Go Development Guide for Safe Security

For New Engineers | Service-Agnostic Patterns & Standards

Table of Contents

---

1. Go Fundamentals at SAFE

1.1 Key Libraries We Use

Library	Purpose
Gorilla Mux	HTTP routing and request handling
GORM	ORM for database operations
Viper	Configuration management
Logrus	Structured logging
Goose	Database migrations
gomock	Mock generation for testing
testify	Test assertions
OpenTelemetry	Distributed tracing

1.2 Internal Shared Library

We maintain `safe-go-libraries` - a private Go module with common utilities:

```Go
import (
    "github.com/safe-security-enterprise/safe-go-libraries/appcontext"
    "github.com/safe-security-enterprise/safe-go-libraries/utils/errors"
    "github.com/safe-security-enterprise/safe-go-libraries/interceptor"
)
```

Key Packages:

- `appcontext` - Logger, tenant context, request context keys
- `utils/errors` - Error wrapping with stack traces
- `interceptor` - HTTP middleware (auth, logging, panic recovery)
- `utils/route` - Route definitions
- `migrator` - Multi-tenant database migrations

> **Rule**: Never duplicate functionality from `safe-go-libraries`. Always use the shared library.

1.3 Context is King

In Go at SAFE, `context.Context` flows through every layer:

```Go
func (s *Service) DoSomething(ctx context.Context, input *Input) (*Output,
error) {
    // Context carries: tenantId, requestId, traceId, logger, database accessor
    logger := appcontext.GetLogger(ctx)
    tenantId := ctx.Value(appcontext.TenantId).(string)
    db := datastore.GetDataStore(ctx)
    // ...
}
```

Context Keys (from appcontext):

```go
const (
    TenantId         = "tenantId"      // Multi-tenant isolation
    ReqId            = "requestId"     // Request tracking
    TraceId          = "traceId"       // Distributed tracing
    UUID             = "uuid"          // User ID
    Datastore        = "datastore"     // Database accessor
    GormAccessor     = "gormaccessor"  // GORM DB connection
)
```

---

2. Service Structure

2.1 Directory Layout

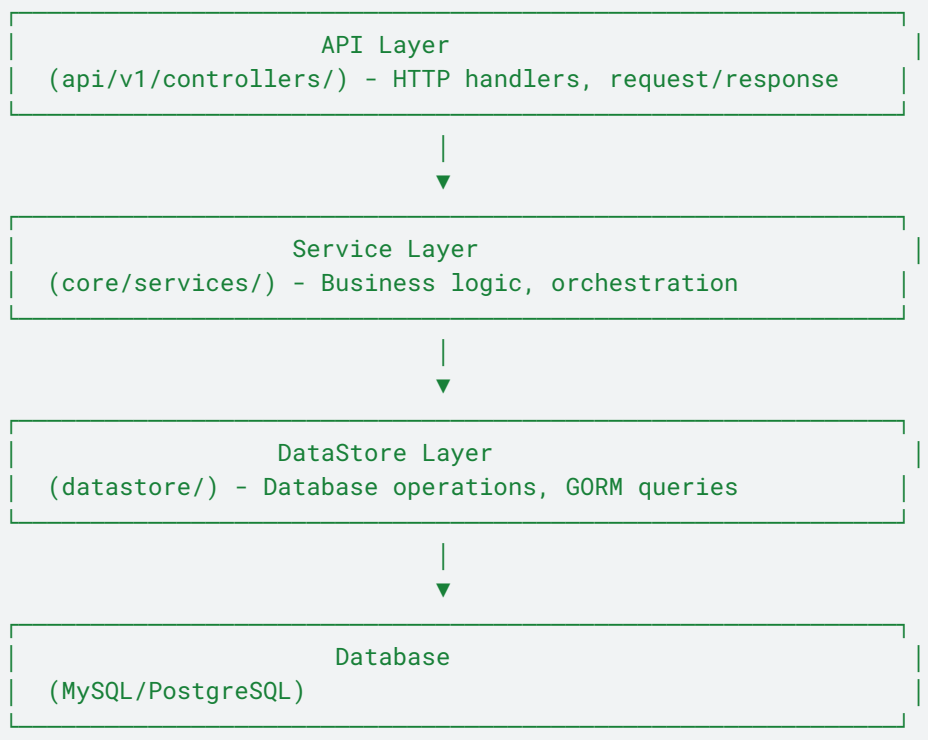
Every Go service follows this structure:

```
service-name/
├── cmd/
│   └── main.go                # Entry point (--api, --migrate flags)
│
├── api/v1/
│   ├── router.go              # Route registration
│   ├── controllers/           # HTTP handlers
│   ├── helpers/               # Request parsing, validation
│   └── types/                 # Request/Response DTOs
│
├── core/
│   ├── services/              # Business logic (interfaces + implementations)
│   │   └── fake/              # Mock implementations
│   ├── model/                 # GORM models, domain entities
│   └── types/                 # Shared types, client aggregations
│
├── datastore/
│   ├── *store.go              # Repository implementations
│   ├── helpers/               # Query builders
│   └── fake/                  # Mock datastore
│
├── config/                    # Viper configuration getters
├── constants/                 # ALL constants (no hardcoding!)
├── dbmigrations/
│   └── migrations/mysql/      # Goose migration files
```

```
│
├── util/
│   └── externalutils/          # External service clients
│
├── tests/
│   ├── integration/            # Integration tests
│   └── storm/                  # E2E API tests
│
├── Makefile or Taskfile.yaml   # Build commands
├── .env.sample                 # Environment template
└── go.mod                      # Go module definition
```

2.2 Three-Layer Architecture

None



Golden Rule: Data flows down, errors flow up. Never skip layers.

