


# Services in Rust with Dependency Injection



Ivan Luchev

# About me

-  in/luchev
- 2 years consuming data from Kafka
- 2 years producing data to Kafka
- 2 years with Rust in ~~production~~ academia



# Program

- Architecting a service using dependency injection
  - Using [https://github.com/TehPers/runtime\\_injector](https://github.com/TehPers/runtime_injector)

# The Problem

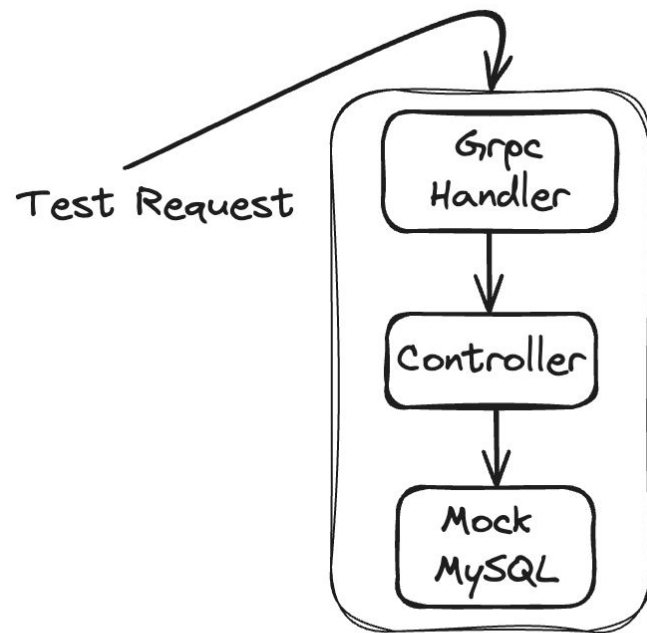
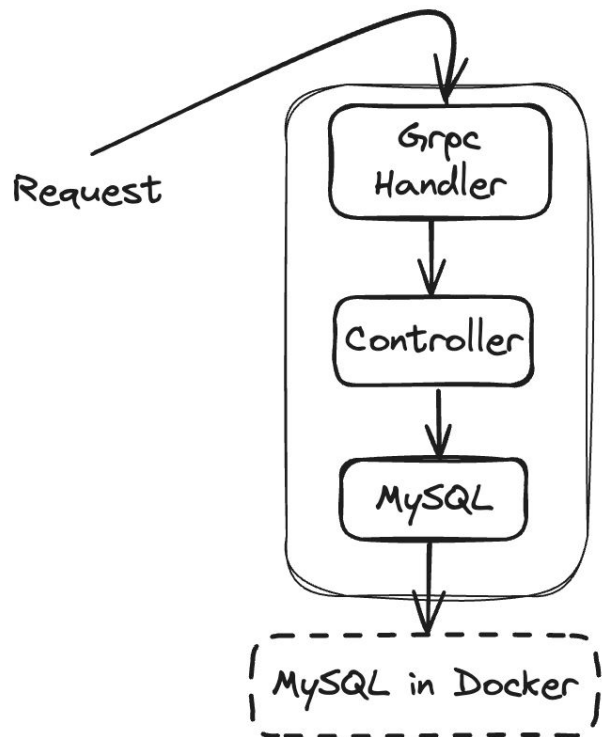
- Integration tests are hard when the app has external dependencies
- Migrations in production are difficult
- Different environments might use different modules

