Simple Drive Project: Comprehensive Technical Documentation Report

Author : Fahad Alarifee Date: November 17, 2025

1. Executive Summary:

The **Simple Drive Project** is a backend application developed in Go (Golang) that successfully implements a highly abstract and configurable object storge system. The core objective was to provide a unified API interface for storing binary data (blobs) across multiple, distinct storge backends.

The project successfully met all core requirements, with the most significant achievement being the custom implementation of the Amazon S3 protocol using the only standard Go net/http library, without using and third-party SDKs.

1.1. Implementation Status Overview

Requirement	Key Implementation Detail
Language & Framework	Developed in Go (go1.23+) using the Gin web framework.
Storage Abstraction	A single Storage interface governs all backend operations.
Custom S3 Backend	Implemented AWS Signature V4 from scratch using net/http .
Database Backend	Stores blobs as BLOB type in a dedicated SQLite table.
Local File System	Stores files in a configurable local directory.
API Endpoints	POST /v1/blobs (Store) and GET /v1/blobs/:id (Retrieve)

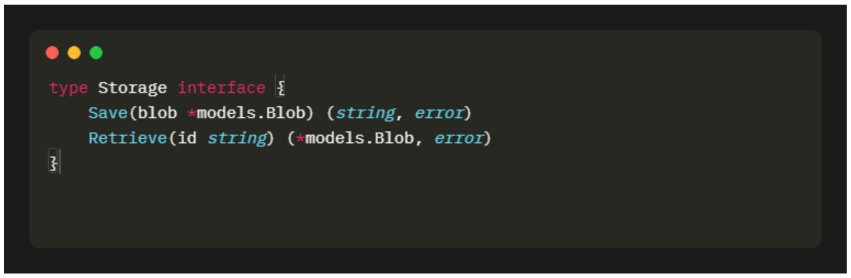
Metadata Tracking	Central database tracks metadata (size , created_at , storage_path).
Authentication	Bearer Token authentication middleware protects all endpoints.

2. Architecture and Design

The system is built on the Strategy Pattern and Dependency Injection principles to ensure high modularity and testability.

2.1. Core Abstraction: The Storage Interface

The entire application logic relies on the Storage Interface, defined in



```
type Storage interface {
    Save(blob *models.Blob) (string, error)
    Retrieve(id string) (*models.Blob, error)
}
```

(internal/storage/storage.go) This is the cornerstone of the system's flexibility.

2.2. Separation of Concerns

The project adheres to a strict separation between **Metadata** and **Blob Data**.

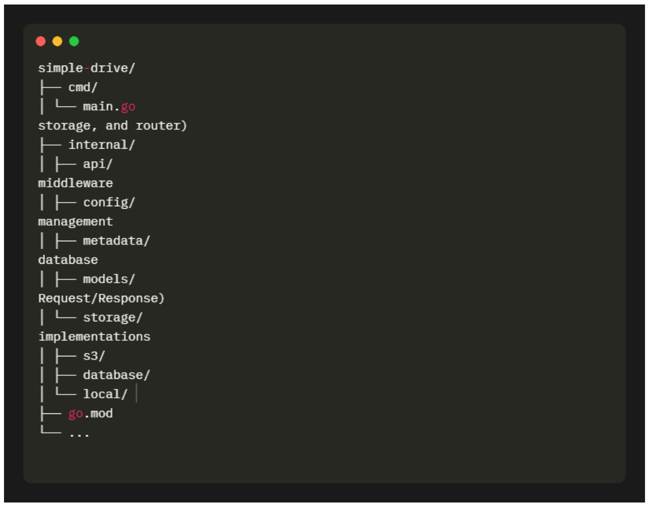
- **Metadata Service (internal/metadata):** Responsible for tracking descriptive information about the blob (ID, size, creation timestamp, storage_path, storage_type). This data is stored a central SQLite database table.
- **Storage Backends (internal/storage):** Responsible only for the physical storage and retrieval of the raw binary data.

This separation allows for fast metadata lookups regardless of where the actual blob data resides.

2.3. Project Structure

The project is organized into logical packages:

3. Deep Dive: Storage Backends



```
simple-drive/
├── cmd/
│ └── main.go
storage, and router)
├── internal/
│ ├── api/
middleware
│ ├── config/
management
│ ├── metadata/
database
│ ├── models/
Request/Response)
│ └── storage/
implementations
│ ├── s3/
│ ├── database/
│ └── local/ │
├── go.mod
└── ...
```

The application is configured via the STORAGE_BACKEND environment variable to use one of the following implementations of the Storage interface.