---

3. API Patterns

3.1 Route Registration

Routes are registered in `api/v1/router.go`:

```go
func AddRoutesToRouter(router *mux.Router) {
    ctrl := v1controller.NewController()

    var routes = []route.Route{
        {
            Path:                  "/internal/v1/assets",
            RequestMethod:         http.MethodGet,
            Handler:
v1controller.ResponseHandler(ctrl.GetAssetsHandler),
            SkipAuth:              false,        // Require authentication
            SnowflakeVerification: false,        // Verify Snowflake access
            DoNotLog:              false,        // Log this request
        },
        {
            Path:                  "/internal/v1/assets/{id}",
            RequestMethod:         http.MethodPut,
            Handler:
v1controller.ResponseHandler(ctrl.UpdateAssetHandler),
            SkipAuth:              false,
        },
    }

    for _, newRoute := range routes {
        // Build interceptor chain
        interceptors := []interceptor.Interceptor{
            interceptor.GetMuxMapAddingInterceptor(),
            interceptor.GetRequestInfoLoggingInterceptor(newRoute.DoNotLog),
        }
        if !newRoute.SkipAuth {
            interceptors = append(interceptors,
                interceptor.AuthResolverInterceptor(v1controller.ErrorHandler))
        }
        // ... add more interceptors

        handler := interceptor.Intercept(newRoute.Handler, interceptors...)
        handler = http.TimeoutHandler(handler, config.GetHandlerTimeout(),
"{}")
        router.Handle(newRoute.Path, handler).Methods(newRoute.RequestMethod)
    }
```

```
}
```

3.2 Controller Pattern

Controllers handle HTTP concerns only - no business logic:

```Go
type controller struct {
    // External clients
    kafkaClient    kafkaclient.KafkaInterface

    // Service dependencies (business logic)
    assetService   services.AssetServiceInterface
    groupService   services.GroupServiceInterface
}

func NewController() *controller {
    // Initialize clients
    kafkaClient, _ := kafkaclient.NewKafKaClient(&kafkaConfig{})

    // Create clients struct for injection
    clients := &coreTypes.Clients{
        KafkaClient: kafkaClient,
    }

    // Initialize services with dependencies
    return &controller{
        kafkaClient:  kafkaClient,
        assetService: services.NewAssetService(clients),
        groupService: services.NewGroupService(clients),
    }
}
```

3.3 Handler Function Pattern

```Go
func (c *controller) GetAssetsHandler(_ http.ResponseWriter, r *http.Request)
(*types.GenericAPIResponse, error) {
    TAG := "[GetAssets]"
    ctx := r.Context()
    logger := appcontext.GetLogger(ctx)

    // 1. Parse and validate request
```

```
    params, err := helpers.ValidateAndParseAssetListParams(ctx, r)
    if err != nil {
        return nil, err  // ErrorHandler formats the response
    }

    // 2. Call service layer
    result, err := c.assetService.GetAssets(ctx, params)
    if err != nil {
        logger.WithError(err).Error(TAG, "error getting assets")
        return nil, err
    }

    // 3. Format and return response
    return &types.GenericAPIResponse{
        StatusCode: http.StatusOK,
        PaginatedAPIResponse: &types.PaginatedAPIResponse{
            Page:   params.Pagination.Page,
            Size:   result.TotalCount,
            Values: formatAssetsResponse(result.Assets),
        },
    }, nil
}
```

3.4 Response/Error Handlers

Go

```
// ResponseHandler wraps handlers for consistent response formatting
func ResponseHandler(f types.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        res, err := f(w, r)
        if err != nil {
            ErrorHandler(err, w, r)
            return
        }
        writeGenericResponse(res, w, r)
    }
}

// ErrorHandler formats error responses
func ErrorHandler(err error, w http.ResponseWriter, r *http.Request) {
    response := &types.GenericAPIResponse{
        Success: false,
        Message: types.ErrInternalServerText,
    }
    statusCode := http.StatusInternalServerError
```

```
    // Check for HTTP errors with status codes
    if httpErr, ok := errors.IsHttpError(err); ok {
        statusCode = httpErr.StatusCode
        response.Message = httpErr.Message
    }

    response.StatusCode = statusCode
    writeGenericResponse(response, w, r)
}
```

3.5 API Response Format

Success Response:

```JSON
{
  "success": true,
  "statusCode": 200,
  "message": "Assets retrieved successfully",
  "page": 1,
  "pagelen": 10,
  "size": 150,
  "values": [...]
}
```

Error Response:

```JSON
{
  "success": false,
  "statusCode": 400,
  "message": "Validation error",
  "error": [
    { "field": "name", "message": "name is required" }
  ]
}
```

3.6 Interceptors (Middleware)

Common interceptors from `safe-go-libraries`:

Interceptor	Purpose
`GetMuxMapAddingInterceptor`	Adds route params to context
`GetRequestInfoLoggingInterceptor`	Logs request details
`AuthResolverInterceptor`	Validates JWT, sets tenantId/userId
`AuthoriseInterceptor`	RBAC authorization via Casbin
`GetApiPanicHandlerInterceptor`	Recovers from panics, returns 500

---

4. Database Access Layer

4.1 DataStore Interface

Define interfaces in `core/model/datastore.go`:

```go
//go:generate mockgen -destination=../../datastore/fake/fake.go -package=fake .
DataStore
type DataStore interface {
    // Composed interfaces
    AssetStore
    GroupStore
    FindingStore

    // Transaction support
    Begin(ctx context.Context) (DataStore, error)
    Commit(ctx context.Context) error
    Rollback(ctx context.Context) error
    InTransactionMode(ctx context.Context) bool
    WithContext(ctx context.Context) DataStore
}

type AssetStore interface {
    CreateAsset(ctx context.Context, asset *Asset) (*Asset, error)
    GetAsset(ctx context.Context, id int64) (*Asset, error)
    UpdateAsset(ctx context.Context, id int64, updates map[string]interface{})
error
    DeleteAsset(ctx context.Context, id int64) error
    GetPaginatedAssets(ctx context.Context, params *AssetListParams)
(*AssetListResult, error)
```

```
}
```