# The Goal

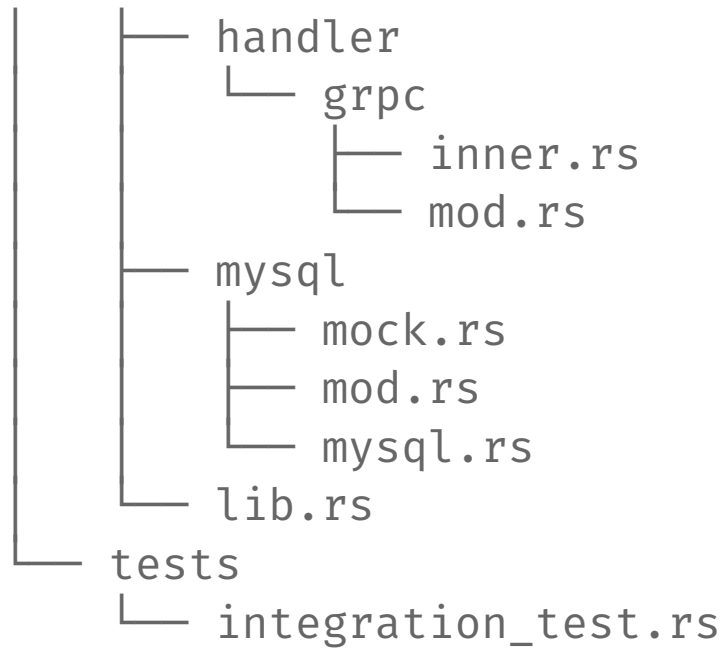
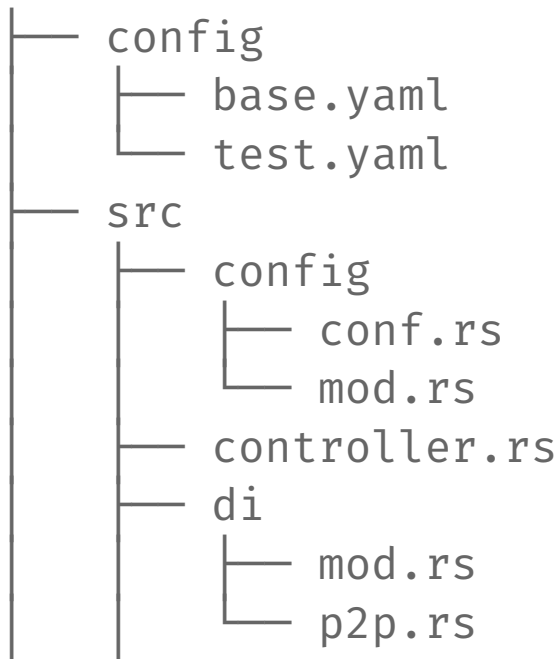
- A service that has swappable components
- Allows integration tests
- D from SOLID: Dependency inversion principle
  - Depend upon abstractions, not concretes

Credits to: [https://github.com/TehPers/runtime\\_injector](https://github.com/TehPers/runtime_injector)

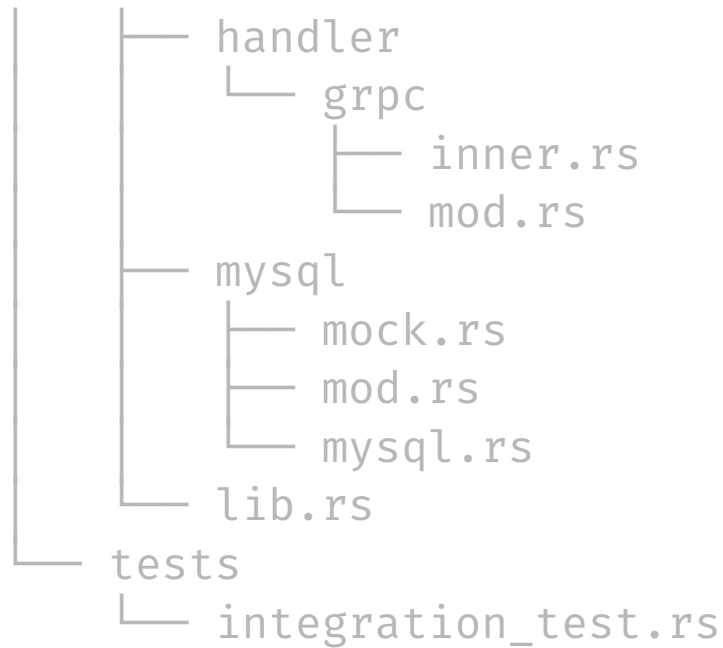
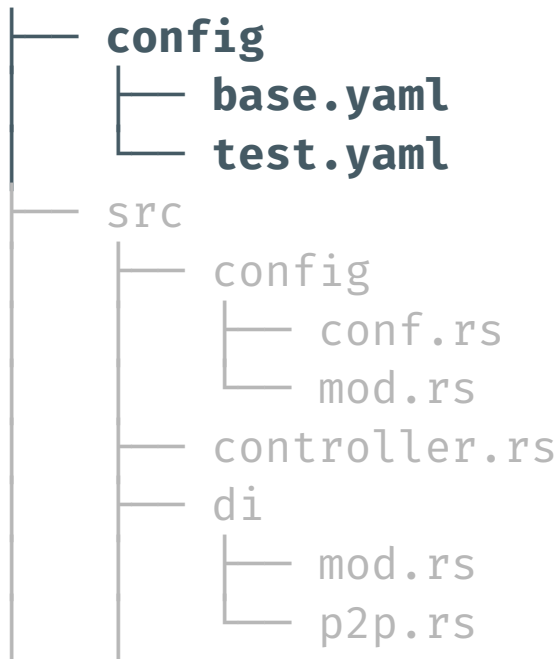
# The Architecture



