Go + Flutter Course

Advanced Patterns

Timur Harin

Lecture 05: **Advanced Patterns**

Building scalable, maintainable, and secure applications

Block 5: Advanced Patterns

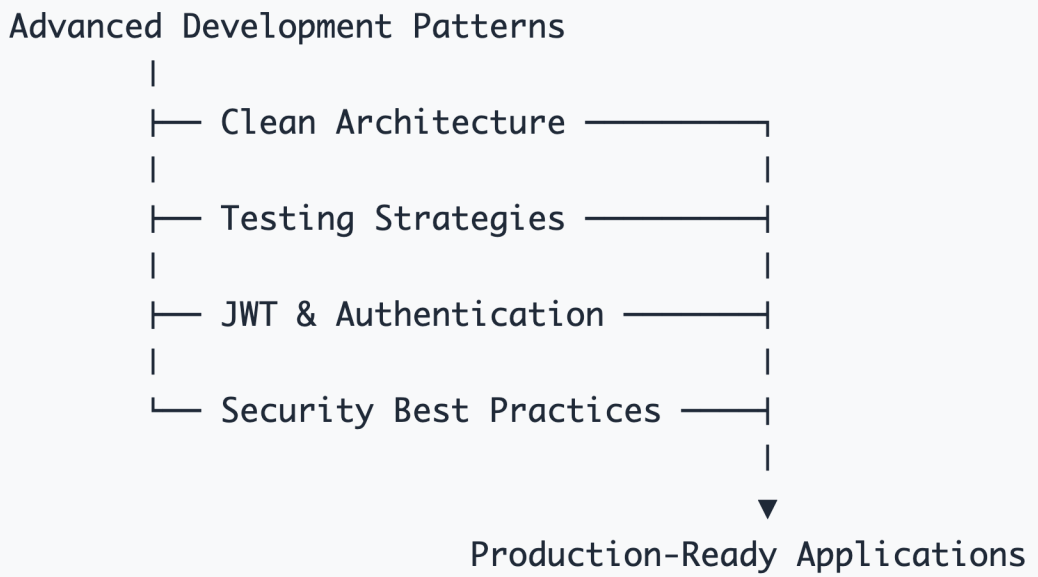
Lecture 05 Overview

- **Clean Architecture**: Domain-driven design principles
- **Testing Strategies**: Unit, integration, and end-to-end testing
- **JWT & Authentication**: Secure user management
- **Security Best Practices**: Protecting applications from threats

What we'll learn

- **Why clean architecture matters for long-term maintenance**
- **Dependency inversion and separation of concerns**
- **Comprehensive testing strategies for both Go and Flutter**
- **JWT implementation with proper security measures**
- **Common security vulnerabilities and prevention**
- **Production-ready security practices**
- **Performance and monitoring considerations**

Learning path



- **Architecture**: Scalable and maintainable code structure
- **Testing**: Comprehensive quality assurance strategies
- **Authentication**: Secure user identity management
- **Security**: Protection against common vulnerabilities

Part I: Clean Architecture

> ***Clean Architecture*** *is a software design philosophy that separates concerns and makes applications independent of frameworks, databases, and external agencies.*

Core principles

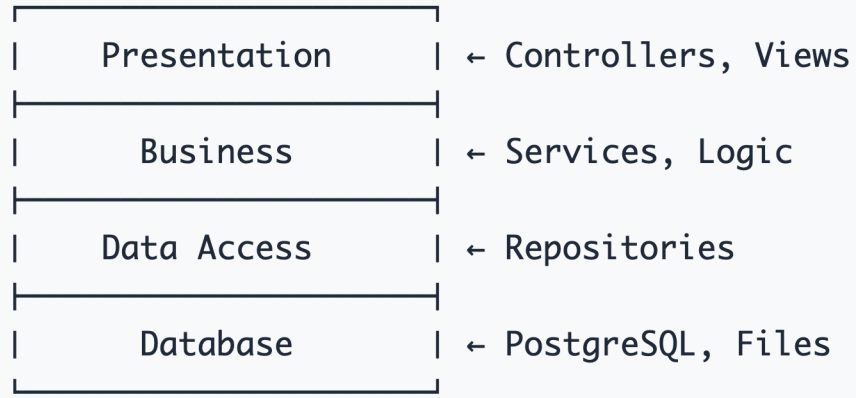
- **Independence**: Business logic doesn't depend on external details
- **Testability**: Each layer can be tested in isolation
- **Flexibility**: Easy to change frameworks, databases, or UI
- **Maintainability**: Clear boundaries and responsibilities