4.2 DataStore Implementation

```go
type DBStore struct {
    db                *gorm.DB
    inTransactionMode bool
}

func NewDataStore(db *gorm.DB) *DBStore {
    return &DBStore{db: db}
}

// Get datastore from context (tenant-isolated)
func GetDataStore(ctx context.Context) model.DataStore {
    gormAccessor := ctx.Value(appcontext.GormAccessor)
    if accessor, ok := gormAccessor.(appcontext.TenantGormAccessor); ok {
        return NewDataStore(accessor.GetDB())
    }
    return ctx.Value(appcontext.Datastore).(model.DataStore)
}
```

4.3 GORM Query Patterns

Simple Query:

```go
func (ds *DBStore) GetAsset(ctx context.Context, id int64) (*model.Asset,
error) {
    var asset model.Asset
    err := ds.db.WithContext(ctx).
        Where("id = ?", id).          // ALWAYS parameterized!
        First(&asset).Error
    if err != nil {
        return nil, errors.Wrap(err, "failed to get asset")
    }
    return &asset, nil
}
```

Create:

```go
func (ds *DBStore) CreateAsset(ctx context.Context, asset *model.Asset)
(*model.Asset, error) {
    if err := ds.db.WithContext(ctx).Create(asset).Error; err != nil {
        return nil, errors.Wrap(err, "failed to create asset")
    }
    return asset, nil
}
```

Update:

```go
func (ds *DBStore) UpdateAsset(ctx context.Context, id int64, updates
map[string]interface{}) error {
    result := ds.db.WithContext(ctx).
        Model(&model.Asset{}).
        Where("id = ?", id).
        Updates(updates)
    if result.Error != nil {
        return errors.Wrap(result.Error, "failed to update asset")
    }
    return nil
}
```

Paginated Query with Filters:

```go
func (ds *DBStore) GetPaginatedAssets(ctx context.Context, params
*AssetListParams) (*AssetListResult, error) {
    var assets []model.Asset
    var totalCount int64

    query := ds.db.WithContext(ctx).Model(&model.Asset{})

    // Apply filters
    if params.Name != "" {
        query = query.Where("name LIKE ?", "%"+params.Name+"%")
    }
    if params.Status != "" {
        query = query.Where("status = ?", params.Status)
    }
```

```
    // Get total count
    if err := query.Count(&totalCount).Error; err != nil {
        return nil, errors.Wrap(err, "failed to count assets")
    }

    // Apply pagination
    offset := (params.Page - 1) * params.PageSize
    if err := query.Offset(offset).Limit(params.PageSize).Find(&assets).Error;
err != nil {
        return nil, errors.Wrap(err, "failed to get assets")
    }

    return &AssetListResult{Assets: assets, TotalCount: totalCount}, nil
}
```

4.4 Transactions

```
Go
func (s *AssetService) CreateAssetWithRelations(ctx context.Context, input
*CreateInput) error {
    db := datastore.GetDataStore(ctx)

    // Begin transaction
    tx, err := db.Begin(ctx)
    if err != nil {
        return errors.Wrap(err, "failed to begin transaction")
    }

    // Ensure rollback on panic or error
    defer func() {
        if r := recover(); r != nil {
            tx.Rollback(ctx)
            panic(r)
        }
    }()

    // Operation 1
    asset, err := tx.CreateAsset(ctx, input.Asset)
    if err != nil {
        tx.Rollback(ctx)
        return errors.Wrap(err, "failed to create asset")
    }

    // Operation 2
    for _, relation := range input.Relations {
```

```
        relation.AssetID = asset.ID
        if err := tx.CreateRelation(ctx, relation); err != nil {
            tx.Rollback(ctx)
            return errors.Wrap(err, "failed to create relation")
        }
    }

    // Commit
    if err := tx.Commit(ctx); err != nil {
        return errors.Wrap(err, "failed to commit transaction")
    }

    return nil
}
```

4.5 GORM Model Definition

```Go
type Asset struct {
    ID                int64          `gorm:"primaryKey;autoIncrement"`
    Name              string         `gorm:"type:varchar(500);not null"`
    CloudResourceID   string         `gorm:"type:varchar(255);uniqueIndex"`
    Status            string
`gorm:"type:varchar(50);default:'active'"`
    Criticality       string         `gorm:"type:varchar(50)"`
    NetworkInterfaces datatypes.JSON `gorm:"type:json"`
    LastAssessmentDate *time.Time    `gorm:"type:timestamp NULL"`
    DelFlag           bool           `gorm:"default:false"`  // Soft delete
    CreatedAt         time.Time
`gorm:"type:timestamp;default:CURRENT_TIMESTAMP"`
    UpdatedAt         time.Time
`gorm:"type:timestamp;default:CURRENT_TIMESTAMP"`
    CreatedBy         datatypes.JSON `gorm:"type:json"`
    UpdatedBy         datatypes.JSON `gorm:"type:json"`
}

func (Asset) TableName() string {
    return "sigma_assets"
}
```

---

5. Coding Standards

5.1 Error Handling

Always wrap errors with context:

```Go
import errors
"github.com/safe-security-enterprise/safe-go-libraries/utils/errors"

// GOOD
if err != nil {
    return errors.Wrap(err, "failed to get asset from database")
}

// BAD - loses stack trace
if err != nil {
    return fmt.Errorf("failed: %v", err)
}

// BAD - no context
if err != nil {
    return err
}
```

Never ignore errors:

```Go
// GOOD
result, err := doSomething()
if err != nil {
    return nil, errors.Wrap(err, "doSomething failed")
}

// BAD - will fail code review
result, _ := doSomething()
```

