

고언어 프로젝트 가이드 A-Z

피쳐 단위 개발에서 엔터프라이즈
어플리케이션 설계까지의 고민

변규현 / Software engineer at Karrot.



Speaker

변규현

Karrot / Chatting.



“당근에서 채팅 조직의 리드를 맡고 있어요. 지난 5년간 고언어로 개발하고 있어요. 또한 **AWS Serverless Hero**로 클라우드 컴퓨팅과 관련하여 다양한 분들에게 지식을 공유하기 위해 노력하고 있습니다 :)”




Index

- 작은 프로젝트와 큰 프로젝트의 차이
- 작은 프로젝트를 위한 코드 패턴
- 기능단위의 프로젝트 코드 패턴
- 엔터프라이즈 어플리케이션을 위한 코드 패턴
- 앱을 릴리즈하기 위해 꼭 붙이는 도구



고언어로 어떻게 개발해요?



많이 들었던 질문이에요.(코드패턴, 아키텍처 설계, 개발 방법론 등)
오늘은 이 질문에 대해서 답변하기 위한 과정이에요.






작은 프로젝트와 큰 프로젝트의 차이



여기서 말하는 작은 프로젝트란?



트래픽이 적다.

기능이 단순하다.

빠른 개발이 필요하다.

이런 프로젝트의 대표적인 형태는 무엇이 있을까요?



작은 프로젝트

- 사용자 풀이 확보되지 않은 상태
- 신규 서비스로 초기 유저 상태가 정의되지 않은 상태
- 신규기능이지만 실험으로 소수의 유저를 통해 가설 검증이 필요한 상태



그러나 우리가 가지고 있는 지식은?

Clean Architecture

Microservices Architecture

Event-Driven Architecture (EDA)

Command Query Responsibility Segregation (CQRS)

Behavior-Driven Development (BDD)

Test-Driven Development (TDD)

Hexagonal architecture

Domain Driven Design



시도한다면 ...



작은 유니콘을 만들어보자!




단순하게 생각하기

필요한 것에 집중하기

그리고 오늘의 주제로 돌아와서 고언어로 구현하기



어떻게 코드를 작성할 수 있을까?



만들고 싶은 명확한 목표를 둔다.
거기에 맞는 기능만 구현한다.



예시 1) 간단한 CLI를 작성하는 경우

간단한 CLI를 만들 때, CLI를 지원하는 라이브러리나 코드 패턴을 찾는 경우도 있어요.
빠르게 돌려야하는 로직을 작성할 때, 크게 고민하지 않고 이런 식으로 작성하면
어떨까요?



```
.
├── README.md
├── cmd
│   └── hello.go
├── go.mod
└── internal
    ├── hello
    │   └── hello.go
    └── hi
        └── hi.go
```

```
import (
    "os"
    "simple-cli-application/internal/hello"
    "simple-cli-application/internal/hi"
)

func main() {
    if len(os.Args) <= 1 {
        panic("Please provide a command")
    }

    switch os.Args[1] {
    case "hello":
        hello.SayHello()
    case "hi":
        hi.SayHi()
    }
}
```



