Go + Flutter Course

Data & APIs

Timur Harin

Lecture 03: **Data & APIs**

Building robust HTTP servers and REST clients

Block 3: Data & APIs

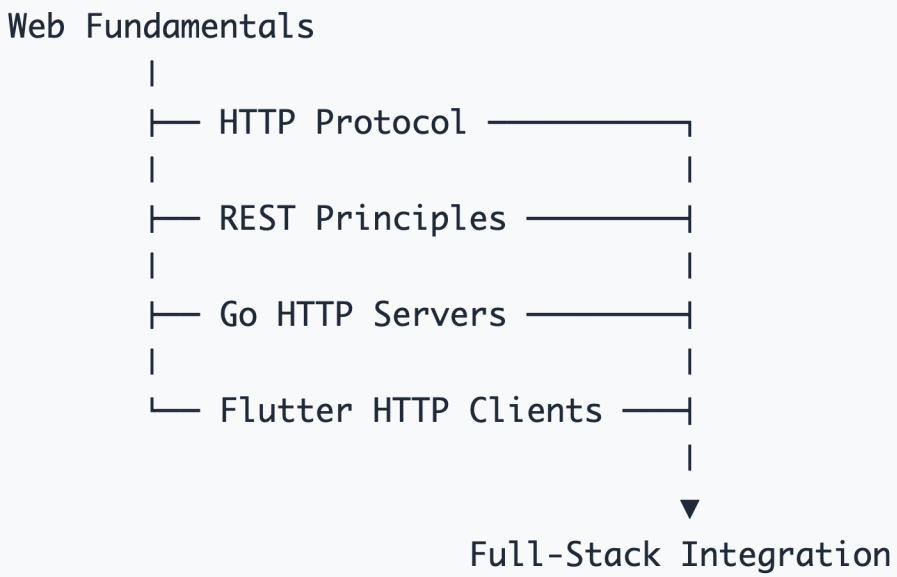
Lecture 03 Overview

- **HTTP Protocol**: Understanding the foundation
- **Go HTTP Servers**: Building robust REST APIs
- **Flutter HTTP Clients**: Consuming APIs effectively
- **Integration**: Full-stack communication patterns

What We'll Learn

- **Why APIs exist and how they evolved**
- **REST architectural principles and design**
- **HTTP protocol deep dive and best practices**
- **Go server development with routing and middleware**
- **Flutter HTTP client patterns and state management**
- **Data serialization and error handling**
- **Real-world integration patterns**

Learning Path



Progressive Learning Structure

- **Foundation First**: HTTP and REST principles
- **Server Development**: Go HTTP servers and middleware
- **Client Development**: Flutter HTTP consumption
- **Integration**: Real-world communication patterns

Part I: API & HTTP Fundamentals

What is an API?

> ***API (Application Programming Interface)*** *is a contract that defines how different software components should interact.*

Why APIs Exist

- **Separation of concerns**: Frontend and backend can evolve independently
- **Reusability**: One API serves multiple clients (mobile, web, desktop)
- **Scalability**: Distribute load across multiple services
- **Security**: Centralized data access control
- **Integration**: Connect different systems and services

Types of APIs

- **REST**: Representational State Transfer (most common)
- **GraphQL**: Query language for APIs
- **gRPC**: High-performance RPC framework

History of Web APIs

Evolution Timeline

- **1990s**: Static HTML pages, no dynamic data
- **Early 2000s**: SOAP (Simple Object Access Protocol) - heavyweight XML
- **2000s**: REST emerges - lightweight, HTTP-based
- **2010s**: JSON becomes dominant over XML
- **2010s+**: GraphQL, gRPC for specialized needs

SOAP (Legacy)

```xml
<soap:Envelope>
  <soap:Header>
    <auth:Authentication>
      <auth:Username>user</auth:Username>
      <auth:Password>pass</auth:Password>
    </auth:Authentication>
  </soap:Header>
  <soap:Body>
    <getUserRequest>
      <userId>123</userId>
    </getUserRequest>
  </soap:Body>
```

REST (Modern)

```http
GET /api/users/123 HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json

{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

HTTP Protocol Deep Dive

> ***HTTP (Hypertext Transfer Protocol)*** *is the foundation of data communication on the World Wide Web.*

HTTP Request Structure

```
METHOD /path/to/resource HTTP/1.1
Host: api.example.com
Authorization: Bearer token
Content-Type: application/json
Content-Length: 123

{
  "key": "value"
}
```

HTTP Response Structure

HTTP Methods & Semantics

Method	Purpose	Idempotent	Safe	Body
`GET`	Retrieve resource	✅	✅	No
`POST`	Create resource	❌	❌	Yes
`PUT`	Replace resource	✅	❌	Yes
`PATCH`	Partial update	❌	❌	Yes
`DELETE`	Remove resource	✅	❌	No
`HEAD`	Get headers only	✅	✅	No
`OPTIONS`	Get allowed methods	✅	✅	No

Key Concepts

- **Safe**: No side effects on server
- **Idempotent**: Multiple identical requests have same effect
- **Cacheable**: Response can be stored and reused

HTTP Status Codes

1xx - Informational

- `100 Continue`
- `101 Switching Protocols`

2xx - Success

- `200 OK` - Standard success
- `201 Created` - Resource created
- `202 Accepted` - Async processing
- `204 No Content` - Success, no body

3xx - Redirection

- `301 Moved Permanently`
- `302 Found` (temporary redirect)
- `304 Not Modified` (cached)

4xx - Client Error

- `400 Bad Request` - Invalid syntax
- `401 Unauthorized` - Authentication required
- `403 Forbidden` - Access denied
- `404 Not Found` - Resource doesn't exist
- `422 Unprocessable Entity` - Validation error
- `429 Too Many Requests` - Rate limit

5xx - Server Error

- `500 Internal Server Error`
- `502 Bad Gateway`
- `503 Service Unavailable`
- `504 Gateway Timeout`

REST Architectural Principles

> ***REST (Representational State Transfer)*** *is an architectural style for designing networked applications.*

Core Principles

1. Client-Server Architecture

- **Separation of concerns**: UI and data storage are independent
- **Portability**: Client can run on different platforms
- **Scalability**: Components can be scaled independently

2. Stateless

- **No session state**: Each request contains all necessary information
- **Scalability**: Server doesn't need to maintain client context
- **Reliability**: No session data to lose

3. Cacheable

REST Principles Continued

4. Uniform Interface

- **Resource identification**: URLs identify resources
- **Resource manipulation**: Standard HTTP methods
- **Self-descriptive messages**: Each message contains metadata
- **HATEOAS**: Hypermedia as the Engine of Application State

5. Layered System

- **Scalability**: Intermediary layers (load balancers, caches)
- **Security**: Firewalls and proxy servers
- **Encapsulation**: Client doesn't know internal architecture

6. Code on Demand (Optional)

- **Flexibility**: Server can send executable code to client
- **Examples**: JavaScript, Java applets

RESTful URL Design

Good RESTful URLs

