pattern (if err != nil).

This structured plan, built on the principles of modularity and clear request handling demonstrated in the source, will allow you to build a service that perfectly demonstrates your fundamental skills to an interviewer.

What part of this plan would you like to explore first? We could start with designing the `Subscription` data model and its validation tags, which is a great place to demonstrate clean coding practices.