# Project Structure



# Yaml configs





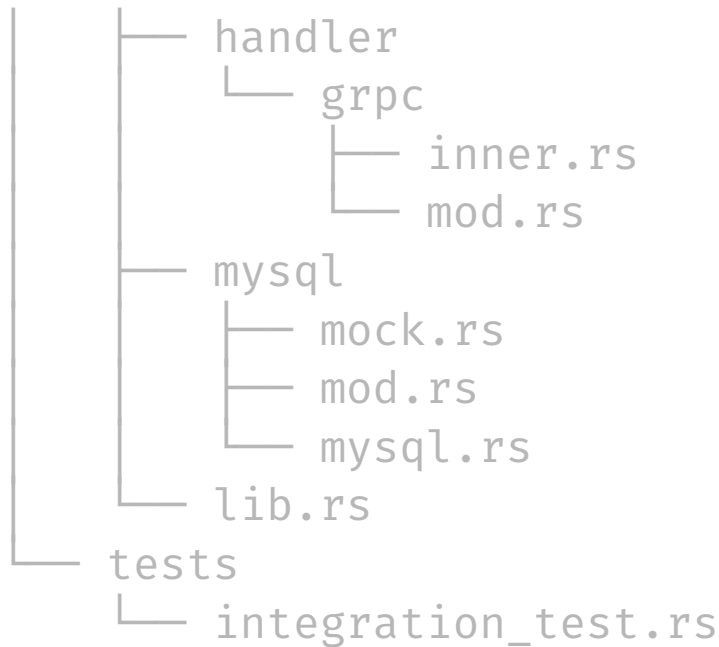
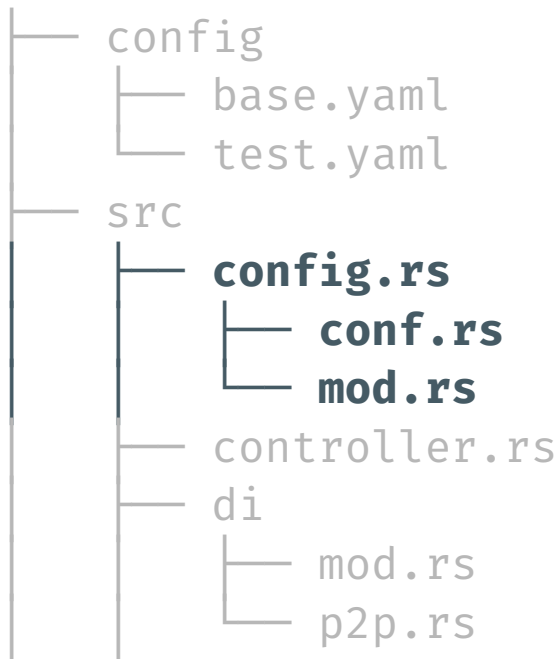
# base.yaml

```
storage:  
  type: mysql  
  
  host: localhost  
  port: 3306  
  username: mysql  
  password: mysql  
  db_name: my_db
```

# test.yaml

```
storage:  
  type: mock
```

# Reading the yaml configs



# The Config Interface

```
pub trait IConfig: Service {  
    fn storage(&self) -> Storage;  
}  
  
pub enum Storage {  
    Mysql(Mysql),  
    Mock,  
}  
  
pub struct Mysql {  
    pub username: String,  
    ...  
}
```

# Implementing the Config

```
pub struct Config {  
    pub storage: Storage,  
}  
  
impl IConfig for Config {  
    fn storage(&self) -> Storage {  
        self.storage.clone()  
    }  
}
```

