

Technical Specification

REST Service for User Subscription Management

1. General Overview

The task is to design and implement a REST service for managing and aggregating data related to users' online service subscriptions.

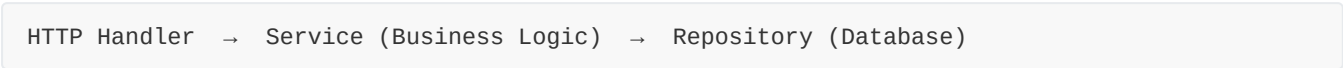
The service provides an API for:

- managing subscriptions (create, read, update, delete, list),
- calculating the total cost of subscriptions over a specified period with filtering options.

The service is **stateless**, does not manage users, and operates exclusively on subscription data.

2. Architecture and Design Principles

The service must be implemented in accordance with **Clean Architecture** principles and use a **three-layer architecture**:



Core principles:

- Clear separation of responsibilities between layers
- Dependency Injection performed at the `main` level
- Use of interfaces for `service` and `repository` layers
- No business logic in the HTTP layer
- Propagation of `context.Context` through all application layers

3. Subscription Data Model

Each subscription record contains the following fields:

Field	Description
<code>id</code>	Unique subscription identifier (UUID)
<code>service_name</code>	Name of the subscription service
<code>price</code>	Monthly subscription cost (integer, rubles only)
<code>user_id</code>	User identifier (UUID)
<code>start_date</code>	Subscription start date (<code>MM-YYYY</code> format)

Field	Description
end_date	Subscription end date (optional, MM-YYYY format)

Constraints:

- Subscription cost is an **integer number of rubles** (no fractional values)
- User existence validation is **not required**
- User management is outside the scope of this service

4. HTTP API (CRUDL + Aggregation)

4.1 CRUDL Operations

Method	Endpoint	Description	Status Codes
POST	/subscriptions	Create a subscription	201 Created, 400
GET	/subscriptions/{id}	Get subscription by ID	200 OK, 404 Not Found
PUT	/subscriptions/{id}	Update subscription	200 OK, 400
DELETE	/subscriptions/{id}	Delete subscription	204 No Content
GET	/subscriptions	List subscriptions	200 OK

Supported features:

- Pagination (limit , offset)
- Filtering by user_id and service_name

4.2 Subscription Aggregation

Endpoint:

```
GET /subscriptions/summary
```

Purpose:

Calculate the total cost of subscriptions over a specified time period.

Query parameters:

- from — period start (MM-YYYY)
- to — period end (MM-YYYY)
- user_id — optional user filter
- service_name — optional service filter

Response:

```
{
  "total": 1200
}
```

5. Data Validation

- Validation is implemented using `go-playground/validator`
- Validated aspects include:
 - date format (`MM-YYYY`)
 - required fields
 - minimum string length
 - UUID correctness
- Validation errors return HTTP `400 Bad Request`

6. Database Layer

Database

- PostgreSQL

Migrations

- Managed using **Goose**
- **Two SQL migration files:**
 - `0001_init_subscriptions.sql` — schema initialization
 - `0002_add_indexes.sql` — index creation
- Supports `up` / `down` migrations
- Migrations are executed automatically on application startup

Indexes

Indexes are created for:

- `user_id`
 - `service_name`
 - date ranges
-

7. Logging

Logging is implemented for:

- HTTP requests (via `LoggingMiddleware`)
- All application layers:
 - `handler`
 - `service`
 - `repository`

Log levels:

- `INFO`
- `ERROR`
- `DEBUG`

8. Configuration

- Configuration management is handled via `Viper`
- Configuration sources:
 - `config.yml` — application configuration
 - `.env` — sensitive data (passwords)
- Environment variables override configuration file values
- Separate configurations for production and testing environments

9. Graceful Shutdown

- Graceful application shutdown is implemented
 - Handles `SIGINT` / `SIGTERM`
 - Shutdown timeout: **5 seconds**
 - Ensures proper termination of:
 - HTTP server
 - database connections
-

10. Testing

Test coverage includes:

- **Unit tests** for the service layer (with mocks)
 - **Integration tests** for:
 - repository layer
 - HTTP handlers
 - Uses a real PostgreSQL database via Docker
-

11. CI/CD

- Implemented using GitHub Actions
 - Pipeline includes:
 - build verification
 - test execution
 - basic project validation
-

12. Documentation

- Swagger (OpenAPI)
 - Generated using `swag`
 - Swagger UI available at `/swagger/index.html`
 - All HTTP endpoints are fully annotated
 - **Godoc comments** are provided for all public structures and methods
-

13. Example: Create Subscription Request

```
{
  "service_name": "Yandex Plus",
  "price": 400,
  "user_id": "60601fee-2bf1-4721-ae6f-7636e79a0cba",
  "start_date": "07-2025"
}
```

14. Technology Stack

- Go
- PostgreSQL
- pgx

- Goose
- chi
- Viper
- go-playground/validator
- Swagger (swag)
- Docker / Docker Compose
- GitHub Actions