Benefits

- **Reduced coupling**: Changes in one layer don't affect others
- **Improved testability**: Mock external dependencies easily
- **Better team collaboration**: Clear boundaries for different teams
- **Future-proofing**: Easy to adapt to new requirements

Clean Architecture layers

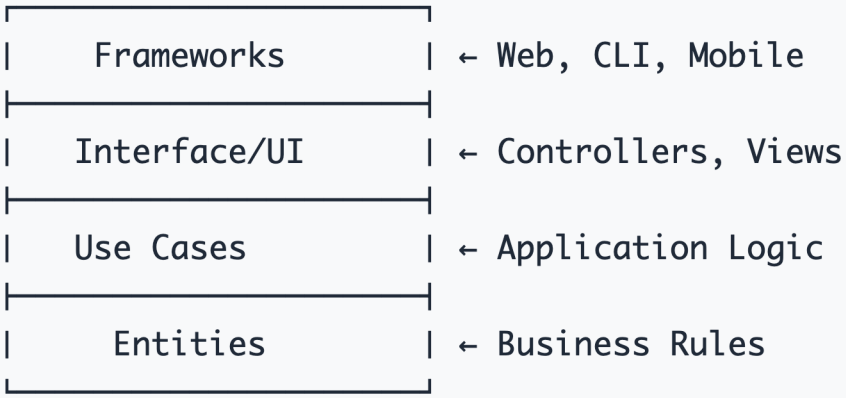
Traditional layered architecture



Problems:

- Database drives design
- Business logic depends on data layer
- Hard to test business rules
- Framework lock-in

Clean architecture layers



Benefits:

- Business rules at center
- Framework independence
- Testable in isolation
- Dependency inversion

Go clean architecture structure

Project structure

```
project/
├── cmd/
|   └── server/          # Main entry point
├── internal/
|   ├── domain/          # Entities & interfaces
|   ├── usecase/         # Application logic
|   ├── repository/      # Data access
|   ├── handler/         # HTTP handlers
|   └── middleware/      # Cross-cutting concerns
├── pkg/                 # Shared utilities
└── migrations/          # Database schemas
```

Dependency direction

```go
// domain/user.go - Core entities
type User struct {
    ID    int
    Name  string
    Email string
}

// domain/interfaces.go - Abstractions
type UserRepository interface {
    Save(user User) error
    FindByID(id int) (*User, error)
}

type UserUseCase interface {
    CreateUser(name, email string) (*User, error)
    GetUser(id int) (*User, error)
}
```

Domain layer implementation

Entities (business objects)

```go
// domain/user.go
type User struct {
    ID        int
    Name      string
    Email     string
    CreatedAt time.Time
}

func (u *User) Validate() error {
    if u.Name == "" {
        return errors.New("name required")
    }
    if !isValidEmail(u.Email) {
        return errors.New("invalid email")
    }
    return nil
}
```

Repository interfaces

```go
// domain/repository.go
type UserRepository interface {
    Save(user *User) error
    FindByID(id int) (*User, error)
    FindByEmail(email string) (*User, error)
    Update(user *User) error
    Delete(id int) error
}

type PostRepository interface {
    Save(post *Post) error
    FindByUserID(userID int) ([]*Post, error)
    FindPublished() ([]*Post, error)
}
```