예시 2) 간단한 **API**서버를 만드는 경우

단순하게 접근해요. 기술을 찾고, 적용하는 걸 하고 싶지만 기본 라이브러리를 활용해요.
API를 연동하고 작게 붙여나가요.



```
.
├─ README.md
├─ go.mod
├─ internal
│   └─ server
│       └─ server.go
└─ main.go
```

```
func main() {
    // New server
    srv := server.New(&server.Config{
        Port: "8080",
    })
    // Start server
    go func() {
        log.Println("Starting server on port 8080")
        err := srv.Start()
        if err == nil {
            log.Println("Server started")
        } else if err == http.ErrServerClosed {
            log.Println("Server closed")
        } else {
            log.Panic(err)
        }
    }()
    // Graceful shutdown
    ctx, stop := signal.NotifyContext(context.Background(), syscall.SIGINT, syscall.SIGTERM)
    defer stop()
    // Wait for signal
    <-ctx.Done()
    // Stop server
    if err := srv.Stop(); err != nil {
        panic(err)
    }
}
```



```
.
├─ README.md
├─ go.mod
├─ internal
│   └─ server
│       └─ server.go
└─ main.go
```

```
type Server struct {
    httpServer *http.Server
}

func New(c *Config) *Server {
    httpServer := &http.Server{
        Addr: ":" + c.Port,

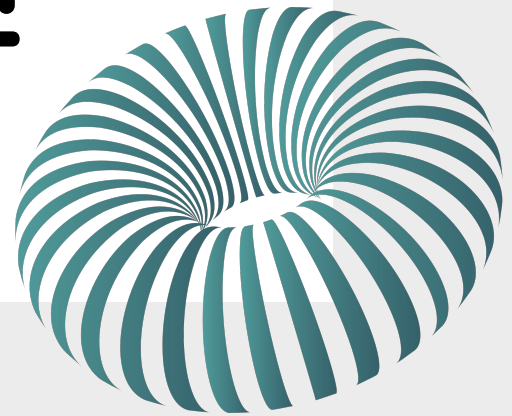
        httpServer.Handler = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if r.URL.Path == "/hello" {
                w.Write([]byte("Hello, World!"))
            } else {
                http.NotFound(w, r)
            }
        })
    }
    return &Server{
        httpServer: httpServer,
    }
}

func (s *Server) Start() error {
    return s.httpServer.ListenAndServe()
}

func (s *Server) Stop() error {
    return s.httpServer.Close()
}
```



기능단위의 프로젝트 코드 패턴



만약 Struct를 활용한다면 ...

```
type HTTPHandler struct {}  
func (h *HTTPHandler) handleHello(w http.ResponseWriter, r  
*http.Request) {  
    // Do Something  
    w.WriteHeader(http.StatusOK)  
    w.Write("Hello")  
}
```



```

type Handler struct {
    cfg config.Config

    aClient      aservice.Client
    bClient      bservice.Client
    // more 10 clients

    chatSvc      chat.Service
    countSvc     count.Service
    // More 6 services

    chatUsecase  usecase.ChatUsecase
    countUsecase usecase.CountUsecase
    // More 16 usecases

    // More 22 fields
}

func New(
    // dependencies...
) *Handler {
    return &Handler{
        // bind dependencies
    }
}

```



그러나 하나의 메서드에서 필요한 건?

- 보통 한 두개의 필드만 필요함. 많다면 5개정도가 필요함
- 하나의 **method**는 수많은 의존성을 지니고 있지 않아도 괜찮아요.
- 기능단위의 서비스는 **method**에서 몇개의 필드만 필요함



```
func (h *Handler) GetArticle(ctx context.Context, req *GetArticleRequest)
(*GetArticleResponse, error) {
    // Check cache
    cachedArticle, err := h.cache.Get(ctx, req.GetId())
    if err != nil {
        return nil, err
    }

    article, err := h.articleDB.Get(ctx, req.GetId())
    if err != nil {
        return nil, err
    }

    h.cache.Save(ctx, article)
    return &GetArticleResponse{
        Article: article,
    }, nil
}
```



HTTP Handlerfunc 예시

```
type Handler interface {ServeHTTP(ResponseWriter, *Request)}  
type HandlerFunc func(ResponseWriter, *Request)  
  
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {  
    f(w, r)  
}
```



```

// Package path: internal/handler
func GetArticle(cache *Cache, articleDB *ArticleDB) func(ctx context.Context, req *GetArticleRequest)
(*GetArticleResponse, error) {
    return func(ctx context.Context, req *GetArticleRequest) (*GetArticleResponse, error){
        cachedArticle, err := cache.Get(ctx, req.GetId())
        if err != nil {
            return nil, err
        }

        article, err := articleDB.Get(ctx, req.GetId())
        if err != nil {
            return nil, err
        }

        cache.Save(ctx, article)
        return &GetArticleResponse{
            Article: article,
        }, nil
    }
}