# Config Constructor

```
pub struct ConfigProvider;

impl ServiceFactory<()> for ConfigProvider {
    type Result = Config;
    fn invoke(&mut self, ...) -> InjectResult<Self::Result> {
        let env_conf = env::var("ENV").unwrap_or_else(|_| "test.yaml".into());
        let mut builder = config::Config::builder();
        if Path::new("base.yaml").exists() {
            builder = builder.add_source(File::with_name("base.yaml"));
        }
        builder.add_source(File::with_name(env_conf).build())
    }
}
```

# Making the Config injectable

```
pub struct ConfigProvider;
impl ServiceFactory<()> for ConfigProvider {
    type Result = Config;
    fn invoke(&mut self, ...) -> InjectResult<Self::Result> {
        let env_conf = env::var("ENV").unwrap_or_else(|_| "dev".into());
        let mut builder = config::Config::builder();
        if Path::new("base.yaml").exists() {
            builder = builder.add_source(File::with_name("base.yaml"));
        }
        builder.add_source(File::with_name(env_conf).build())
    }
}
```

# Registering the Interface in the DI

```
pub trait IConfig: Service { ... }  
impl IConfig for Config { ... }  
pub struct ConfigProvider;  
impl ServiceFactory<()> for ConfigProvider {  
    type Result = Config;  
    fn invoke(&mut self, ...) -> InjectResult<Self::Result> { ... }  
}
```

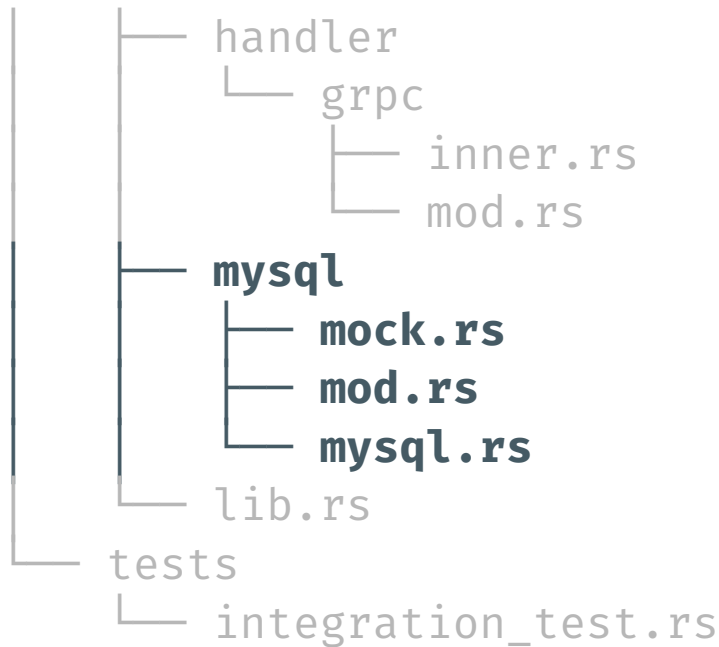
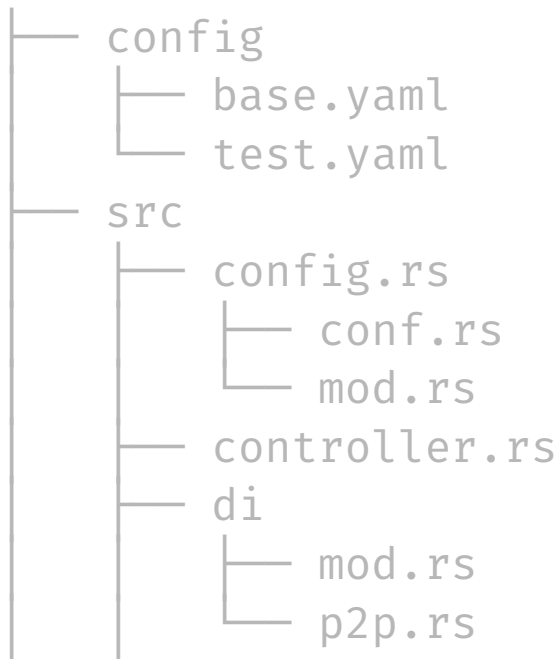
```
interface! {  
    dyn IConfig = [Config]  
}
```