```
GET    /api/users              # List all users
GET    /api/users/123          # Get specific user
POST   /api/users              # Create new user
PUT    /api/users/123          # Replace user
PATCH  /api/users/123          # Update user
DELETE /api/users/123          # Delete user

GET    /api/users/123/posts    # User's posts
POST   /api/users/123/posts    # Create post for user
GET    /api/posts/456/comments # Post's comments
```

Poor URL Design

```
GET    /api/getAllUsers        # Verb in URL
GET    /api/user?id=123        # Should use path param
POST   /api/deleteUser         # Wrong method
GET    /api/users/123/delete   # Action in URL
POST   /api/createPost         # Redundant verb
```

Best Practices

- **Use nouns**, not verbs
- **Plural resource names** for collections
- **Nested resources** for relationships
- **Query parameters** for filtering
- **Consistent naming** conventions

Data Serialization Formats

JSON (JavaScript Object Notation)

```json
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com",
  "active": true,
  "roles": ["user", "admin"],
  "profile": {
    "age": 30,
    "city": "Boston"
  },
  "created_at": "2025-07-02T10:00:00Z"
}
```

Pros:

- Human-readable
- Lightweight
- Native JavaScript support
- Wide language support

XML (Extensible Markup Language)

```xml
<user>
  <id>123</id>
  <name>John Doe</name>
  <email>john@example.com</email>
  <active>true</active>
  <roles>
    <role>user</role>
    <role>admin</role>
  </roles>
  <profile>
    <age>30</age>
    <city>Boston</city>
  </profile>
  <created_at>2025-07-02T10:00:00Z</created_at>
</user>
```

Pros:

- Schema validation
- Namespace support
- Mature ecosystem

Part II: Go HTTP Server Development

Go's HTTP Package

> *Go's `net/http` package provides a powerful, production-ready HTTP server with excellent performance characteristics.*

Key Features

- **Built-in HTTP server**: No external dependencies needed
- **Multiplexer**: Route requests to handlers
- **Middleware support**: Chain request processing
- **Context integration**: Request cancellation and timeouts
- **TLS support**: HTTPS out of the box
- **High performance**: Handles thousands of concurrent connections

Why Go for HTTP APIs?

- **Fast compilation**: Quick development cycle
- **Low memory footprint**: Efficient resource usage

Basic HTTP Server Setup

Minimal HTTP Server

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, World!")
}

func main() {
    http.HandleFunc("/hello", helloHandler)

    log.Println("Server starting on :8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

With Custom Server Configuration

```go
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/hello", helloHandler)

    server := &http.Server{
        Addr:         ":8080",
        Handler:      mux,
        ReadTimeout:  15 * time.Second,
        WriteTimeout: 15 * time.Second,
        IdleTimeout:  60 * time.Second,
    }

    log.Println("Server starting on :8080")
    log.Fatal(server.ListenAndServe())
}
```

HTTP Handlers Deep Dive

Handler Function Signature

```
type HandlerFunc func(http.ResponseWriter, *http.Request)

func userHandler(w http.ResponseWriter, r *http.Request) {
    // w: Write response back to client
    // r: Read request from client

    // Set response headers
    w.Header().Set("Content-Type", "application/json")

    // Write response body
    w.Write([]byte(`{"message": "Hello"}`))
}
```

Handler Interface

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type userController struct {
    db *Database
}

func (uc *userController) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case http.MethodGet:
        uc.getUsers(w, r)
    case http.MethodPost:
        uc.createUser(w, r)
    default:
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
    }
}
```

JSON Handling in Go

Data Models with JSON Tags

```
type User struct {
    ID       int       `json:"id"`
    Name     string    `json:"name"`
    Email    string    `json:"email"`
    Password string    `json:"-"`          // Never include in JSON
    Active   bool      `json:"active"`
    Created  time.Time `json:"created_at"`
}

type CreateUserRequest struct {
    Name     string `json:"name" validate:"required"`
    Email    string `json:"email" validate:"required,email"`
    Password string `json:"password" validate:"required,min=8"`
}

type APIResponse struct {
    Success bool        `json:"success"`
    Data    interface{} `json:"data,omitempty"`
    Error   string      `json:"error,omitempty"`
}
```

JSON Encoding/Decoding

```
func createUserHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodPost {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    var req CreateUserRequest

    // Decode JSON request body
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&req); err != nil {
        http.Error(w, "Invalid JSON", http.StatusBadRequest)
        return
    }
    defer r.Body.Close()

    // Create user (simulate)
    user := User{
        ID:      123,
        Name:    req.Name,
        Email:   req.Email,
        Active:  true,
        Created: time.Now(),
    }
```