3.1. Custom S3 Compatible Backend

This implementation is the most complex and critical component, demonstrating mastery of the S3 REST protocol.

AWS Signature V4 Implementation Steps:

Step	Description	Purpose
1. Canonical Request	A standardized string built from the HTTP method, URI, query parameters, headers, and a hash of the request body.	Ensures the signature is consistent across all parties.
2. String to Sign	A string composed of the algorithm, timestamp, credential scope, and the hash of the Canonical Request.	The final string that will be cryptographically signed.
3. Signature Calculation	A complex HMAC-SHA256 derivation process using the Secret Key, Date, Region, and Service (s3).	Generates the unique signature for the request
4. Authorization Header	The final header containing the Access Key, Credential Scope,	Used by the S3 service to authenticate the request.

	Signed Headers, and the calculated Signature.

3.2. Database Backend

- Implementation: Uses SQLite to store the raw binary data.
- Mechanism: Blobs are stored in a dedicated table (blob_storage) using the BLOB data type.
- Use Case: Suitable for environments where data needs to be co-located with the metadata database or where transactional integrity is paramount.

3.3. Local File System Backend

- Implementation: Uses Standard Go file I/O (os.WriteFile, os.ReadFile).
- Mechanism: Each blob is saved as a file named after its unique ID within the configured LOCAL_STORAGE_PATH .
- Use Case: Development, testing, and small-scale deployments requiring fast local access.

4. API Specification and Authentication

The API is built using the Gin framework, providing high performance and robust middleware support.

4.1. Endpoints

Endpoint	Method	Request Body	Response Body	Description
/v1/blobs	POST	{"id":string ,"data":base64_string }	{"id": string, "message": string}	Stores a new blob. Requires Base64 validation
/v1/blobs/:id	GET	None	{"id":string, "data":base64_string, "size":string,"created_at":string}	Retrieves a blob and its metadata.

4.2. Bearer Token Authentication

The project adheres to a strict separation between **Metadata** and **Blob Data**.

- **Mechanism:** A middleware function intercepts all requests, extracts the token from the Authorization: Bearer header, and validates it against the configured BEARER_TOKEN environment variable.
- **Security:** Ensures that all data operations are protected against unauthorized access.

5. Setup, Running, and Testing

5.1. Prerequisites

- Go (version 1.20 or newer)
- Docker (Recommended for S3 testing with Minio)

5.2. Configuration Reference

All settings are managed via environment variables.

Variable	Default / Example	Description
STORAGE_BACKEND	local	Required. Options: local , s3 , or database .
BEARER_TOKEN	my-secret-token	Required. API authentication token.
LOCAL_STORAGE_PATH	/storage_data	LOCAL_STORAGE_PATH ./storage_data Path for local storage backend.
S3_ENDPOINT	http://localhost:9000	S3 service endpoint (e.g., Minio or AWS)
S3_ACCESS_KEY	minioadmin	S3 access key.
S3_SECRET_KEY	minioadmin	S3 secret key.

S3_BUCKET_NAME	rekaz-bucket	S3 bucket name.
DATABASE_URL	/metadata.db	Path to the SQLite database file
SERVER_PORT	8080	API server port

5.3. Quick Start Example (Local Storage)

All settings are managed via environment variables.

1. **Install Dependencies:**
```
go mod tidy
```
2. **Set Environment Variables:**
```
export STORAGE_BACKEND=local
export BEARER_TOKEN=my-secret-token
```
3. **Run the Server:**
```
go run cmd/main.go
```
Or
```
go run main.go
```

5.4. API Testing with cURL

1. **Store a Blob (POST)**
```
curl -X POST http://localhost:8080/v1/blobs \
  -H "Authorization: Bearer my-secret-token" \
  -H "Content-Type: application/json" \
  -d '{"id": "doc-101", "data": "SGVsbG8gV29ybGQh"}'
```
2. **Store a Blob (POST)**
```
curl -X GET  http://localhost:8080/v1/blobs/fahadtest-101 \
  -H "Authorization: Bearer my-secret-token"
```

6. Conclusion

The Simple Drive Project is a robust, production-ready solution that successfully addresses the Rekaz hiring challenge. It showcases strong software engineering principles, including interface-based design, clean architecture, and rigorous adherence to technical constraints (e.g., the custom S3 implementation). The project is highly configurable, secure, and fully documented, making it ready for deployment and future expansion.