```



// Handler 방식

```
func New(c *Config) *Handler {
    remoteCache := cache.New(),
    articleDB := articledb.New(),
    return &Server{
        articleHandler: handler.NewArticleHandler(
            remoteCache,
            articleDB,
            // more dependencies.
        )
    }
}

func (s *Server) GetArticle(ctx context.Context, req
*GetArticleRequest) (*GetArticleResponse, error) {
    return s.articleHandler.GetArticle(ctx, req)
}
```

// HandlerFunc 방식

```
func New(c *Config) *Handler {
    return &Server{
        cache: *cache.New(),
        articleDB: *articledb.New(),
    }
}

func (s *Server) GetArticle(ctx context.Context, req
*GetArticleRequest) (*GetArticleResponse, error) {
    return handler.GetArticle(s.remoteCache,
s.articleDB)(ctx, req)
}
```



Handler / HandlerFunc

Handler 방식의 장점

- 복잡한 상태를 관리하기 쉬움
- 의존성과 상태가 구조체 필드로 명시적으로 표현되어 명확한 구조가 됨
- 확장성 측면에서 관련 메서드를 쉽게 추가할 수 있음.
- 다른 인터페이스를 쉽게 구현할 수 있음.
- 복잡한 로직을 구조화하기 좋음.



Handler / HandlerFunc

HandlerFunc 방식의 장점

- 간단한 핸들러를 빠르게 작성 가능
- 의존성 주입이 쉽고 클로저를 활용할 수 있음
- 함수형 프로그래밍 가능. 함수 조합과 고차 함수 활용이 용이.
- 빠른 프로토타이핑 가능.
- 작은 애플리케이션이나 마이크로서비스에 적합.
- 단순한 함수는 의존성 모킹이 쉬워 테스트 코드 작성이 용이함.



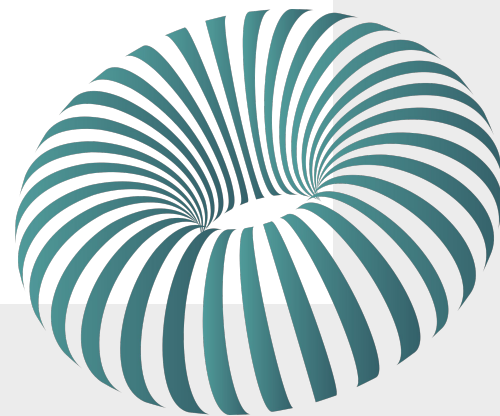
Handler / HandlerFunc

선택 기준은 어떻게 가져갈까요?

- 간단한 핸들러, 빠른 개발이 필요한 경우 **HandlerFunc** 방식이 유리함.
- 복잡한 비즈니스 로직, 많은 의존성, 상태 관리가 필요한 경우 **Handler** 방식이 유리함.



엔터프라이즈 서비스를 위한 코드 패턴



```
.
├── cmd
├── internal
│   ├── recorder
│   ├── domain
│   ├── model
│   ├── handler
│   ├── presenter(internal, event, public)
│   ├── repository
│   ├── service
│   └── usecase
└── ...
```



Domain Driven Design으로 작성하는 레이어의 분리

- 기능 위주의 로직을 비즈니스 위주의 로직으로 분리
- 변경사항 발생 시, 해당 부분의 로직을 중점적으로 개선할 수 있음
- **Bounded Context**로 특정 도메인 모델이 적용되는 명확한 경계를 구분
- 각 서비스는 자신의 도메인에 가장 적합한 기술 스택을 선택할 수 있음
- 점진적 업그레이드와 리팩토링이 가능함