No panic in production code:

```go
// GOOD
if value == nil {
    return nil, errors.New("value cannot be nil")
}

// BAD - crashes the service
if value == nil {
    panic("value is nil")
}
```

5.2 Logging

Use structured logging:

```go
logger := appcontext.GetLogger(ctx)

// GOOD - structured with fields
logger.WithField("assetId", id).Info("Asset updated successfully")
logger.WithFields(logrus.Fields{
    "count":    len(assets),
    "duration": elapsed,
}).Info("Batch processing completed")

// GOOD - error with stack trace
logger.WithError(err).Error("Failed to process request")

// BAD - no structure
fmt.Println("Asset updated")
log.Printf("Error: %v", err)
```

Include TAG for traceability:

```go
func (c *controller) GetAssetsHandler(...) {
    TAG := "[GetAssets]"
    logger := appcontext.GetLogger(ctx)

    logger.Info(TAG, "Starting asset retrieval")
    // ...
    logger.WithError(err).Error(TAG, "Failed to get assets")
}
```

5.3 Multi-Tenancy (Critical!)

Every database query MUST include tenant isolation:

```go
// GOOD - tenant isolated
func (ds *DBStore) GetAssets(ctx context.Context) ([]*Asset, error) {
    tenantId := ctx.Value(appcontext.TenantId).(string)

    var assets []*Asset
    err := ds.db.WithContext(ctx).
        Where("tenant_id = ?", tenantId).  // REQUIRED!
        Find(&assets).Error
    return assets, err
}

// BAD - no tenant isolation (SECURITY VULNERABILITY!)
func (ds *DBStore) GetAssets(ctx context.Context) ([]*Asset, error) {
    var assets []*Asset
    err := ds.db.WithContext(ctx).Find(&assets).Error
    return assets, err
}
```

5.4 Constants (No Hardcoding!)

Organize constants by entity:

```go
// constants/asset.go
package constants

type AssetStatus string
const (
    AssetStatusActive   AssetStatus = "active"
    AssetStatusInactive AssetStatus = "inactive"
    AssetStatusDeleted  AssetStatus = "deleted"
)

type AssetCriticality string
const (
    CriticalityCritical   AssetCriticality = "Critical"
    CriticalityHigh       AssetCriticality = "High"
    CriticalityMedium     AssetCriticality = "Medium"
    CriticalityLow        AssetCriticality = "Low"
)
```

```go
// Valid values for validation
var ValidAssetStatuses = []AssetStatus{
    AssetStatusActive,
    AssetStatusInactive,
}
```

Usage:

```go
Go
// GOOD
if asset.Status == constants.AssetStatusActive {
    // ...
}

// BAD - hardcoded string
if asset.Status == "active" {
    // ...
}
```

5.5 Configuration

Use Viper via config package:

```go
Go
// config/config.go
func GetDatabaseHost() string {
    return GetSettings().GetString("DATABASE_HOST")
}

func GetHandlerTimeout() time.Duration {
    return GetSettings().GetDuration("HANDLER_TIMEOUT")
}

// Usage
host := config.GetDatabaseHost()
timeout := config.GetHandlerTimeout()
```

5.6 Security Standards

Rule	Requirement
SQL Injection	Use parameterized queries only
Secrets	Never hardcode; use AWS Secrets Manager
JWT	RS256 signing only (not HS256)
Token Expiration	15-minute maximum
RBAC	Casbin policy validation required
Input Validation	Validate all user inputs

SQL Injection Prevention:

```go
// GOOD - parameterized
db.Where("name = ?", userInput).Find(&results)
db.Exec("UPDATE assets SET name = ? WHERE id = ?", name, id)

// BAD - SQL injection vulnerability!
db.Exec("UPDATE assets SET name = '" + userInput + "' WHERE id = " + id)
db.Where("name = " + userInput).Find(&results)
```

---

6. Software Engineering Best Practices

Writing code that works is just the beginning. Writing code that is **maintainable**, **readable**, and **scalable** is what separates good engineers from great ones. These principles will help you write better code.

6.1 SOLID Principles

SOLID is an acronym for five design principles that make software designs more understandable, flexible, and maintainable.

S - Single Responsibility Principle (SRP)

A struct/function should have only one reason to change.

Every module, struct, or function should do one thing and do it well.

```go
// BAD - AssetService doing too many things
type AssetService struct{}

func (s *AssetService) CreateAsset(ctx context.Context, asset *Asset) error {
    // Validate asset
    if asset.Name == "" {
        return errors.New("name required")
    }
    // Save to database
    db.Create(asset)
    // Send notification email
    sendEmail(asset.Owner, "Asset Created")
    // Update analytics
    analytics.Track("asset_created", asset.ID)
    // Generate report
    generatePDFReport(asset)
    return nil
}

// GOOD - Each service has a single responsibility
type AssetService struct {
    validator   ValidatorInterface
    repo        AssetRepositoryInterface
    notifier    NotifierInterface
}

func (s *AssetService) CreateAsset(ctx context.Context, asset *Asset) error {
    if err := s.validator.Validate(asset); err != nil {
        return err
    }
    if err := s.repo.Create(ctx, asset); err != nil {
        return err
    }
    s.notifier.NotifyAssetCreated(ctx, asset)  // Async, doesn't block
    return nil
}

// Separate services for separate concerns
type AssetValidator struct{}
type AssetRepository struct{}
```

```
type AssetNotifier struct{}
type AnalyticsService struct{}
```

At SAFE: Our three-layer architecture (API → Service → DataStore) enforces SRP. Controllers handle HTTP, Services handle business logic, DataStores handle persistence.