Use case layer implementation

User use cases

```go
// usecase/user.go
type UserUseCase struct {
    userRepo UserRepository
    logger   Logger
}

func (uc *UserUseCase) CreateUser(name, email string) (*User, error) {
    // Validate input
    user := &User{Name: name, Email: email}
    if err := user.Validate(); err != nil {
        return nil, err
    }

    // Check if user exists
    existing, _ := uc.userRepo.FindByEmail(email)
    if existing != nil {
        return nil, errors.New("user already exists")
    }

    // Save user
    if err := uc.userRepo.Save(user); err != nil {
        uc.logger.Error("Failed to save user", err)
        return nil, err
    }

    return user, nil
}
```

Dependency injection

```go
// usecase/interfaces.go
type UserUseCase interface {
    CreateUser(name, email string) (*User, error)
    GetUser(id int) (*User, error)
    UpdateUser(id int, name, email string) error
    DeleteUser(id int) error
}

// Wire dependencies
func NewUserUseCase(repo UserRepository, logger Logger) UserUseCase {
    return &userUseCase{
        userRepo: repo,
        logger:   logger,
    }
}
```

Repository implementation

PostgreSQL repository

```go
// repository/user_postgres.go
type postgresUserRepository struct {
    db *sql.DB
}

func (r *postgresUserRepository) Save(user *User) error {
    query := `INSERT INTO users (name, email)
              VALUES ($1, $2) RETURNING id, created_at`

    return r.db.QueryRow(query, user.Name, user.Email).
        Scan(&user.ID, &user.CreatedAt)
}

func (r *postgresUserRepository) FindByID(id int) (*User, error) {
    query := `SELECT id, name, email, created_at
              FROM users WHERE id = $1`

    user := &User{}
    err := r.db.QueryRow(query, id).
        Scan(&user.ID, &user.Name, &user.Email, &user.CreatedAt)

    return user, err
}
```

Memory repository (for testing)

```go
// repository/user_memory.go
type memoryUserRepository struct {
    users map[int]*User
    mutex sync.RWMutex
    nextID int
}

func (r *memoryUserRepository) Save(user *User) error {
    r.mutex.Lock()
    defer r.mutex.Unlock()

    r.nextID++
    user.ID = r.nextID
    user.CreatedAt = time.Now()
    r.users[user.ID] = user

    return nil
}

func (r *memoryUserRepository) FindByID(id int) (*User, error) {
    r.mutex.RLock()
    defer r.mutex.RUnlock()

    user, exists := r.users[id]
    if !exists {
        return nil, errors.New("user not found")
    }
```

Flutter clean architecture

Project structure

```
lib/
├── core/
|   ├── error/           # Error handling
|   ├── network/         # Network utilities
|   └── usecases/        # Base use case
├── features/
|   └── user/
|       ├── data/        # Data sources & repos
|       ├── domain/      # Entities & use cases
|       └── presentation/# UI & state management
└── injection/           # Dependency injection
```

Entities

```dart
// features/user/domain/entities/user.dart
class User extends Equatable {
  final int id;
  final String name;
  final String email;
  final DateTime createdAt;

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

  @override
  List<Object> get props => [id, name, email, createdAt];
}
```

Flutter use cases

Base use case

```dart
// core/usecases/usecase.dart
abstract class UseCase<Type, Params> {
  Future<Either<Failure, Type>> call(Params params);
}

class NoParams extends Equatable {
  @override
  List<Object> get props => [];
}
```

Specific use case

```dart
// features/user/domain/usecases/get_user.dart
class GetUser implements UseCase<User, GetUserParams> {
  final UserRepository repository;

  GetUser(this.repository);

  @override
  Future<Either<Failure, User>> call(GetUserParams params) async {
    return await repository.getUser(params.id);
  }
```

Repository contract

```dart
// features/user/domain/repositories/user_repository.dart
abstract class UserRepository {
  Future<Either<Failure, User>> getUser(int id);
  Future<Either<Failure, User>> createUser(String name, String email);
  Future<Either<Failure, List<User>>> getUsers();
}

// Parameters class
class GetUserParams extends Equatable {
  final int id;

  const GetUserParams({required this.id});

  @override
  List<Object> get props => [id];
}
```