API Model

Presenter

Handler

Domain

Usecase

•

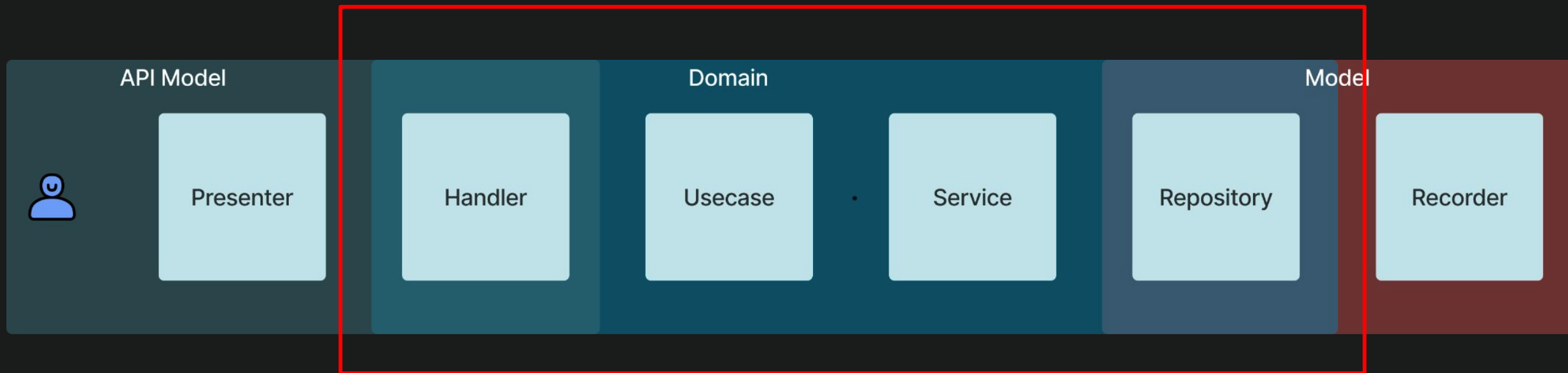
Service

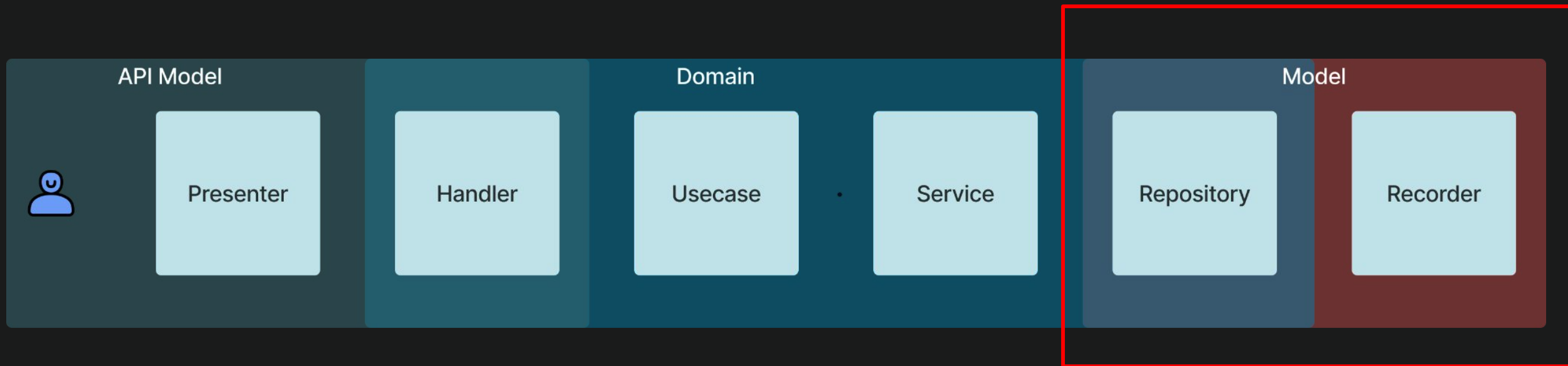
Model

Repository

Recorder





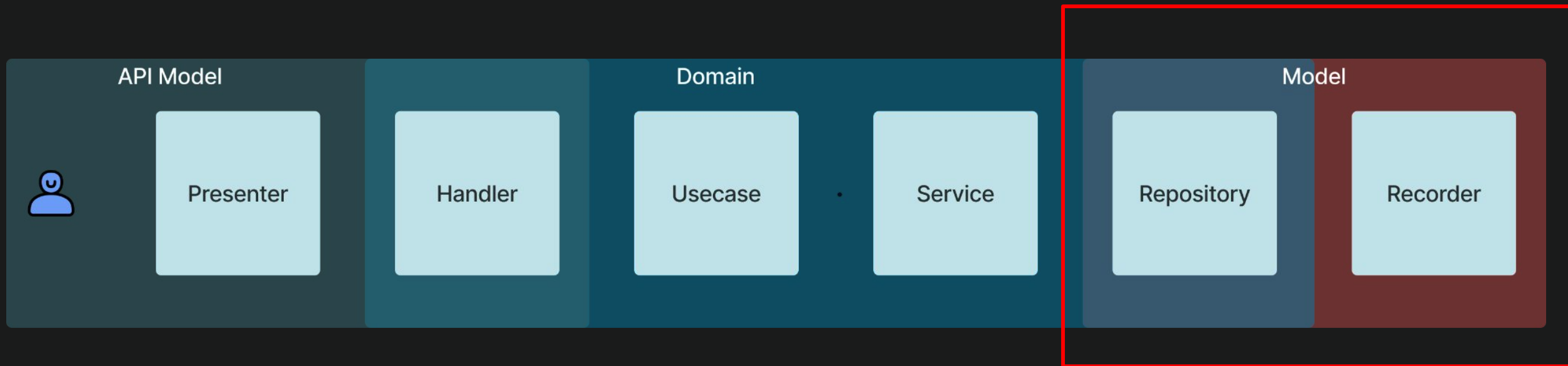


NoSQL을 활용할 때의 모델링

DynamoDB

- Partition Key Name
 - Attribute Name: pk
 - value: 채팅방 ID
- Sort Key
 - Attribute Name: sk
 - value: AA#CreatedAt





각 레이어의 역할

- Presenter: Domain Model to API Model
- Handler: API Model Serving
- Usecase: Business Logic Handling
- (Service): Business Logic Handling
- Repository: CRUD for Primitive Domain Model
- (Recorder): Database Model Handling



Presenter

- 다양한 클라이언트에 대해 서로 다른 **Presenter**를 만들 수 있음.
(예: **admin**, **web**, **mobile** 등)
- **API** 버전 관리가 용이함.
- 민감한 정보를 클라이언트에 노출하지 않도록 제어하는 데 용이함.
- 도메인 모델이 프레젠테이션 계층에 의존하지 않게 됨.
- 필요한 경우 번역이나 로케일에 따른 데이터 변환 용이



```
// Presenter
```

```
func PresentAnnouncement(a
*domain.Announcement) *APIAnnouncement {
    if a == nil {
        return nil
    }

    return &APIAnnouncement{
        Id:          a.ID,
        ChatID:      a.ChatID,
        Content:     a.Content,
        CreatedAt:   a.CreatedAt,
        CreateBy:    createBy,
    }
}
```

```
// Domain Model
```

```
type Announcement struct {
    ChatID      string
    ID          int
    CreateBy    int
    Content     string
    MessageID   *int
    CreatedAt   time.Time
    DeletedAt   *time.Time
}
```



Handler

- **Handler**는 HTTP/gRPC 요청 처리와 응답 생성에만 집중함
- 비즈니스 로직은 **usecase**에 위임되어 있음
- 프레젠테이션 계층(**Handler**)과 애플리케이션 계층(**Usecase**)이 명확히 구분됨
- **Usecase**를 **Mocking**하여 **Handler**를 독립적으로 테스트하기 쉬움



```
func (h *Handler) GetAnnouncement(
    ctx context.Context,
    req *v1.GetAnnouncementRequest,
) (*v1.CreateAnnouncementResponse, error) {
    an, err := h.announcementUsecase.GetAnnouncement(ctx, req.ChatID)
    if err != nil {
        return nil, err
    }

    return &v1.CreateAnnouncementResponse{
        Announcement: presenter.PresentAnnouncement(an),
    }, nil
}
```