O - Open/Closed Principle (OCP)

> **Software entities should be open for extension but closed for modification.**

You should be able to add new functionality without changing existing code.

```Go
// BAD - Need to modify function every time we add a new asset type
func CalculateRiskScore(asset *Asset) int {
    switch asset.Type {
    case "server":
        return asset.Vulnerabilities * 10
    case "database":
        return asset.Vulnerabilities * 20
    case "application":
        return asset.Vulnerabilities * 15
    // Need to add more cases here for new types...
    default:
        return 0
    }
}

// GOOD - Open for extension via interface
type RiskCalculator interface {
    CalculateRisk(asset *Asset) int
}

type ServerRiskCalculator struct{}
func (c *ServerRiskCalculator) CalculateRisk(asset *Asset) int {
    return asset.Vulnerabilities * 10
}

type DatabaseRiskCalculator struct{}
func (c *DatabaseRiskCalculator) CalculateRisk(asset *Asset) int {
    return asset.Vulnerabilities * 20
```

```go
}

// New types just implement the interface - no modification needed
type ContainerRiskCalculator struct{}
func (c *ContainerRiskCalculator) CalculateRisk(asset *Asset) int {
    return asset.Vulnerabilities * 5
}

// Factory to get the right calculator
func GetRiskCalculator(assetType string) RiskCalculator {
    calculators := map[string]RiskCalculator{
        "server":    &ServerRiskCalculator{},
        "database":  &DatabaseRiskCalculator{},
        "container": &ContainerRiskCalculator{},
    }
    return calculators[assetType]
}
```

L - Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

If you use an interface, any implementation should work without breaking the code.

```go
// Interface defines the contract
type AssetStore interface {
    GetAsset(ctx context.Context, id int64) (*Asset, error)
    SaveAsset(ctx context.Context, asset *Asset) error
}

// MySQL implementation
type MySQLAssetStore struct {
    db *gorm.DB
}

func (s *MySQLAssetStore) GetAsset(ctx context.Context, id int64) (*Asset,
error) {
    var asset Asset
    err := s.db.WithContext(ctx).First(&asset, id).Error
    return &asset, err
}
```

```go
// PostgreSQL implementation - fully substitutable
type PostgreSQLAssetStore struct {
    db *gorm.DB
}

func (s *PostgreSQLAssetStore) GetAsset(ctx context.Context, id int64) (*Asset,
error) {
    var asset Asset
    err := s.db.WithContext(ctx).First(&asset, id).Error
    return &asset, err
}

// In-memory implementation for testing - fully substitutable
type InMemoryAssetStore struct {
    assets map[int64]*Asset
}

func (s *InMemoryAssetStore) GetAsset(ctx context.Context, id int64) (*Asset,
error) {
    if asset, ok := s.assets[id]; ok {
        return asset, nil
    }
    return nil, errors.New("not found")
}

// Service doesn't care which implementation is used
type AssetService struct {
    store AssetStore  // Works with ANY implementation
}
```

At SAFE: This is why we use interfaces everywhere. `AssetServiceInterface`, `DataStore`, `KafkaInterface` - all allow substitution with mocks for testing.

I - Interface Segregation Principle (ISP)

> **Clients should not be forced to depend on interfaces they don't use.**

Keep interfaces small and focused. Many small interfaces are better than one large interface.

Go

```go
// BAD - One large interface forces implementers to implement everything
type AssetManager interface {
    CreateAsset(ctx context.Context, asset *Asset) error
    UpdateAsset(ctx context.Context, asset *Asset) error
    DeleteAsset(ctx context.Context, id int64) error
    GetAsset(ctx context.Context, id int64) (*Asset, error)
    ListAssets(ctx context.Context) ([]*Asset, error)
    ExportAssets(ctx context.Context) ([]byte, error)
    ImportAssets(ctx context.Context, data []byte) error
    CalculateRisk(ctx context.Context, id int64) (int, error)
    SendNotification(ctx context.Context, id int64) error
    GenerateReport(ctx context.Context, id int64) ([]byte, error)
}

// GOOD - Segregated interfaces
type AssetReader interface {
    GetAsset(ctx context.Context, id int64) (*Asset, error)
    ListAssets(ctx context.Context) ([]*Asset, error)
}

type AssetWriter interface {
    CreateAsset(ctx context.Context, asset *Asset) error
    UpdateAsset(ctx context.Context, asset *Asset) error
    DeleteAsset(ctx context.Context, id int64) error
}

type AssetExporter interface {
    ExportAssets(ctx context.Context) ([]byte, error)
    ImportAssets(ctx context.Context, data []byte) error
}

// Compose interfaces when needed
type AssetStore interface {
    AssetReader
    AssetWriter
}

// A read-only service only needs AssetReader
type ReportingService struct {
    assets AssetReader  // Only needs read access
}

// An admin service needs both
type AdminService struct {
    assets AssetStore  // Needs read + write
}
```

At SAFE: Our DataStore interface is composed of smaller interfaces (`AssetStore`, `GroupStore`, `FindingStore`). Services only depend on what they need.

D - Dependency Inversion Principle (DIP)

> **High-level modules should not depend on low-level modules. Both should depend on abstractions.**

Depend on interfaces, not concrete implementations.

```Go
// BAD - Service depends on concrete implementation
type AssetService struct {
    db *gorm.DB  // Concrete dependency
}

func (s *AssetService) GetAsset(ctx context.Context, id int64) (*Asset, error)
{
    var asset Asset
    s.db.First(&asset, id)  // Directly coupled to GORM
    return &asset, nil
}

// GOOD - Service depends on abstraction
type AssetService struct {
    store AssetStoreInterface  // Interface dependency
}

func NewAssetService(store AssetStoreInterface) *AssetService {
    return &AssetService{store: store}
}

func (s *AssetService) GetAsset(ctx context.Context, id int64) (*Asset, error)
{
    return s.store.GetAsset(ctx, id)  // Calls interface method
}

// Now you can inject any implementation
func main() {
    // Production: real database
    realStore := datastore.NewDBStore(db)
    service := NewAssetService(realStore)

    // Testing: mock store
    mockStore := fake.NewMockAssetStore(ctrl)
```

```
    testService := NewAssetService(mockStore)
}
```

At SAFE: We inject dependencies through constructors. The `Clients` struct aggregates all external dependencies for injection.