# Injecting the Interface in the DI

```
pub fn dependency_injector() -> Result<Injector> {  
    let mut injector = Injector::builder();  
  
    injector.provide(  
        ConfigProvider.singleton()  
        .with_interface::<dyn IConfig>()  
    );  
  
    Ok(injector.build())  
}
```

# The Mysql module



# MySQL module

```
pub trait IMysql: Service {  
    async fn ping(&self) -> Result<()>;  
}  
  
pub struct Mysql { pub config: config::Mysql }  
  
pub struct MysqlMock { pings: HashMap<String, i32> }  
  
impl IMysql for Mysql { ... }  
impl IMysql for MysqlMock { ... }
```

# ServiceFactory does not work

```
pub struct MysqlProvider;

impl ServiceFactory<()> for MysqlProvider {
    type Result = Mysql;

    fn invoke(&mut self, injector: &Injector) ->
        InjectResult<Self::Result> {
        . . .
        Ok(match config {
            Storage::Mysql(config) => Svc::new(Mysql::new(config))
            Storage::Mock => Svc::new(MysqlMock::new())
        })
    }
}
```

# Replacing ServiceFactory with Provider

```
pub struct MysqlProvider { result: Option<DynSvc> }
impl Provider for MysqlProvider {
    fn provide(&mut self, injector: &Injector) ->
        InjectResult<DynSvc> {

        . . .

        Ok(match config {
            Storage::Mysql(config) => Svc::new(Mysql::new(config)) as DynSvc
            Storage::Mock => Svc::new(MysqlMock::new()) as DynSvc
        })
    }
}
```

# Why did we use the ServiceFactory?

```
pub fn dependency_injector() -> Result<Injector> {  
    let mut injector = Injector::builder();  
  
    injector.provide(  
        MysqlProvider.singleton()  
        .with_interface::<dyn IMysql>()  
    );  
  
    Ok(injector.build())  
}
```

# Implementing a Singleton

```
pub struct MysqlProvider { result: Option<DynSvc> }
impl Provider for MysqlProvider {
    fn provide(&mut self, injector: &Injector) ->
        InjectResult<DynSvc> {
        if let Some(ref service) = self.result {
            return Ok(service.clone());
        }
        . . .
        let result = match config {
            Storage::Mysql(config) => Svc::new(Mysql::new(config)) as DynSvc
        };
        self.result = Some(result.clone());
        Ok(result)
    }
}
```

# Getting the config from the Injector

```
pub struct MysqlProvider { result: Option<DynSvc> }
impl Provider for MysqlProvider {
    fn provide(&mut self, injector: &Injector) ->
        InjectResult<DynSvc> {
        . . .

        let config = injector.get::<Svc<dyn IConfig>>()?.storage();
        let result = match config {
            Storage::Mysql(config) => Svc::new(Mysql::new(config)) as DynSvc
        };
        . . .
    }
}
```



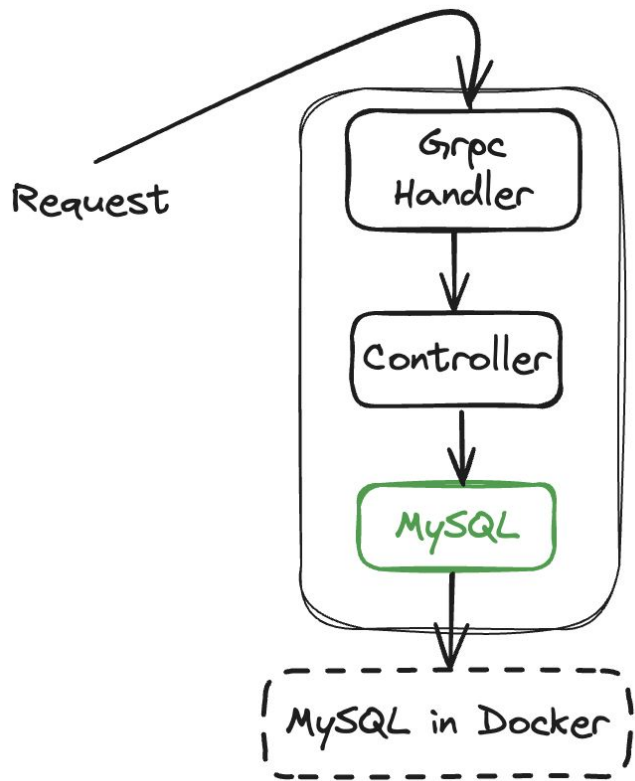
# Injecting the Interface in the DI

```
pub fn dependency_injector() -> Result<Injector> {  
    let mut injector = Injector::builder();  
  
    injector.provide(  
        ConfigProvider.singleton()  
        .with_interface::<dyn IConfig>()  
    );  
  
    Ok(injector.build())  
}
```