JSON Handling Continued

```
    // Encode JSON response
    response := APIResponse{
        Success: true,
        Data:    user,
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)

    encoder := json.NewEncoder(w)
    if err := encoder.Encode(response); err != nil {
        log.Printf("Error encoding response: %v", err)
    }
}
```

JSON Helper Functions

```
func writeJSON(w http.ResponseWriter, status int, data interface{}) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)

    if err := json.NewEncoder(w).Encode(data); err != nil {
        log.Printf("Error encoding JSON: %v", err)
        http.Error(w, "Internal server error", http.StatusInternalServerError)
    }
}

func readJSON(r *http.Request, dst interface{}) error {
    decoder := json.NewDecoder(r.Body)
    return decoder.Decode(dst)
}
```

URL Routing Patterns

Manual Path Parsing

```
func userHandler(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Path

    // Extract user ID from path like /users/123
    parts := strings.Split(path, "/")
    if len(parts) < 3 {
        http.Error(w, "Invalid path", http.StatusBadRequest)
        return
    }

    userID := parts[2]

    switch r.Method {
    case http.MethodGet:
        getUserByID(w, r, userID)
    case http.MethodPut:
        updateUser(w, r, userID)
    case http.MethodDelete:
        deleteUser(w, r, userID)
    default:
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
    }
}
```

Using Gorilla Mux Router

```
import "github.com/gorilla/mux"

func setupRoutes() *mux.Router {
    r := mux.NewRouter()

    // API v1 routes
    api := r.PathPrefix("/api/v1").Subrouter()

    // User routes
    api.HandleFunc("/users", getUsersHandler).Methods("GET")
    api.HandleFunc("/users", createUserHandler).Methods("POST")
    api.HandleFunc("/users/{id:[0-9]+}", getUserHandler).Methods("GET")
    api.HandleFunc("/users/{id:[0-9]+}", updateUserHandler).Methods("PUT")
    api.HandleFunc("/users/{id:[0-9]+}", deleteUserHandler).Methods("DELETE")

    // Post routes
    api.HandleFunc("/users/{userId:[0-9]+}/posts", getPostsHandler).Methods("GET")
    api.HandleFunc("/posts/{id:[0-9]+}", getPostHandler).Methods("GET")

    return r
}
```

Advanced Routing with Gorilla Mux

Path Variables & Validation

```
func getUserHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    userID := vars["id"]

    // userID is guaranteed to be numeric due to regex pattern
    id, _ := strconv.Atoi(userID)

    user, err := getUserFromDB(id)
    if err != nil {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }

    writeJSON(w, http.StatusOK, user)
}
```

Query Parameters & Filtering

```
func getUsersHandler(w http.ResponseWriter, r *http.Request) {
    query := r.URL.Query()

    // Parse query parameters
    page := getIntParam(query, "page", 1)
    limit := getIntParam(query, "limit", 10)
    active := getBoolParam(query, "active", true)
    search := query.Get("search")

    filters := UserFilters{
        Page:   page,
        Limit:  limit,
        Active: &active,
        Search: search,
    }

    users, total, err := getUsersWithFilters(filters)
    if err != nil {
        http.Error(w, "Error fetching users", http.StatusInternalServerError)
        return
    }

    response := PaginatedResponse{
        Data:  users,
        Total: total,
        Page:  page,
        Limit: limit,
    }
```

Middleware Patterns

> ***Middleware*** *is code that runs before and after your main handler, allowing you to add cross-cutting concerns.*

Basic Middleware Structure

```
type Middleware func(http.Handler) http.Handler

func loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()

        // Call the next handler
        next.ServeHTTP(w, r)

        // Log after request completes
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}

// Usage
http.Handle("/api/", loggingMiddleware(http.HandlerFunc(apiHandler)))
```

CORS Middleware

```
func corsMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type, Authorization")

        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next.ServeHTTP(w, r)
    })
}
```

Authentication Middleware

JWT Authentication Middleware

```
func authMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        authHeader := r.Header.Get("Authorization")
        if authHeader == "" {
            http.Error(w, "Authorization header required", http.StatusUnauthorized)
            return
        }

        // Extract Bearer token
        parts := strings.Split(authHeader, " ")
        if len(parts) != 2 || parts[0] != "Bearer" {
            http.Error(w, "Invalid authorization header", http.StatusUnauthorized)
            return
        }

        token := parts[1]
        claims, err := validateJWT(token)
        if err != nil {
            http.Error(w, "Invalid token", http.StatusUnauthorized)
            return
        }
        // Add user info to request context
        ctx := context.WithValue(r.Context(), "userID", claims.UserID)
        ctx = context.WithValue(ctx, "userRole", claims.Role)

        next.ServeHTTP(w, r.WithContext(ctx))
    })
}

// Helper to get user from context
func getCurrentUser(r *http.Request) (int, error) {
    userID, ok := r.Context().Value("userID").(int)
    if !ok {
        return 0, errors.New("user not found in context")
    }
    return userID, nil
}
```

Middleware Chaining

Manual Chaining

```
func setupServer() {
    mux := http.NewServeMux()
    mux.HandleFunc("/api/users", usersHandler)

    // Chain middleware manually
    handler := loggingMiddleware(
        corsMiddleware(
            authMiddleware(mux),
        ),
    )

    server := &http.Server{
        Addr:    ":8080",
        Handler: handler,
    }

    server.ListenAndServe()
}
```

Middleware Chain Helper

```
func chain(middlewares ...Middleware) Middleware {
    return func(final http.Handler) http.Handler {
        for i := len(middlewares) - 1; i >= 0; i-- {
            final = middlewares[i](final)
        }
        return final
    }
}

// Usage
func setupServerWithChain() {
    mux := http.NewServeMux()
    mux.HandleFunc("/api/users", usersHandler)

    // Clean chaining
    handler := chain(
        loggingMiddleware,
        corsMiddleware,
        authMiddleware,
    )(mux)

    server := &http.Server{
        Addr:    ":8080",
        Handler: handler,
    }
```