6.2 DRY - Don't Repeat Yourself

> **Every piece of knowledge must have a single, unambiguous representation in the system.**

Duplication leads to bugs when you update one place but forget another.

```Go
// BAD - Duplicated validation logic
func CreateAsset(asset *Asset) error {
    if asset.Name == "" {
        return errors.New("name is required")
    }
    if len(asset.Name) > 500 {
        return errors.New("name too long")
    }
    // ... create logic
}

func UpdateAsset(asset *Asset) error {
    if asset.Name == "" {
        return errors.New("name is required")  // Duplicated!
    }
    if len(asset.Name) > 500 {
        return errors.New("name too long")     // Duplicated!
    }
    // ... update logic
}

// GOOD - Single source of truth for validation
func ValidateAsset(asset *Asset) error {
    if asset.Name == "" {
        return errors.New("name is required")
    }
    if len(asset.Name) > 500 {
```

```
        return errors.New("name too long")
    }
    return nil
}

func CreateAsset(asset *Asset) error {
    if err := ValidateAsset(asset); err != nil {
        return err
    }
    // ... create logic
}

func UpdateAsset(asset *Asset) error {
    if err := ValidateAsset(asset); err != nil {
        return err
    }
    // ... update logic
}
```

DRY also applies to:

- Constants (use `constants/` package)
- Configuration values (use `config/` package)
- Error messages (define once, reuse)
- Common utilities (use `safe-go-libraries`)

6.3 KISS - Keep It Simple, Stupid

> **Simplicity should be a key goal in design. Avoid unnecessary complexity.**

The simplest solution that works is usually the best.

```
Go
// BAD - Over-engineered for a simple task
type AssetNameFormatterFactory struct {
    formatters map[string]AssetNameFormatter
}

type AssetNameFormatter interface {
    Format(name string) string
```

```
}

type UpperCaseFormatter struct{}
type LowerCaseFormatter struct{}
type TitleCaseFormatter struct{}

func (f *AssetNameFormatterFactory) GetFormatter(style string)
AssetNameFormatter {
    return f.formatters[style]
}

// Just to format a name...
name := factory.GetFormatter("upper").Format(asset.Name)

// GOOD - Simple and clear
func FormatAssetName(name string, style string) string {
    switch style {
    case "upper":
        return strings.ToUpper(name)
    case "lower":
        return strings.ToLower(name)
    case "title":
        return strings.Title(name)
    default:
        return name
    }
}

// Usage
name := FormatAssetName(asset.Name, "upper")
```

Signs of over-engineering:

- Creating abstractions for things that won't change
- Using design patterns just because they exist
- Building for hypothetical future requirements
- More than 3 levels of indirection to understand code flow

6.4 YAGNI - You Aren't Gonna Need It

> **Don't add functionality until you actually need it.**

Build for today's requirements, not tomorrow's speculation.

```go
// BAD - Building features "just in case"
type Asset struct {
    ID              int64
    Name            string
    Type            string
    // "We might need these later..."
    FutureField1    string  // Never used
    FutureField2    string  // Never used
    PluginSupport   bool    // Never implemented
    CustomRenderer  func()  // Never called
}

func (a *Asset) ToJSON() string { ... }
func (a *Asset) ToXML() string { ... }   // Nobody asked for XML
func (a *Asset) ToYAML() string { ... }  // Nobody asked for YAML
func (a *Asset) ToCSV() string { ... }   // Nobody asked for CSV

// GOOD - Build what you need now
type Asset struct {
    ID   int64
    Name string
    Type string
}

func (a *Asset) ToJSON() string { ... }  // The only format we actually need
```

YAGNI in practice:

- Don't create configuration options for things that are unlikely to change
- Don't build plugin systems until you have plugins
- Don't create abstractions until you have at least 2-3 concrete uses
- Delete commented-out code - version control has history

6.5 Composition Over Inheritance

> **Favor object composition over class inheritance.**

Go doesn't have inheritance, but the principle still applies: build complex types by combining simple ones.

```go
// Using embedding for composition
type Timestamps struct {
    CreatedAt time.Time
    UpdatedAt time.Time
}

type SoftDelete struct {
    DeletedAt *time.Time
    DelFlag   bool
}

type AuditInfo struct {
    CreatedBy string
    UpdatedBy string
}

// Compose models from smaller pieces
type Asset struct {
    ID   int64
    Name string
    Type string

    Timestamps            // Embedded - Asset "has" timestamps
    SoftDelete            // Embedded - Asset "has" soft delete
    AuditInfo             // Embedded - Asset "has" audit info
}

// Usage - fields are promoted
asset := &Asset{
    Name:      "Server-01",
    CreatedAt: time.Now(),  // From Timestamps
    CreatedBy: "user-123",  // From AuditInfo
}
```

Compose services similarly:

```go
type AssetService struct {
    validator  *AssetValidator
    repository *AssetRepository
    notifier   *Notifier
    logger     *Logger
}
```

```
// Each component is focused and reusable
```

6.6 Fail Fast

> **If something is going to fail, it should fail immediately and visibly.**

Don't let errors propagate silently. Catch them early.

```Go
// BAD - Silent failures, errors discovered much later
func ProcessAssets(assets []*Asset) []*Result {
    var results []*Result
    for _, asset := range assets {
        result, err := processOne(asset)
        if err != nil {
            continue  // Silent skip - debugging nightmare!
        }
        results = append(results, result)
    }
    return results  // No idea if anything failed
}

// GOOD - Fail fast, fail loudly
func ProcessAssets(assets []*Asset) ([]*Result, error) {
    if len(assets) == 0 {
        return nil, errors.New("no assets provided")  // Fail fast on invalid
input
    }

    var results []*Result
    var errs []error

    for _, asset := range assets {
        result, err := processOne(asset)
        if err != nil {
            errs = append(errs, errors.Wrapf(err, "failed to process asset %d",
asset.ID))
            continue
        }
        results = append(results, result)
    }
```

```
    if len(errs) > 0 {
        return results, fmt.Errorf("processed %d/%d assets, errors: %v",
            len(results), len(assets), errs)
    }

    return results, nil
}
```