Data layer implementation

Data models

```dart
// features/user/data/models/user_model.dart
class UserModel extends User {
  const UserModel({
    required super.id,
    required super.name,
    required super.email,
    required super.createdAt,
  });

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

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

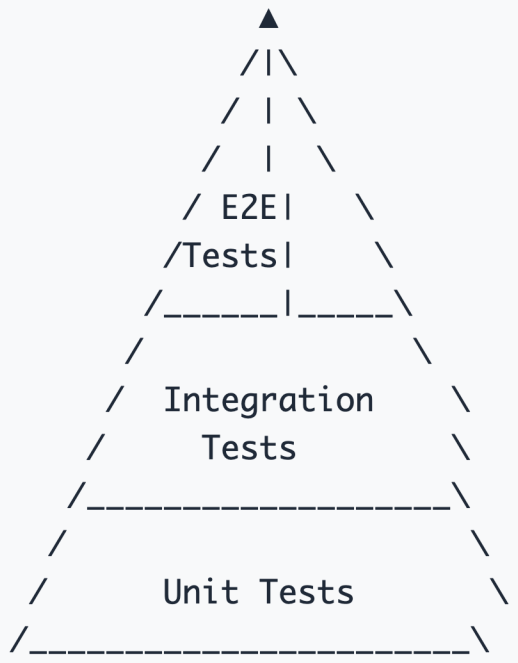
Repository implementation

```dart
// features/user/data/repositories/user_repository_impl.dart
class UserRepositoryImpl implements UserRepository {
  final UserRemoteDataSource remoteDataSource;
  final UserLocalDataSource localDataSource;
  final NetworkInfo networkInfo;

  UserRepositoryImpl({
    required this.remoteDataSource,
    required this.localDataSource,
    required this.networkInfo,
  });

  @override
  Future<Either<Failure, User>> getUser(int id) async {
    if (await networkInfo.isConnected) {
      try {
        final user = await remoteDataSource.getUser(id);
        await localDataSource.cacheUser(user);
        return Right(user);
      } catch (e) {
        return Left(ServerFailure());
      }
    } else {
      try {
        final user = await localDataSource.getUser(id);
        return Right(user);
      } catch (e) {
        return Left(CacheFailure());
      }
```

Part II: Testing Strategies

> ***Testing*** *is crucial for building reliable applications. Different types of tests serve different purposes and provide varying levels of confidence.*

Testing pyramid



Go testing fundamentals

Basic unit test

```go
// user_test.go
func TestUser_Validate(t *testing.T) {
    tests := []struct {
        name    string
        user    User
        wantErr bool
    }{
        {"valid user", User{Name: "John", Email: "john@test.com"}, false},
        {"empty name", User{Name: "", Email: "john@test.com"}, true},
        {"invalid email", User{Name: "John", Email: "invalid"}, true},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            err := tt.user.Validate()
            if (err != nil) != tt.wantErr {
                t.Errorf("Validate() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}
```

Testing with mocks

```go
// usecase_test.go
func TestUserUseCase_CreateUser(t *testing.T) {
    mockRepo := &MockUserRepository{}
    mockLogger := &MockLogger{}
    useCase := NewUserUseCase(mockRepo, mockLogger)

    // Setup mock expectations
    mockRepo.On("FindByEmail", "test@example.com").Return(nil, nil)
    mockRepo.On("Save", mock.AnythingOfType("*User")).Return(nil)

    // Execute
    user, err := useCase.CreateUser("Test User", "test@example.com")

    // Assert
    assert.NoError(t, err)
    assert.Equal(t, "Test User", user.Name)
    mockRepo.AssertExpectations(t)
}
```

Testing repository layer

Integration test with database

```go
// repository_test.go
func TestPostgresUserRepository_Save(t *testing.T) {
    db := setupTestDB(t)
    defer cleanupTestDB(t, db)

    repo := NewPostgresUserRepository(db)
    user := &User{
        Name:  "Test User",
        Email: "test@example.com",
    }

    err := repo.Save(user)

    assert.NoError(t, err)
    assert.NotZero(t, user.ID)
    assert.NotZero(t, user.CreatedAt)
}
```

Test helpers