Error Handling Strategies

Custom Error Types

```go
type APIError struct {
    Code    int    `json:"code"`
    Message string `json:"message"`
    Details string `json:"details,omitempty"`
}

func (e APIError) Error() string {
    return e.Message
}

type ErrorResponse struct {
    Error APIError `json:"error"`
}

// Predefined errors
var (
    ErrUserNotFound = APIError{
        Code:    404,
        Message: "User not found",
    }

    ErrInvalidInput = APIError{
        Code:    400,
        Message: "Invalid input data",
    }

    ErrUnauthorized = APIError{
        Code:    401,
```

Error Handling Middleware

```go
func errorHandlingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
                log.Printf("Panic recovered: %v", err)

                response := ErrorResponse{
                    Error: APIError{
                        Code:    500,
                        Message: "Internal server error",
                    },
                }

                w.Header().Set("Content-Type", "application/json")
                w.WriteHeader(http.StatusInternalServerError)
                json.NewEncoder(w).Encode(response)
            }
        }()

        next.ServeHTTP(w, r)
    })
}
```

Error Response Helpers

```go
func writeError(w http.ResponseWriter, apiErr APIError) {
    response := ErrorResponse{Error: apiErr}

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(apiErr.Code)

    if err := json.NewEncoder(w).Encode(response); err != nil {
        log.Printf("Error encoding error response: %v", err)
    }
}

func writeErrorf(w http.ResponseWriter, code int, format string, args ...interface{}) {
    apiErr := APIError{
        Code:    code,
        Message: fmt.Sprintf(format, args...),
    }
    writeError(w, apiErr)
}
```

```go
// Usage in handlers
func getUserHandler(w http.ResponseWriter, r *http.Request) {
    userID := getUserIDFromPath(r.URL.Path)
    if userID == 0 {
        writeError(w, ErrInvalidInput)
        return
    }

    user, err := userService.GetUser(userID)
    if err != nil {
        if errors.Is(err, sql.ErrNoRows) {
            writeError(w, ErrUserNotFound)
            return
        }

        log.Printf("Database error: %v", err)
        writeErrorf(w, 500, "Database error occurred")
        return
    }

    writeJSON(w, http.StatusOK, user)
}
```

Testing HTTP Endpoints

Basic HTTP Testing

```go
func TestGetUserHandler(t *testing.T) {
    // Create request
    req, err := http.NewRequest("GET", "/api/users/123", nil)
    if err != nil {
        t.Fatal(err)
    }

    // Create response recorder
    rr := httptest.NewRecorder()

    // Create handler
    handler := http.HandlerFunc(getUserHandler)

    // Execute request
    handler.ServeHTTP(rr, req)

    // Check status code
    if status := rr.Code; status != http.StatusOK {
        t.Errorf("Expected status %v, got %v", http.StatusOK, status)
    }

    // Check response body
    expected := `{"id":123,"name":"John Doe"}`
    if strings.TrimSpace(rr.Body.String()) != expected {
        t.Errorf("Expected body %v, got %v", expected, rr.Body.String())
    }
}
```

Testing with Mux Router

```go
func TestUserRoutes(t *testing.T) {
    router := setupRoutes()

    tests := []struct {
        name           string
        method         string
        url            string
        expectedStatus int
    }{
        {"Get all users", "GET", "/api/v1/users", 200},
        {"Get specific user", "GET", "/api/v1/users/123", 200},
        {"User not found", "GET", "/api/v1/users/999", 404},
        {"Invalid user ID", "GET", "/api/v1/users/abc", 400},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            req, _ := http.NewRequest(tt.method, tt.url, nil)
            rr := httptest.NewRecorder()

            router.ServeHTTP(rr, req)

            if rr.Code != tt.expectedStatus {
                t.Errorf("Expected %d, got %d", tt.expectedStatus, rr.Code)
            }
        })
    }
}
```

Testing with JSON Payloads

Testing POST Requests

```go
func TestCreateUserHandler(t *testing.T) {
    user := CreateUserRequest{
        Name:     "Jane Doe",
        Email:    "jane@example.com",
        Password: "securepassword",
    }

    jsonData, _ := json.Marshal(user)

    req, err := http.NewRequest("POST", "/api/users", bytes.NewBuffer(jsonData))
    if err != nil {
        t.Fatal(err)
    }
    req.Header.Set("Content-Type", "application/json")

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(createUserHandler)

    handler.ServeHTTP(rr, req)

    if status := rr.Code; status != http.StatusCreated {
        t.Errorf("Expected %v, got %v", http.StatusCreated, status)
    }

    // Parse response
    var response APIResponse
    if err := json.NewDecoder(rr.Body).Decode(&response); err != nil {
        t.Fatal("Could not decode response")
    }

    if !response.Success {
```

Testing Error Cases

```go
func TestCreateUserValidation(t *testing.T) {
    tests := []struct {
        name           string
        payload        CreateUserRequest
        expectedStatus int
        expectedError  string
    }{
        {
            name:           "Missing name",
            payload:        CreateUserRequest{Email: "test@test.com", Password: "password"},
            expectedStatus: 400,
            expectedError:  "Name is required",
        },
        {
            name:           "Invalid email",
            payload:        CreateUserRequest{Name: "Test", Email: "invalid", Password: "password"},
            expectedStatus: 400,
            expectedError:  "Invalid email format",
        },
        {
            name:           "Weak password",
            payload:        CreateUserRequest{Name: "Test", Email: "test@test.com", Password: "123"},
            expectedStatus: 400,
            expectedError:  "Password too weak",
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            jsonData, _ := json.Marshal(tt.payload)
            req, _ := http.NewRequest("POST", "/api/users", bytes.NewBuffer(jsonData))
            req.Header.Set("Content-Type", "application/json")

            rr := httptest.NewRecorder()
            handler := http.HandlerFunc(createUserHandler)
            handler.ServeHTTP(rr, req)

            if rr.Code != tt.expectedStatus {
                t.Errorf("Expected %d, got %d", tt.expectedStatus, rr.Code)
```