Fail fast patterns:

- Validate inputs at function entry
- Check preconditions before doing work
- Return errors immediately, don't queue them
- Use timeouts to fail fast on slow operations

6.7 Separation of Concerns

> **Different concerns should be handled in different places.**

Keep things that change for different reasons separate.

```
Go
// BAD - Everything mixed together
func CreateAssetHandler(w http.ResponseWriter, r *http.Request) {
    // HTTP parsing
    body, _ := ioutil.ReadAll(r.Body)
    var asset Asset
    json.Unmarshal(body, &asset)

    // Validation
    if asset.Name == "" {
        http.Error(w, "name required", 400)
        return
    }

    // Business logic
    asset.Status = "active"
    asset.CreatedAt = time.Now()

    // Database
```

```
    db.Create(&asset)

    // Response
    json.NewEncoder(w).Encode(asset)
}

// GOOD - Each concern in its own place
// Handler - HTTP concerns only
func (c *controller) CreateAssetHandler(w http.ResponseWriter, r *http.Request)
(*Response, error) {
    input, err := helpers.ParseCreateAssetRequest(r)  // HTTP parsing
    if err != nil {
        return nil, err
    }

    asset, err := c.assetService.CreateAsset(r.Context(), input)  // Delegates
to service
    if err != nil {
        return nil, err
    }

    return &Response{StatusCode: 201, Data: asset}, nil
}

// Service - Business logic only
func (s *AssetService) CreateAsset(ctx context.Context, input *CreateInput)
(*Asset, error) {
    if err := s.validator.Validate(input); err != nil {  // Validation
        return nil, err
    }

    asset := &Asset{
        Name:      input.Name,
        Status:    constants.AssetStatusActive,
        CreatedAt: time.Now(),
    }

    return s.repo.Create(ctx, asset)  // Delegates to repository
}

// Repository - Persistence only
func (r *AssetRepository) Create(ctx context.Context, asset *Asset) (*Asset,
error) {
    if err := r.db.WithContext(ctx).Create(asset).Error; err != nil {
        return nil, errors.Wrap(err, "failed to create asset")
    }
    return asset, nil
```

```
}
```

6.8 Clean Code Principles

Meaningful Names

```Go
// BAD
func calc(a []*Asset) int {
    var t int
    for _, x := range a {
        t += x.V
    }
    return t
}

// GOOD
func CalculateTotalVulnerabilities(assets []*Asset) int {
    var totalVulnerabilities int
    for _, asset := range assets {
        totalVulnerabilities += asset.VulnerabilityCount
    }
    return totalVulnerabilities
}
```

Functions Should Do One Thing

```Go
// BAD - Function does multiple things
func ProcessAsset(asset *Asset) error {
    // Validates
    // Transforms
    // Saves to DB
    // Sends notification
    // Updates cache
    // Logs metrics
    return nil
}

// GOOD - One function, one job
```

```go
func ProcessAsset(asset *Asset) error {
    if err := validateAsset(asset); err != nil {
        return err
    }
    transformedAsset := transformAsset(asset)
    if err := saveAsset(transformedAsset); err != nil {
        return err
    }
    notifyAssetCreated(transformedAsset)
    return nil
}
```

Small Functions (< 20 lines ideal)

```go
// BAD - 100+ line function
func HandleAssetRequest(r *http.Request) (*Response, error) {
    // 100+ lines of code doing everything
}

// GOOD - Small, focused functions
func HandleAssetRequest(r *http.Request) (*Response, error) {
    input, err := parseRequest(r)
    if err != nil {
        return nil, err
    }
    return processAsset(input)
}
```

Comments: Explain Why, Not What

```go
// BAD - Comment explains what (code already shows that)
// Loop through assets
for _, asset := range assets {
    // Check if asset is active
    if asset.Status == "active" {
        // Add to result
        result = append(result, asset)
    }
}

// GOOD - Comment explains why
```

```
// Filter out inactive assets because they shouldn't appear in risk
calculations
// per compliance requirement CR-2024-001
for _, asset := range assets {
    if asset.Status == "active" {
        result = append(result, asset)
    }
}

// BETTER - Self-documenting code needs no comment
activeAssets := filterActiveAssets(assets)
```

6.9 Error Handling Best Practices

```Go
// 1. Always handle errors
result, err := doSomething()
if err != nil {
    return errors.Wrap(err, "context about what failed")
}

// 2. Wrap errors with context
if err := db.Create(&asset).Error; err != nil {
    return errors.Wrap(err, "failed to create asset in database")
}

// 3. Use custom error types for different handling
type NotFoundError struct {
    Resource string
    ID       int64
}

func (e *NotFoundError) Error() string {
    return fmt.Sprintf("%s with ID %d not found", e.Resource, e.ID)
}

// 4. Check specific error types
if errors.Is(err, gorm.ErrRecordNotFound) {
    return &NotFoundError{Resource: "asset", ID: id}
}

// 5. Don't ignore errors with _
```

```
result, _ := doSomething()  // BAD - never do this
```

6.10 Summary: The Good Code Checklist

Before submitting code, ask yourself:

Question	**Principle**
Does each function/struct do ONE thing?	SRP
Can I add features without modifying existing code?	OCP
Do my implementations honor their interface contracts?	LSP
Are my interfaces small and focused?	ISP
Am I depending on abstractions, not concretions?	DIP
Is there any duplicated logic?	DRY
Is this the simplest solution that works?	KISS
Am I building something actually needed now?	YAGNI
Do errors fail fast and visibly?	Fail Fast
Are different concerns in different places?	Separation of Concerns
Are names meaningful and self-documenting?	Clean Code
Are functions small (< 20 lines)?	Clean Code
Are all errors handled and wrapped with context?	Error Handling