```go
// test_helpers.go
func setupTestDB(t *testing.T) *sql.DB {
    db, err := sql.Open("postgres", "postgres://test:test@localhost/testdb?sslmode=disable")
    require.NoError(t, err)

    // Run migrations
    err = runMigrations(db)
    require.NoError(t, err)

    return db
}

func cleanupTestDB(t *testing.T, db *sql.DB) {
    _, err := db.Exec("TRUNCATE users, posts RESTART IDENTITY CASCADE")
    require.NoError(t, err)
    db.Close()
}
```

Flutter testing approaches

Unit testing entities

```dart
// test/features/user/domain/entities/user_test.dart
void main() {
  group('User', () {
    test('should be a subclass of Equatable', () {
      // arrange
      const user = User(id: 1, name: 'Test', email: 'test@test.com', createdAt: DateTime(2025));

      // assert
      expect(user, isA<Equatable>());
    });

    test('should return correct props', () {
      // arrange
      const user = User(id: 1, name: 'Test', email: 'test@test.com', createdAt: DateTime(2025));

      // assert
      expect(user.props, [1, 'Test', 'test@test.com', DateTime(2025)]);
    });
  });
}
```

Testing use cases

```dart
// test/features/user/domain/usecases/get_user_test.dart
void main() {
  late GetUser usecase;
  late MockUserRepository mockRepository;

  setUp(() {
    mockRepository = MockUserRepository();
    usecase = GetUser(mockRepository);
  });

  test('should get user from repository', () async {
    // arrange
    const testUser = User(id: 1, name: 'Test', email: 'test@test.com', createdAt: DateTime(2025));
    when(mockRepository.getUser(any)).thenAnswer((_) async => const Right(testUser));

    // act
    final result = await usecase(const GetUserParams(id: 1));

    // assert
    expect(result, const Right(testUser));
    verify(mockRepository.getUser(1));
    verifyNoMoreInteractions(mockRepository);
  });
}
```

Widget testing

Basic widget test

```dart
// test/features/user/presentation/widgets/user_card_test.dart
void main() {
  testWidgets('UserCard displays user information', (tester) async {
    // arrange
    const user = User(
      id: 1,
      name: 'John Doe',
      email: 'john@example.com',
      createdAt: DateTime(2025),
    );

    // act
    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          body: UserCard(user: user),
        ),
      ),
    );

    // assert
    expect(find.text('John Doe'), findsOneWidget);
    expect(find.text('john@example.com'), findsOneWidget);
  });
}
```

Testing with BLoC

```dart
// test/features/user/presentation/bloc/user_bloc_test.dart
void main() {
  late UserBloc bloc;
  late MockGetUser mockGetUser;

  setUp(() {
    mockGetUser = MockGetUser();
    bloc = UserBloc(getUser: mockGetUser);
  });

  group('GetUserEvent', () {
    test('should emit [Loading, Loaded] when data is gotten successfully', () {
      // arrange
      when(mockGetUser(any)).thenAnswer((_) async => const Right(testUser));

      // assert later
      final expected = [
        UserLoading(),
        const UserLoaded(user: testUser),
      ];
      expectLater(bloc.stream, emitsInOrder(expected));

      // act
      bloc.add(const GetUserEvent(id: 1));
    });
  });
}
```

Test coverage and quality

Go test coverage

```
# Run tests with coverage
go test -v -cover ./...

# Generate detailed coverage report
go test -coverprofile=coverage.out ./...
go tool cover -html=coverage.out

# Set coverage threshold
go test -cover ./... | grep "coverage:"
```

Flutter test coverage

```
# Run tests with coverage
flutter test --coverage

# Generate HTML report
genhtml coverage/lcov.info -o coverage/html

# View coverage
open coverage/html/index.html
```

Coverage goals

- **Domain layer**: 95%+ (business logic)
- **Use cases**: 90%+ (application logic)
- **Repositories**: 80%+ (data access)
- **Handlers**: 70%+ (HTTP layer)

Best practices

- **Test behavior, not implementation**
- **One assertion per test**
- **Use descriptive test names**
- **Setup and teardown properly**
- **Mock external dependencies**

Part III: JWT & Authentication

> ***JWT (JSON Web Tokens)*** *provide a stateless way to handle authentication and authorization in distributed systems.*

JWT structure