Part III: Flutter HTTP Client Development

Dart's HTTP Package

> `package:http` *is the standard HTTP client library for Dart and Flutter applications.*

Key Features

- **Simple API**: Easy-to-use methods for common HTTP operations
- **Async/await support**: Natural integration with Dart's async model
- **Request customization**: Headers, timeouts, body content
- **Response handling**: Status codes, headers, body parsing
- **Error handling**: Network errors, timeouts, HTTP errors

Installation

```
dependencies:
```

Basic HTTP Requests in Flutter

GET Request

```dart
Future<User> fetchUser(int userId) async {
  final response = await http.get(
    Uri.parse('https://api.example.com/users/$userId'),
    headers: {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
    },
  );

  if (response.statusCode == 200) {
    final Map<String, dynamic> json = jsonDecode(response.body);
    return User.fromJson(json);
  } else if (response.statusCode == 404) {
    throw UserNotFoundException('User not found');
  } else {
    throw ApiException('Failed to load user: ${response.statusCode}');
  }
}
```

POST Request

```dart
Future<User> createUser(CreateUserRequest request) async {
  final response = await http.post(
    Uri.parse('https://api.example.com/users'),
    headers: {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
    },
    body: jsonEncode(request.toJson()),
  );

  if (response.statusCode == 201) {
    final Map<String, dynamic> json = jsonDecode(response.body);
    return User.fromJson(json['data']);
  } else if (response.statusCode == 400) {
    final Map<String, dynamic> error = jsonDecode(response.body);
    throw ValidationException(error['error']['message']);
  } else {
    throw ApiException('Failed to create user: ${response.statusCode}');
  }
}
```

HTTP Request Methods

PUT Request

```dart
Future<User> updateUser(int userId, UpdateUserRequest request) async {
  final response = await http.put(
    Uri.parse('https://api.example.com/users/$userId'),
    headers: {
      'Content-Type': 'application/json',
      'Authorization': 'Bearer $token',
    },
    body: jsonEncode(request.toJson()),
  );

  if (response.statusCode == 200) {
    return User.fromJson(jsonDecode(response.body));
  } else {
    throw ApiException('Failed to update user');
  }
}
```

DELETE Request

```dart
Future<void> deleteUser(int userId) async {
  final response = await http.delete(
    Uri.parse('https://api.example.com/users/$userId'),
    headers: {
      'Authorization': 'Bearer $token',
    },
  );

  if (response.statusCode == 204) {
    // Success - no content
    return;
  } else if (response.statusCode == 404) {
    throw UserNotFoundException('User not found');
  } else {
    throw ApiException('Failed to delete user');
  }
}
```

Data Models & Serialization

User Model

```dart
class User {
  final int id;
  final String name;
  final String email;
  final bool active;
  final DateTime createdAt;

  User({
    required this.id,
    required this.name,
    required this.email,
    required this.active,
    required this.createdAt,
  });

  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      id: json['id'] as int,
      name: json['name'] as String,
      email: json['email'] as String,
      active: json['active'] as bool,
      createdAt: DateTime.parse(json['created_at'] as String),
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'name': name,
      'email': email,
      'active': active,
      'created_at': createdAt.toIso8601String(),
```

Request/Response Models

```dart
class CreateUserRequest {
  final String name;
  final String email;
  final String password;

  CreateUserRequest({
    required this.name,
    required this.email,
    required this.password,
  });

  Map<String, dynamic> toJson() {
    return {
      'name': name,
      'email': email,
      'password': password,
    };
  }
}

class ApiResponse<T> {
  final bool success;
  final T? data;
  final String? error;

  ApiResponse({
    required this.success,
    this.data,
    this.error,
  });

  factory ApiResponse.fromJson(
    Map<String, dynamic> json,
    T Function(Map<String, dynamic>) fromJsonT,
  ) {
    return ApiResponse<T>(
      success: json['success'] as bool,
      data: json['data'] != null ? fromJsonT(json['data']) : null,
```

API Service Layer Pattern

Centralized API Service

```dart
class ApiService {
  static const String baseUrl = 'https://api.example.com';
  static const Duration timeoutDuration = Duration(seconds: 30);

  final http.Client _client;
  String? _authToken;

  ApiService() : _client = http.Client();

  void setAuthToken(String token) {
    _authToken = token;
  }

  Map<String, String> get _headers {
    final headers = {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
    };

    if (_authToken != null) {
      headers['Authorization'] = 'Bearer $_authToken';
    }

    return headers;
  }

  Future<T> _handleResponse<T>(
    http.Response response,
    T Function(Map<String, dynamic>) fromJson,
  ) async {
    if (response.statusCode >= 200 && response.statusCode < 300) {
      final Map<String, dynamic> data = jsonDecode(response.body);
      return fromJson(data);
    } else if (response.statusCode == 401) {
      throw UnauthorizedException('Authentication required');
    } else if (response.statusCode == 404) {
      throw NotFoundException('Resource not found');
    } else if (response.statusCode >= 400 && response.statusCode < 500) {
      final Map<String, dynamic> error = jsonDecode(response.body);
      throw ApiException(error['message'] ?? 'Client error occurred');
    } else {
      throw ServerException('Server error occurred');
    }
  }

  void dispose() {
    _client.close();
  }
}
```