Usecase

- **Service** 또는 **Repository** 를 의존성으로 받아 사용함 느슨한 결합을 유지함
- 순수한 도메인 객체를 사용함.
- 단일 메서드에 여러 **service** 또는 **repository**를 조합하여 복잡한 비즈니스 로직을 구현
- **Repository** 인터페이스를 모킹하여 **Usecase** 테스트 코드 작성에 용이



```
func (u announcementUsecase) CreateAnnouncement(
    ctx context.Context,
    chatID string, userID int, content string) (*domain.Announcement, error) {
    if !validateContent(content) {
        return nil, errors.New("invalid content")
    }

    writer, err := u.memberRepo.GetMember(ctx, userID)
    if err != nil {
        return nil, err
    }

    if err := u.checkPermission(writer); err != nil {
        return nil, err
    }

    anno, err := u.announcementRepo.CreateAnnouncement(ctx, chatID, user, content)
    if err != nil {
        return nil, err
    }

    go u.notiService.notifyAnnouncementToAllMembers(chatID)
    return anno, nil
}
```



Service(Optional)

- **Service** 계층으로 분리함으로써 **Usecase**의 복잡성을 줄일 수 있음
- 여러 **Usecase**에서 공통으로 사용되는 복잡한 로직을 **Service**로 분리하여 코드 재사용성을 높임
- **Usecase**는 흐름 제어와 조정에 집중하고, **Service**는 구체적인 비즈니스 로직 구현에 집중
- 필요한 경우에만 **Service**를 도입함으로써 불필요한 추상화를 피하고, 시스템의 유연성을 유지
- 복잡한 로직을 독립적인 **Service**로 분리하여 단위 테스트에 용이해짐



Repository

- 비즈니스 로직과 데이터 접근에서 **Repository**는 데이터 접근 로직만을 담당
- 상위 계층(**Usecase** 등)이 구체적인 데이터 저장 방식을 알 필요가 없음
- 도메인 모델을 반환함으로써, **Repository**가 도메인 계층에 의존하게 되어 의존성 역전 원칙을 따름
- 도메인 로직이 인프라 세부사항으로부터 독립적이게 함
- **Repository**를 쉽게 모킹하여 테스트가 용이함



```
func (r repository) CreateAnnouncement(
    ctx context.Context,
    chatID string, userID int, content string,
) (*domain.Announcement, error) {
    anno, err := r.recorder.CreateAnnouncement(ctx, chatID, userID, content)
    if err != nil {
        return nil, err
    }
    return toDomainModel(anno), nil
}
```



Recorder(Optional)

- DynamoDB의 특정 API나 쿼리 언어를 추상화
- 상위 계층(Repository, Usecase 등)은 DynamoDB의 세부사항을 알 필요가 없음
- 데이터 모델(Key-Value 또는 Document 모델) 로직을 중앙화
- DynamoDB의 특성(예: 파티션 키, 정렬 키 등)을 고려한 최적화된 쿼리를 구현
- 추후 다른 데이터베이스로 마이그레이션해야 할 경우, recorder 레이어만 수정하면 됨



```
func (r *Recorder) GetAnnouncement(  
    ctx context.Context,  
    chatID string,  
) (*model.Announcement, error) {  
    var announcement model.Announcement  
    err := r.AnnouncementTable.Get("chat_id", chatID).  
        OneWithContext(ctx, &announcement)  
    if err != nil {  
        return nil, err  
    }  
    return &announcement, err  
}
```