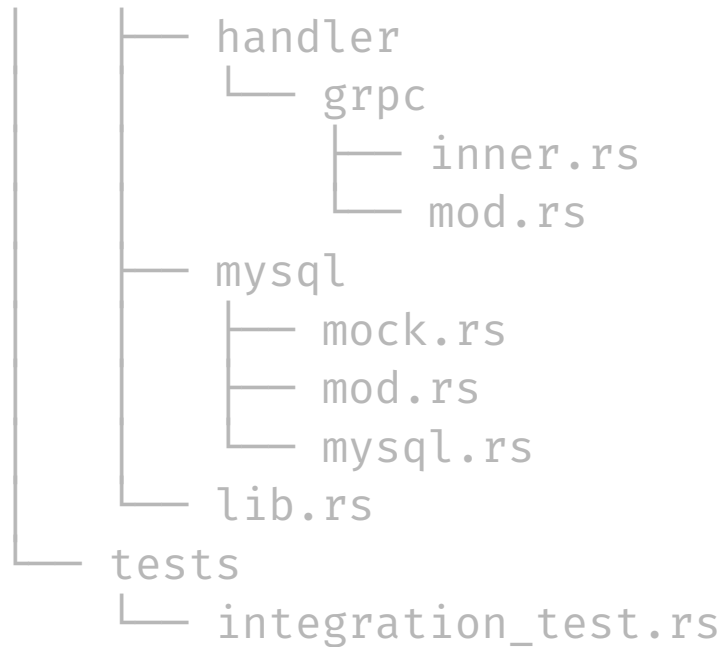
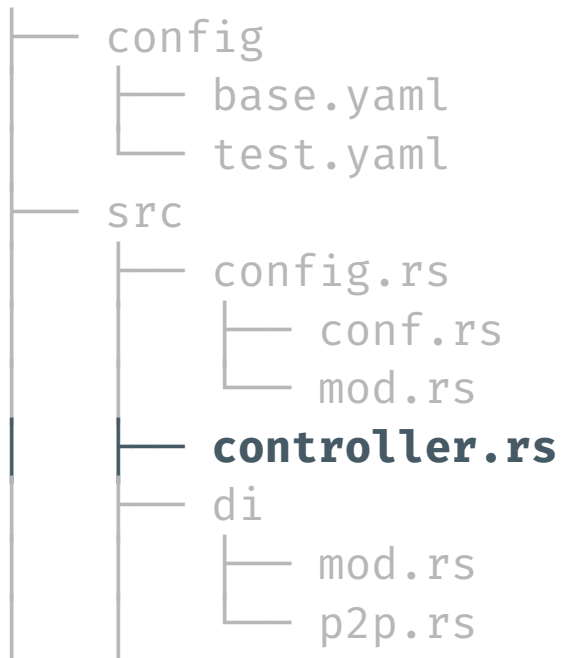
# Why did we use the ServiceFactory?

```
pub fn dependency_injector() -> Result<Injector> {  
  
    injector.provide(  
        ConfigProvider.singleton()  
        .with_interface::<dyn IConfig>()  
    );  
    injector.provide(  
        MysqlProvider::default()  
    );  
}
```