```
Header.Payload.Signature
```

Components

- **Header**: Algorithm and token type
- **Payload**: Claims (user data, permissions)
- **Signature**: Cryptographic signature for verification

Benefits

- **Stateless**: No server-side session storage
- **Scalable**: Works across multiple services
- **Portable**: Can include user claims

JWT implementation in Go

JWT service

```go
// internal/auth/jwt.go
type JWTService struct {
    secretKey string
    issuer    string
    expiry    time.Duration
}

func (j *JWTService) GenerateToken(userID int, email string) (string, error) {
    claims := jwt.MapClaims{
        "user_id": userID,
        "email":   email,
        "iss":     j.issuer,
        "exp":     time.Now().Add(j.expiry).Unix(),
        "iat":     time.Now().Unix(),
    }

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

Token validation

```go
func (j *JWTService) ValidateToken(tokenString string) (*Claims, error) {
    token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("unexpected signing method")
        }
        return []byte(j.secretKey), nil
    })

    if err != nil || !token.Valid {
        return nil, errors.New("invalid token")
    }

    claims, ok := token.Claims.(jwt.MapClaims)
    if !ok {
        return nil, errors.New("invalid claims")
    }

    return &Claims{
        UserID: int(claims["user_id"].(float64)),
        Email:  claims["email"].(string),
    }, nil
}
```

Authentication middleware

JWT middleware

```go
// internal/middleware/auth.go
func JWTAuthMiddleware(jwtService *JWTService) func(http.Handler) http.Handler {
    return func(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
            }

            tokenString := strings.TrimPrefix(authHeader, "Bearer ")
            claims, err := jwtService.ValidateToken(tokenString)
            if err != nil {
                http.Error(w, "Invalid token", http.StatusUnauthorized)
                return
            }

            ctx := context.WithValue(r.Context(), "user", claims)
            next.ServeHTTP(w, r.WithContext(ctx))
        })
    }
}
```

Protected routes

```go
// internal/handlers/protected.go
func GetUserProfile(w http.ResponseWriter, r *http.Request) {
    claims, ok := r.Context().Value("user").(*Claims)
    if !ok {
        http.Error(w, "User not found in context", http.StatusInternalServerError)
        return
    }

    user, err := userService.GetUser(claims.UserID)
    if err != nil {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }

    writeJSON(w, http.StatusOK, user)
}

// Apply middleware
r.Handle("/api/profile", JWTAuthMiddleware(jwtService)(http.HandlerFunc(GetUserProfile)))
```

Flutter JWT handling

Token storage

```dart
// core/auth/token_storage.dart
class TokenStorage {
  static const _tokenKey = 'auth_token';
  static const _refreshTokenKey = 'refresh_token';

  static Future<void> saveTokens(String token, String refreshToken) async {
    await SecureStorage.write(_tokenKey, token);
    await SecureStorage.write(_refreshTokenKey, refreshToken);
  }

  static Future<String?> getToken() async {
    return await SecureStorage.read(_tokenKey);
  }

  static Future<String?> getRefreshToken() async {
    return await SecureStorage.read(_refreshTokenKey);
  }

  static Future<void> clearTokens() async {
    await SecureStorage.delete(_tokenKey);
    await SecureStorage.delete(_refreshTokenKey);
  }
}
```

Auth interceptor

```dart
// core/network/auth_interceptor.dart
class AuthInterceptor extends Interceptor {
  @override
  void onRequest(RequestOptions options, RequestInterceptorHandler handler) async {
    final token = await TokenStorage.getToken();
    if (token != null) {
      options.headers['Authorization'] = 'Bearer $token';
    }
    super.onRequest(options, handler);
  }

  @override
  void onError(DioError err, ErrorInterceptorHandler handler) async {
    if (err.response?.statusCode == 401) {
      final refreshed = await _refreshToken();
      if (refreshed) {
        // Retry request with new token
        final newToken = await TokenStorage.getToken();
        err.requestOptions.headers['Authorization'] = 'Bearer $newToken';
        final response = await Dio().fetch(err.requestOptions);
        return handler.resolve(response);
      }
    }
    super.onError(err, handler);
  }
}
```