API Service Methods

```dart
extension UserService on ApiService {
  Future<List<User>> getUsers({
    int page = 1,
    int limit = 10,
    String? search,
  }) async {
    final uri = Uri.parse('$baseUrl/api/users').replace(
      queryParameters: {
        'page': page.toString(),
        'limit': limit.toString(),
        if (search != null && search.isNotEmpty) 'search': search,
      },
    );

    final response = await _client
        .get(uri, headers: _headers)
        .timeout(timeoutDuration);

    return _handleResponse(response, (data) {
      final List<dynamic> usersJson = data['data'];
      return usersJson.map((json) => User.fromJson(json)).toList();
    });
  }
```

```dart
  Future<User> getUser(int userId) async {
    final response = await _client
        .get(
          Uri.parse('$baseUrl/api/users/$userId'),
          headers: _headers,
        )
        .timeout(timeoutDuration);

    return _handleResponse(response, (data) => User.fromJson(data));
  }

  Future<User> createUser(CreateUserRequest request) async {
    final response = await _client
        .post(
          Uri.parse('$baseUrl/api/users'),
          headers: _headers,
          body: jsonEncode(request.toJson()),
        )
        .timeout(timeoutDuration);

    return _handleResponse(response, (data) => User.fromJson(data['data']));
  }
}
```

Custom Exception Handling

Exception Types

```
abstract class ApiException implements Exception {
  final String message;
  const ApiException(this.message);

  @override
  String toString() => 'ApiException: $message';
}

class NetworkException extends ApiException {
  const NetworkException(String message) : super(message);
}

class UnauthorizedException extends ApiException {
  const UnauthorizedException(String message) : super(message);
}

class NotFoundException extends ApiException {
  const NotFoundException(String message) : super(message);
}

class ValidationException extends ApiException {
  final Map<String, List<String>>? fieldErrors;

  const ValidationException(String message, {this.fieldErrors})
      : super(message);
}
```

Global Error Handler

```
class ApiErrorHandler {
  static void handleError(dynamic error) {
    if (error is SocketException) {
      throw NetworkException('No internet connection');
    } else if (error is TimeoutException) {
      throw NetworkException('Request timeout');
    } else if (error is FormatException) {
      throw ApiException('Invalid response format');
    } else if (error is ApiException) {
      rethrow;
    } else {
      throw ApiException('Unexpected error occurred');
    }
  }

  static String getErrorMessage(dynamic error) {
    if (error is NetworkException) {
      return 'Please check your internet connection';
    } else if (error is UnauthorizedException) {
      return 'Please log in again';
    } else if (error is NotFoundException) {
      return 'The requested item was not found';
    } else if (error is ValidationException) {
      return error.message;
    } else if (error is ServerException) {
      return 'Server is temporarily unavailable';
    } else {
      return 'An unexpected error occurred';
```

State Management with APIs

Provider Pattern

```
class UserProvider extends ChangeNotifier {
  final ApiService _apiService;

  List<User> _users = [];
  User? _selectedUser;
  bool _isLoading = false;
  String? _error;

  UserProvider(this._apiService);

  List<User> get users => _users;
  User? get selectedUser => _selectedUser;
  bool get isLoading => _isLoading;
  String? get error => _error;

  Future<void> loadUsers() async {
    _setLoading(true);
    _error = null;

    try {
      _users = await _apiService.getUsers();
      notifyListeners();
    } catch (e) {
      _error = ApiErrorHandler.getErrorMessage(e);
    } finally {
```

```
  Future<void> createUser(CreateUserRequest request) async {
    _setLoading(true);
    _error = null;

    try {
      final newUser = await _apiService.createUser(request);
      _users.add(newUser);
      notifyListeners();
    } catch (e) {
      _error = ApiErrorHandler.getErrorMessage(e);
      rethrow;
    } finally {
      _setLoading(false);
    }
  }

  void _setLoading(bool loading) {
    _isLoading = loading;
    notifyListeners();
  }

  void clearError() {
    _error = null;
    notifyListeners();
  }
}
```