# MySQL



# Controller



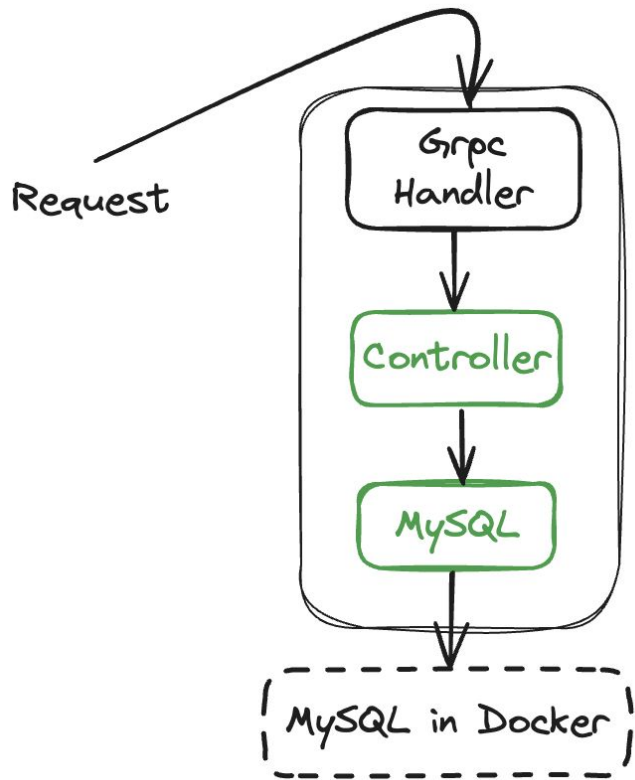
# Controller

```
pub trait ILocalController: Service { ... }

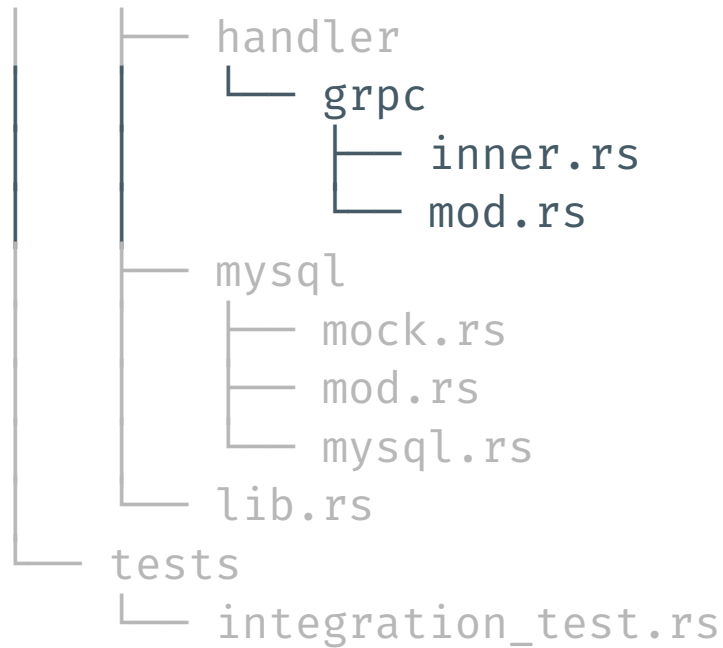
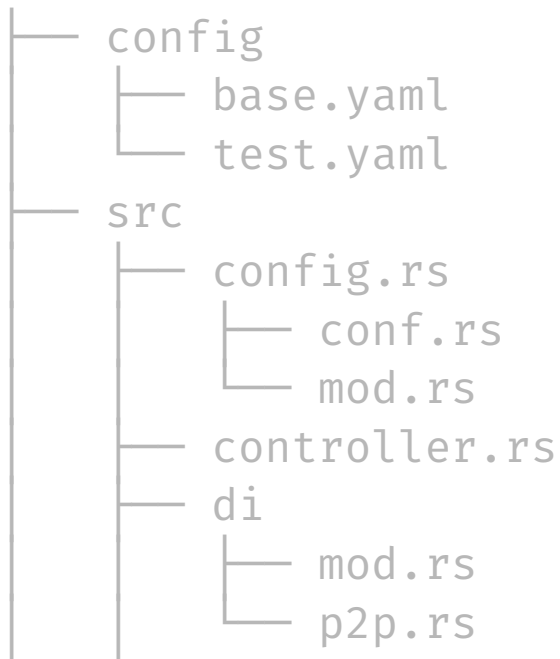
pub struct LocalControllerProvider;
impl ServiceFactory<()> for LocalControllerProvider {
    type Result = LocalController;
    fn invoke(&mut self, injector: &Injector) -> InjectResult<Self::Result> {
        let mysql = injector.get::<Svc<dyn IMysql>>()?;
        Ok(LocalController { mysql })
    }
}

interface! {
    dyn ILocalController = [LocalController]
}
```