Part IV: Security Best Practices

> ***Security*** *is not an afterthought but should be built into every layer of your application from the beginning.*

Common vulnerabilities (OWASP Top 10)

- **Injection**: SQL, NoSQL, Command injection
- **Broken Authentication**: Weak passwords, session management
- **Sensitive Data Exposure**: Unencrypted data transmission/storage
- **XML External Entities (XXE)**: Processing untrusted XML
- **Broken Access Control**: Unauthorized access to resources
- **Security Misconfiguration**: Default configs, unnecessary features
- **Cross-Site Scripting (XSS)**: Malicious script injection
- **Insecure Deserialization**: Arbitrary code execution
- **Using Components with Known Vulnerabilities**: Outdated dependencies

Input validation and sanitization

Go input validation

```go
// internal/validators/user.go
type CreateUserRequest struct {
    Name     string `json:"name" validate:"required,min=2,max=100"`
    Email    string `json:"email" validate:"required,email"`
    Password string `json:"password" validate:"required,min=8"`
}

func ValidateCreateUserRequest(req *CreateUserRequest) error {
    validate := validator.New()
    if err := validate.Struct(req); err != nil {
        return fmt.Errorf("validation failed: %w", err)
    }

    // Custom business rules
    if containsBadWords(req.Name) {
        return errors.New("name contains inappropriate content")
    }

    return nil
}
```

SQL injection prevention

```go
// ❌ Vulnerable to SQL injection
query := fmt.Sprintf("SELECT * FROM users WHERE email = '%s'", email)

// ✅ Safe with prepared statements
query := "SELECT * FROM users WHERE email = $1"
rows, err := db.Query(query, email)

// ✅ Safe with query builder
query := squirrel.Select("*").From("users").Where(squirrel.Eq{"email": email})
sql, args, _ := query.ToSql()
rows, err := db.Query(sql, args...)
```

Password security

Password hashing

```go
// internal/auth/password.go
import "golang.org/x/crypto/bcrypt"

func HashPassword(password string) (string, error) {
    // Use bcrypt with cost 12 (recommended)
    bytes, err := bcrypt.GenerateFromPassword([]byte(password), 12)
    return string(bytes), err
}

func CheckPassword(password, hash string) bool {
    err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
    return err == nil
}

// Password strength validation
func ValidatePasswordStrength(password string) error {
    if len(password) < 8 {
        return errors.New("password must be at least 8 characters")
    }
    // Add more complexity rules
    return nil
}
```

Rate limiting

```go
// internal/middleware/rate_limit.go
import "golang.org/x/time/rate"

func RateLimitMiddleware(rps rate.Limit, burst int) func(http.Handler) http.Handler {
    limiter := rate.NewLimiter(rps, burst)

    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if !limiter.Allow() {
                http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
                return
            }
            next.ServeHTTP(w, r)
        })
    }
}

// Apply to login endpoint
r.Handle("/api/login", RateLimitMiddleware(rate.Every(time.Second), 5)(loginHandler))
```

HTTPS and TLS configuration

TLS server configuration

```
// cmd/server/main.go
func setupTLSServer() *http.Server {
    server := &http.Server{
        Addr:         ":8443",
        Handler:      setupRoutes(),
        ReadTimeout:  15 * time.Second,
        WriteTimeout: 15 * time.Second,
        IdleTimeout:  60 * time.Second,
        TLSConfig: &tls.Config{
            MinVersion:         tls.VersionTLS12,
            CurvePreferences:   []tls.CurveID{tls.CurveP521, tls.CurveP384, tls.CurveP256},
            PreferServerCiphers: true,
            CipherSuites: []uint16{
                tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
                tls.TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,
                tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
            },
        },
    }
    return server
}
```