---

7. Testing

7.1 Test Organization

```
None
asset.go          → asset_test.go         (same directory)
asset_service.go  → asset_service_test.go (same directory)
```

7.2 Unit Test with Mocks

```go
Go
func TestAssetService_GetAsset_Success(t *testing.T) {
    // Setup
    mockCtrl := gomock.NewController(t)
    defer mockCtrl.Finish()

    mockDataStore := fakeDataStore.NewMockDataStore(mockCtrl)

    // Set expectations
    expectedAsset := &model.Asset{ID: 1, Name: "Test Asset"}
    mockDataStore.EXPECT().
        GetAsset(gomock.Any(), int64(1)).
        Return(expectedAsset, nil)

    // Create service with mock
    service := &AssetService{}

    // Create context with mock datastore
    ctx := context.WithValue(context.Background(), appcontext.Datastore,
mockDataStore)

    // Execute
    result, err := service.GetAsset(ctx, 1)

    // Assert
    assert.NoError(t, err)
    assert.Equal(t, expectedAsset.Name, result.Name)
}
```

7.3 Handler Test

```Go
func TestController_GetAssetsHandler_Success(t *testing.T) {
    mockCtrl := gomock.NewController(t)
    defer mockCtrl.Finish()

    mockAssetService := fakeServices.NewMockAssetServiceInterface(mockCtrl)
    mockAssetService.EXPECT().
        GetAssets(gomock.Any(), gomock.Any()).
        Return(&types.AssetListResult{Count: 1}, nil)

    ctrl := &controller{assetService: mockAssetService}

    test := testUtil.NewHandlerTest()
    test.Route = "/internal/v1/assets?page=1"
    test.Method = http.MethodGet
    test.ContextMap = map[any]any{
        appcontext.TenantId: "tenant-123",
    }
    test.Handler = ResponseHandler(ctrl.GetAssetsHandler)

    resp := test.ExecuteRequest(t)

    assert.Equal(t, http.StatusOK, resp.StatusCode)
}
```

7.4 Generate Mocks

```Shell
# Generate all mocks
go generate ./...

# Mocks are generated based on //go:generate directives in interfaces
```

7.5 Coverage Requirements

Metric	**Threshold**
Functions	83%
Branches	88%
Statements	90%

```shell
# Run tests with coverage
make test-coverage
# or
task test-coverage

# View coverage report
go tool cover -html=coverage.out
```

---

8. Local Development

8.1 Prerequisites

```shell
# Install Go 1.24+
brew install go

# Install tools
go install github.com/golang/mock/mockgen@v1.6.0
go install github.com/pressly/goose/v3/cmd/goose@latest
brew install golangci-lint

# Set private module access
export GITHUB_TOKEN=your_token
export GOPRIVATE=github.com/safe-security-enterprise/*
```

8.2 First-Time Setup

```shell
# Clone and enter service directory
cd sigma-service

# Copy environment config
cp .env.sample .env
# Edit .env with your local settings

# Start dependencies
docker-compose -f docker-compose.dev.yml up -d
```

```
# Run migrations
go run cmd/main.go --migrate --migration-command=up

# Start service
go run cmd/main.go --api --port=9123

# Test
curl http://localhost:9123/healthcheck
```

8.3 Common Commands

Shell

```
# Build
make build        # or: task build

# Run
make run          # or: task run

# Test
make test         # or: task test
go test -v -run TestGetAsset ./core/services  # Single test

# Lint
make lint         # or: task lint

# Format
go fmt ./...

# Generate mocks
go generate ./...
```

8.4 Database Migrations

Shell

```
# Apply all pending migrations
go run cmd/main.go --migrate --migration-command=up

# Rollback last migration
go run cmd/main.go --migrate --migration-command=down

# Migration file naming: YYYYMMDDHHMSS_description.go
# Example: 20241007154124_create_assets_table.go
```

---

9. Quick Reference

9.1 Creating a New Endpoint Checklist

1. **Add Route** (`api/v1/router.go`)

```Go
{Path: "/internal/v1/things", Handler: ctrl.GetThingsHandler, ...}
```

2. **Create Handler** (`api/v1/controllers/`)
    - Parse request → Call service → Format response
3. **Add Service Method** (`core/services/`)
    - Define interface method
    - Implement business logic
    - Call datastore
4. **Add DataStore Method** (`datastore/`)
    - Define interface method
    - Implement GORM query
5. **Write Tests**
    - Handler test with mock service
    - Service test with mock datastore
6. **Verify**

```Shell
go run cmd/main.go --api --port=9123
curl http://localhost:9123/internal/v1/things
```

9.2 Do's and Don'ts

DO	DON'T
Use `errors.Wrap()` for all errors	Use `fmt.Errorf()`
Use `appcontext.GetLogger(ctx)`	Use `fmt.Println` or `log.Printf`
Include `tenantId` in all queries	Write queries without tenant context
Use constants from `constants/`	Hardcode strings
Use parameterized queries	Concatenate SQL strings
Write tests for new code	Skip tests
Use interfaces for dependencies	Use concrete types
Return early on errors	Nest error handling

9.3 Service Ports

Service	Port
sigma-service	9123
iris-service	4005
grip-service	7863
ce-service	8080

9.4 Useful Links

- **Code Standards**: `/CODEANT_GUIDELINES.md`
- **Shared Libraries**: `/safe-go-libraries/`
- **Templates**: `/ninja-cookbook/`

Questions?

Reach out to your team lead or check the service-specific documentation in each repository's `CLAUDE.md` or `.cursor/rules/` directory.