API Model

Presenter

Handler

Domain

Usecase

•

Service

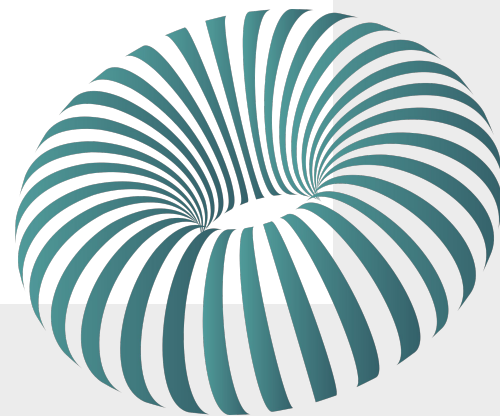
Model

Repository

Recorder



각 레이어에서의 테스트 코드



Fake Implementations

counterfeiter

- <https://github.com/maxbrunsfeld/counterfeiter>
- 복잡한 인터페이스에 대해서도 빠르게 모킹 객체를 생성
- 자동 생성된 모의 객체는 일관된 구조와 네이밍 규칙을 따름(**Fake prefix**)
- 인터페이스의 변경사항을 자동으로 반영할 수 있어, 모의 객체가 항상 최신 상태를 유지
- 메서드 호출 횟수, 반환값, 에러 등을 쉽게 조작 가능




```
package foo
```

```
//go:generate go run github.com/maxbrunsfeld/counterfeiter/v6 .
```

```
MySpecialInterface
```

```
type MySpecialInterface interface {  
    DoThings(string, uint64) (int, error)  
}
```



```
$ go generate ./...
```

```
Writing `FakeMySpecialInterface` to
```

```
`foofakes/fake_my_special_interface.go`... Done
```



```

tests := []struct {
...
    }{
        {
            name: "test",
            fields: fields{
                mySepcialInterface: &foofakes.FakeMySpecialInterface{
                    DoThingsStub: func(s string, u uint64) (int, error) {
                        return 42, nil
                    },
                },
            },
            want: 42,
            wantErr: false,
        },
    }
    for _, tt := range tests {
        tt.Run(tt.name, func(t *testing.T) {
            f := &Foo{
                mySepcialInterface: tt.fields.mySepcialInterface,
            }
            got, err := f.Do()
            . . .
        })
    }
}

```



```
.
├─ cmd
└─ internal
    ├─ recorder
    │   └─ recorderfakes
    ├─ domain -> domain model 및 repository interface가 정의된 곳
    │   └─ domainfakes
    ├─ handler
    ├─ model
    ├─ presenter
    ├─ repository
    ├─ service
    │   └─ xxx
    │       └─ xxxfakes
    └─ usecase
        └─ usecasefakes
```



앱을 릴리즈하기 위해 꼭 붙이는 도구



앱을 릴리즈하기 위해 꼭 붙이는 도구

- APM -> Datadog
- Error monitoring -> Sentry or Datadog
- Metric Collection -> Prometheus
- Log service

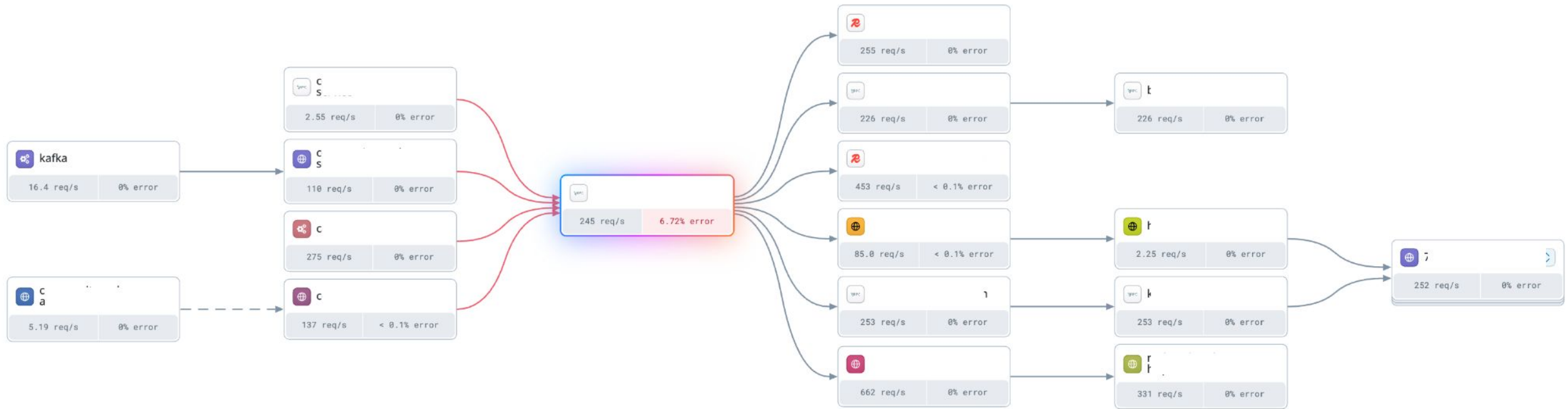


Application Performance Monitoring

- 애플리케이션, 인프라, 네트워크 전반에 걸친 통합된 가시성을 확보할 수 있음
- 성능 문제를 즉시 감지하고 대응 가능함
- 마이크로서비스 아키텍처에서 전체 경로 추적에 용이