Security headers

```
// internal/middleware/security.go
func SecurityHeadersMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // HTTPS enforcement
        w.Header().Set("Strict-Transport-Security", "max-age=31536000; includeSubDomains")

        // XSS Protection
        w.Header().Set("X-Content-Type-Options", "nosniff")
        w.Header().Set("X-Frame-Options", "DENY")
        w.Header().Set("X-XSS-Protection", "1; mode=block")

        // Content Security Policy
        w.Header().Set("Content-Security-Policy", "default-src 'self'")

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

Flutter security considerations

Certificate pinning

```dart
// core/network/certificate_pinning.dart
class CertificatePinner {
  static Dio createPinnedClient() {
    final dio = Dio();

    // Add certificate pinning
    (dio.httpClientAdapter as DefaultHttpClientAdapter).onHttpClientCreate = (client) {
      client.badCertificateCallback = (cert, host, port) {
        // Verify certificate fingerprint
        final fingerprint = sha256.convert(cert.der).toString();
        return fingerprint == expectedFingerprint;
      };
      return client;
    };

    return dio;
  }
}
```

Secure data handling

```dart
// core/security/data_protection.dart
class DataProtection {
  // Encrypt sensitive data before storage
  static Future<String> encryptData(String data) async {
    final key = await _getOrCreateKey();
    final encrypted = AES(key).encrypt(data);
    return encrypted.base64;
  }

  // Sanitize sensitive data from logs
  static Map<String, dynamic> sanitizeForLogging(Map<String, dynamic> data) {
    final sanitized = Map<String, dynamic>.from(data);

    // Remove sensitive fields
    sanitized.remove('password');
    sanitized.remove('token');
    sanitized.remove('credit_card');

    return sanitized;
  }
}
```

Security checklist

Backend Security ✅

- [ ] **Input validation** on all endpoints
- [ ] **SQL injection** prevention with prepared statements
- [ ] **Authentication** with secure JWT implementation
- [ ] **Rate limiting** on sensitive endpoints
- [ ] **HTTPS** with proper TLS configuration
- [ ] **Security headers** (HSTS, CSP, X-Frame-Options)
- [ ] **Password hashing** with bcrypt or similar
- [ ] **Environment variables** for secrets
- [ ] **Dependency scanning** for vulnerabilities
- [ ] **Error handling** without information leakage

Frontend Security ✅

- [ ] **Certificate pinning** for API connections
- [ ] **Secure storage** for sensitive data
- [ ] **Input sanitization** for user inputs
- [ ] **Token management** with automatic refresh
- [ ] **Deep link validation** and authorization

What we've learned

Clean Architecture

- **Separation of concerns**: Domain, use case, and infrastructure layers
- **Dependency inversion**: Abstract interfaces over concrete implementations
- **Testability**: Each layer can be tested in isolation
- **Maintainability**: Clear boundaries and responsibilities

Testing Strategies

- **Testing pyramid**: Unit, integration, and end-to-end tests
- **Mocking**: Isolating units under test from dependencies
- **Coverage**: Measuring and improving test coverage
- **Quality**: Best practices for reliable and maintainable tests

What we've learned (continued)

JWT & Authentication

- **Stateless authentication**: JWT structure and benefits
- **Token management**: Generation, validation, and refresh
- **Security considerations**: Secret management and token expiry
- **Implementation**: Go JWT services and Flutter token handling

Security Best Practices

- **Input validation**: Preventing injection attacks
- **Password security**: Proper hashing and strength requirements
- **HTTPS/TLS**: Secure communication channels
- **Security headers**: Browser protection mechanisms
- **Mobile security**: Certificate pinning and secure storage

Thank You!

What's Next:

- Lab 05: Implement clean architecture with comprehensive testing and security

Resources:

- Clean Architecture: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Go Testing: https://pkg.go.dev/testing
- Flutter Testing: https://docs.flutter.dev/testing
- JWT Best Practices: https://auth0.com/blog/a-look-at-the-latest-draft-for-jwt-bcp/
- OWASP Top 10: https://owasp.org/www-project-top-ten/
- Course Repository: https://github.com/timur-harin/sum25-go-flutter-course

Contact:

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

Next Lecture: Deployment & DevOps

Questions?