Using APIs in Flutter Widgets

Consumer Widget

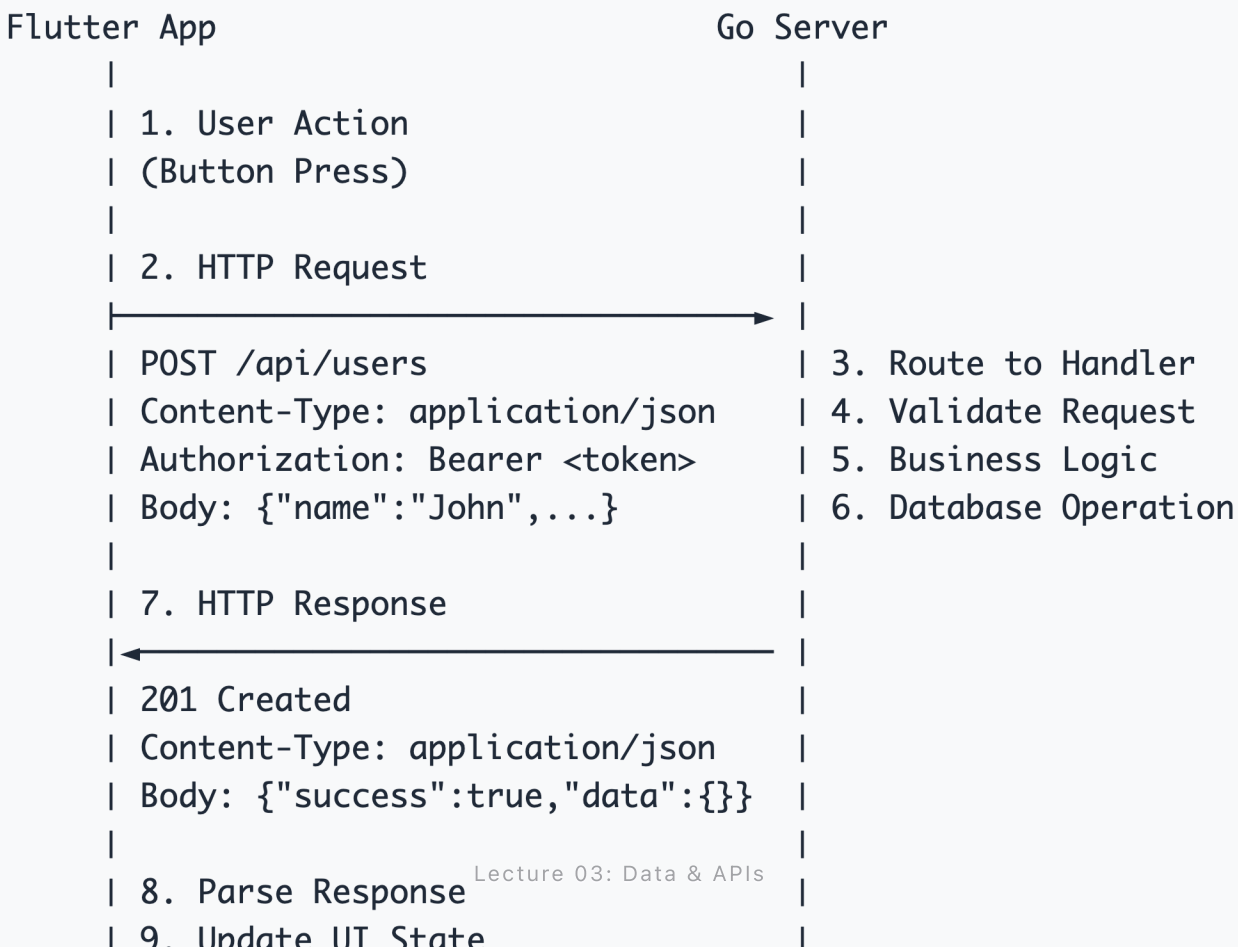
```dart
class UserListScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Users')),
      body: Consumer<UserProvider>(
        builder: (context, userProvider, child) {
          if (userProvider.isLoading) {
            return Center(child: CircularProgressIndicator());
          }

          if (userProvider.error != null) {
            return Center(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  Text(
                    userProvider.error!,
                    style: TextStyle(color: Colors.red),
                    textAlign: TextAlign.center,
                  ),
                  SizedBox(height: 16),
                  ElevatedButton(
                    onPressed: () {
                      userProvider.clearError();
                      userProvider.loadUsers();
                    },
                    child: Text('Retry'),
                  ),
                ],
              ),
            );
          }

          return RefreshIndicator(
            onRefresh: userProvider.loadUsers,
            child: ListView.builder(
              itemCount: userProvider.users.length,
              itemBuilder: (context, index) {
                final user = userProvider.users[index];
                return ListTile(
                  leading: CircleAvatar(
                    child: Text(user.name[0].toUpperCase()),
                  ),
                  title: Text(user.name),
                  subtitle: Text(user.email),
                  trailing: Icon(
                    user.active ? Icons.check_circle : Icons.cancel,
                    color: user.active ? Colors.green : Colors.red,
                  ),
                  onTap: () {
                    Navigator.push(
                      context,
                      MaterialPageRoute(
                        builder: (context) => UserDetailScreen(user: user),
                      ),
                    );
                  },
                );
              },
            ),
          );
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _showCreateUserDialog(context),
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Part IV: Integration & Best Practices

Full-Stack Communication Flow



Authentication Flow

Go JWT Middleware

```go
type Claims struct {
    UserID int    `json:"user_id"`
    Email  string `json:"email"`
    Role   string `json:"role"`
    jwt.StandardClaims
}

func generateJWT(userID int, email, role string) (string, error) {
    claims := Claims{
        UserID: userID,
        Email:  email,
        Role:   role,
        StandardClaims: jwt.StandardClaims{
            ExpiresAt: time.Now().Add(24 * time.Hour).Unix(),
            IssuedAt:  time.Now().Unix(),
        },
    }

    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return token.SignedString([]byte(jwtSecret))
}

func validateJWT(tokenString string) (*Claims, error) {
    token, err := jwt.ParseWithClaims(tokenString, &Claims{}, func(token *jwt.Token) (interface{}, error) {
        return []byte(jwtSecret), nil
    })

    if claims, ok := token.Claims.(*Claims); ok && token.Valid {
        return claims, nil
    }

    return nil, err
}
```

Flutter Auth Service

```dart
class AuthService extends ChangeNotifier {
  String? _token;
  User? _currentUser;
  final ApiService _apiService;

  AuthService(this._apiService);

  bool get isAuthenticated => _token != null;
  User? get currentUser => _currentUser;

  Future<void> login(String email, String password) async {
    try {
      final response = await _apiService.login(email, password);
      _token = response.token;
      _currentUser = response.user;

      _apiService.setAuthToken(_token!);
      await _saveToken(_token!);

      notifyListeners();
    } catch (e) {
      rethrow;
    }
  }

  Future<void> logout() async {
    _token = null;
    _currentUser = null;
    _apiService.setAuthToken(null);
    await _clearToken();
    notifyListeners();
  }

  Future<void> _saveToken(String token) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('auth_token', token);
  }

  Future<void> _clearToken() async {
```

CORS Configuration

Go CORS Setup

```go
func corsMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        origin := r.Header.Get("Origin")

        // Allow specific origins in production
        allowedOrigins := []string{
            "http://localhost:3000",   // Flutter web dev
            "https://myapp.com",       // Production domain
        }

        for _, allowed := range allowedOrigins {
            if origin == allowed {
                w.Header().Set("Access-Control-Allow-Origin", origin)
                break
            }
        }

        w.Header().Set("Access-Control-Allow-Methods",
            "GET, POST, PUT, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers",
            "Content-Type, Authorization, X-Requested-With")
        w.Header().Set("Access-Control-Allow-Credentials", "true")
        w.Header().Set("Access-Control-Max-Age", "86400")

        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }
```

Environment-Specific Configuration