Application Performance Monitoring



Application Performance Monitoring

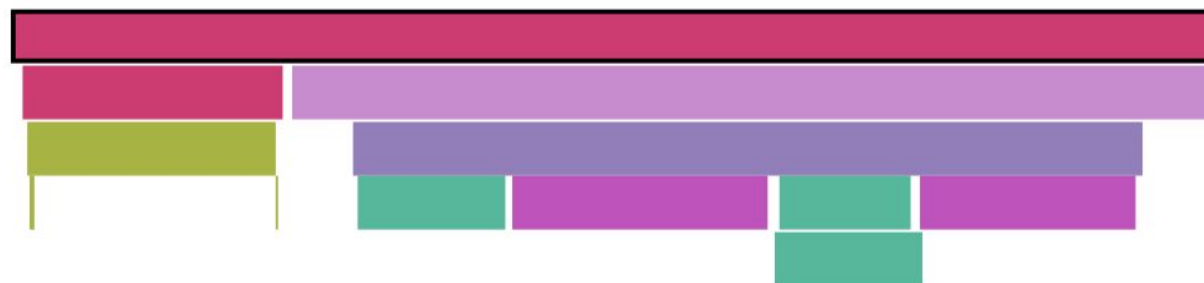
🕒 21.0 ms | Oct 03 19:59:05.738 (10m ago)

Trace: Flame Graph Waterfall Span List 21 Map

🔍 Search...

Color by Service ▾ | [Hide Legend](#) 🗲

0 5 ms 10 ms 15 ms 20 ms



% Exec Time ▾

k	.	49.6%	<div></div>
k	..	23.3%	<div></div>
r	..	14.9%	<div></div>
r	..	7.87%	<div></div>
k		2.18%	<div></div>
r		1.49%	<div></div>
r	..	0.66%	<div></div>

Error monitoring

- 실시간으로 에러를 감지하여 큰 문제로 확대되기 전에 대응
- 에러로 인한 서비스 중단이나 성능 저하를 최소화
- 비정상적인 접근 시도나 취약점 악용을 조기에 발견
- ...



Tags ⓘ

app

os

android

version

24.39.0

environment

prod

level

error

os

name

linux

release

v24.40.2

runtime

go

name

go

server_name

c59f-mbxc5

user

All

Custom

Application

Client

Other

Contexts ⓘ

User

ID

Operating System

Name

linux

Device

Architecture

num_cpu

Runtime

go_maxprocs

4

go_numcgocalls

3

go_numroutines

19195

Name

go

Version

All Tags ⓘ

release

v24.40.2

75%

v24.40.2

75%

v24.40.1

22%

v24.40.3

2%

environment

prod

100%

level

error

100%

os.name

100%

runtime

go

97%

runtime.name

go

100%

server_name

rocket-chat-consumer-d...

17%

er-d57d89bb9-2mdkk

17%

er-d57d89bb9-szd4k

14%

er-d57d89bb9-mkt44

12%

er-d57d89bb9-7nmv2

10%

Other

45%

Prometheus & Log Service 연동

- Prometheus Metric을 통해 인프라 단위 스케일링에 대해서 대응 및 장애 지점 파악에 편리
- 로그서비스를 연동함으로써 원자적으로 **API** 호출에 대한 디버깅이 가능
- ...



오늘의 내용 요약

고언어로 작은
서비스부터
엔터프라이즈
서비스까지
모두 운영할 수 있다.



당근에서 함께 문제를 해결해나가고 성장할 동료들을 찾고
있어요! 🥕



about.daangn.com/jobs/



Thank you!