```go
type Config struct {
    Port           string
    DatabaseURL    string
    JWTSecret      string
    AllowedOrigins []string
    Environment    string
}

func loadConfig() *Config {
    return &Config{
        Port:           getEnv("PORT", "8080"),
        DatabaseURL:    getEnv("DATABASE_URL", ""),
        JWTSecret:      getEnv("JWT_SECRET", "dev-secret"),
        AllowedOrigins: strings.Split(
            getEnv("ALLOWED_ORIGINS", "http://localhost:3000"),
            ",",
        ),
        Environment:    getEnv("ENVIRONMENT", "development"),
    }
}

func getEnv(key, defaultValue string) string {
    if value := os.Getenv(key); value != "" {
        return value
    }
```

API Versioning Strategy

Go API Versioning

```go
func setupRoutes() *mux.Router {
    r := mux.NewRouter()

    // API v1
    v1 := r.PathPrefix("/api/v1").Subrouter()
    v1.HandleFunc("/users", getUsersV1).Methods("GET")
    v1.HandleFunc("/users", createUserV1).Methods("POST")

    // API v2 with enhanced features
    v2 := r.PathPrefix("/api/v2").Subrouter()
    v2.HandleFunc("/users", getUsersV2).Methods("GET")
    v2.HandleFunc("/users", createUserV2).Methods("POST")
    v2.HandleFunc("/users/batch", createUsersV2).Methods("POST")

    return r
}

func getUsersV1(w http.ResponseWriter, r *http.Request) {
    // Legacy implementation
    users := getUsersLegacy()
    writeJSON(w, http.StatusOK, users)
}

func getUsersV2(w http.ResponseWriter, r *http.Request) {
    // Enhanced implementation with pagination, filters
    users, pagination := getUsersEnhanced(r)
    response := struct {
        Data       []User     `json:"data"`
        Pagination Pagination `json:"pagination"`
    }{
        Data:       users,
        Pagination: pagination,
```

Flutter Version Handling

```dart
class ApiConfig {
  static const String baseUrl = 'https://api.example.com';
  static const String currentVersion = 'v2';

  static String get apiUrl => '$baseUrl/api/$currentVersion';
}

class UserService {
  Future<List<User>> getUsers() async {
    final response = await http.get(
      Uri.parse('${ApiConfig.apiUrl}/users'),
      headers: _headers,
    );

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);

      // Handle both v1 and v2 response formats
      if (data is List) {
        // v1 format: direct array
        return data.map((json) => User.fromJson(json)).toList();
      } else {
        // v2 format: with pagination
        final List<dynamic> users = data['data'];
        return users.map((json) => User.fromJson(json)).toList();
      }
    } else {
      throw ApiException('Failed to load users');
```

Performance Optimization

Go Server Optimizations

- **Connection pooling**: Reuse database connections
- **Caching**: Redis for frequently accessed data
- **Compression**: Gzip middleware for large responses
- **Rate limiting**: Prevent API abuse

Flutter Client Optimizations

- **HTTP client reuse**: Single client instance
- **Response caching**: Cache static or rarely changing data
- **Pagination**: Load data in chunks
- **Background sync**: Update data when app becomes active

```go
// Go: Response compression
func gzipMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if !strings.Contains(r.Header.Get("Accept-Encoding"), "gzip") {
            next.ServeHTTP(w, r)
            return
        }

        w.Header().Set("Content-Encoding", "gzip")
        gz := gzip.NewWriter(w)
```

```dart
// Flutter: Response caching
class CacheService {
  static final Map<String, CacheEntry> _cache = {};

  static void set(String key, dynamic data, Duration ttl) {
    _cache[key] = CacheEntry(
      data: data,
      expiresAt: DateTime.now().add(ttl),
    );
```

What We've Learned

Fundamental Understanding

- **API Evolution**: From SOAP to REST to modern patterns
- **HTTP Protocol**: Methods, status codes, headers, and semantics
- **REST Principles**: Stateless, cacheable, uniform interface
- **Data Serialization**: JSON vs XML, best practices

Go HTTP Server Mastery

- **net/http package**: Handlers, routing, middleware patterns
- **JSON handling**: Encoding/decoding, struct tags
- **Error handling**: Custom types, consistent responses
- **Testing**: httptest package, table-driven tests
- **Security**: Authentication, CORS, input validation

What We've Learned - Continued

Flutter HTTP Client Excellence

- **http package**: GET, POST, PUT, DELETE operations
- **Data models**: Serialization with fromJson/toJson
- **Error handling**: Custom exceptions, user-friendly messages
- **State management**: Provider pattern with API integration
- **UI patterns**: FutureBuilder, error states, loading indicators

Integration Patterns

- **Authentication**: JWT tokens, secure storage
- **CORS configuration**: Cross-origin request handling
- **API versioning**: Backward compatibility strategies
- **Performance**: Caching, compression, connection pooling

Best Practices Summary

Go API Development

- **Use proper HTTP status codes** for different scenarios
- **Implement comprehensive error handling** with custom types
- **Add middleware** for cross-cutting concerns (logging, auth, CORS)
- **Write tests** for all endpoints and edge cases
- **Validate input data** to prevent security vulnerabilities
- **Use context** for request cancellation and timeouts

Flutter HTTP Client

- **Create centralized API service** classes for maintainability
- **Handle all error scenarios** with user-friendly messages
- **Implement proper loading states** to improve UX
- **Cache responses** when appropriate to reduce network calls
- **Use proper state management** patterns (Provider, Bloc, Riverpod)
- **Dispose HTTP clients** to prevent memory leaks

Thank You!

What's Next:

- Lab 03: Build a complete REST API with Go backend and Flutter frontend

Resources:

- Go HTTP Package: https://pkg.go.dev/net/http
- Flutter HTTP Package: https://pub.dev/packages/http
- REST API Design: https://restfulapi.net/
- Course Repository: https://github.com/timur-harin/sum25-go-flutter-course

Contact:

- Email: timur.harin@mail.com
- Telegram: @timur_harin

Next Lecture: Database & Persistence

Questions?