# Controller



# gRPC Module



# gRPC Service contains a gRPC Handler

```
pub trait IGrpcService: Service {  
    async fn start(&self) -> Result<()>;  
}  
  
pub trait IGrpcHandler: Service {  
    async fn ping(&self) -> Result<()>;  
    async fn store(&self, s: String) -> Result<()>;  
}  
  
pub struct GrpcService {  
    handler: Svc<dyn IGrpcHandler>,  
}
```



# gRPC Service Provider

```
pub struct GrpcService {  
    handler: Svc<dyn IGrpcHandler>,  
}  
  
pub struct GrpcServiceProvider;  
impl ServiceFactory<()> for GrpcServiceProvider {  
    type Result = GrpcService;  
  
    fn invoke(&mut self, injector: &Injector) -> InjectResult<Self::Result> {  
        let handler = injector.get::<Svc<dyn IGrpcHandler>>()?;  
        Ok(GrpcService { handler, ... })  
    }  
}
```

# The tonic Server expects an object

```
pub struct GrpcService {  
    handler: Svc<dyn IGrpcHandler>,  
}  
  
impl IGrpcService for GrpcService {  
    async fn start(&self) -> Result<()> {  
        . . .  
        Server::builder()  
            .add_service(AppServiceServer::new(self.handler))  
            .await?;  
        . . .  
    }  
}
```

# Providing GrpcHandler as an object

```
pub struct GrpcServiceProvider;
impl ServiceFactory<()> for GrpcServiceProvider {
    type Result = GrpcService;

    fn invoke(&mut self, injector: &Injector) -> InjectResult<Self::Result> {
        let handler = injector.get:::<GrpcHandler>()?;
        Ok(GrpcService { handler, ... })
    }
}

pub struct GrpcService {
    handler: Svc<dyn GrpcHandler,
}
```

# Implementing the InjectorRequest

```
pub struct GrpcHandlerProvider;
impl ServiceFactory<()> for GrpcHandlerProvider {
    fn invoke(injector: &Injector) -> InjectResult<Self::Result> {
        GrpcHandler::request(injector, ...)
    }
}

impl InjectorRequest for GrpcHandler {
    fn request(injector: &Injector, ...) -> InjectResult<Self> {
        let controller = injector.get:::<Svc<dyn ILocalController>>()?;
        Ok(GrpcHandler { controller })
    }
}
```

# Providing GrpcHandler as an object

```
pub struct GrpcServiceProvider;
impl ServiceFactory<()> for GrpcServiceProvider {
    type Result = GrpcService;

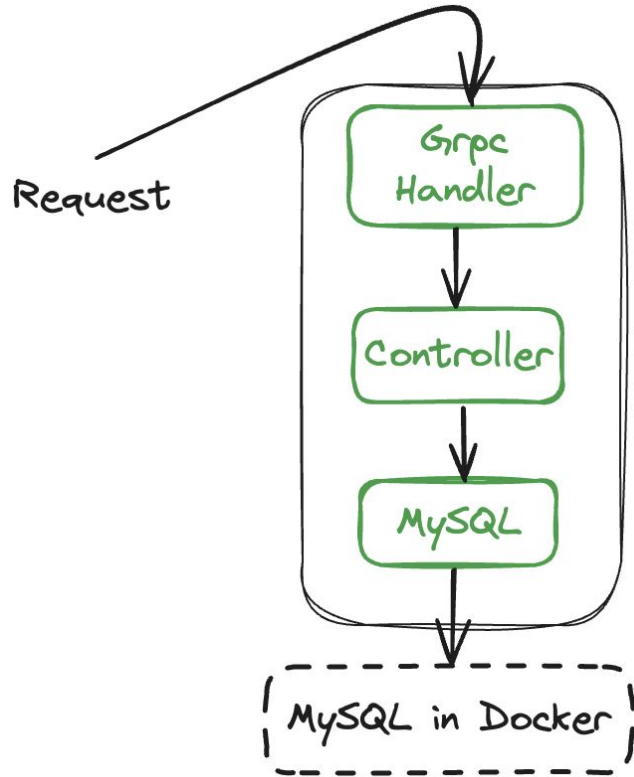
    fn invoke(&mut self, injector: &Injector) -> InjectResult<Self::Result> {
        let handler = injector.get::<GrpcHandler>()?;
        Ok(GrpcService { handler, ... })
    }
}

pub struct GrpcService {
    handler: GrpcHandler,
}
```

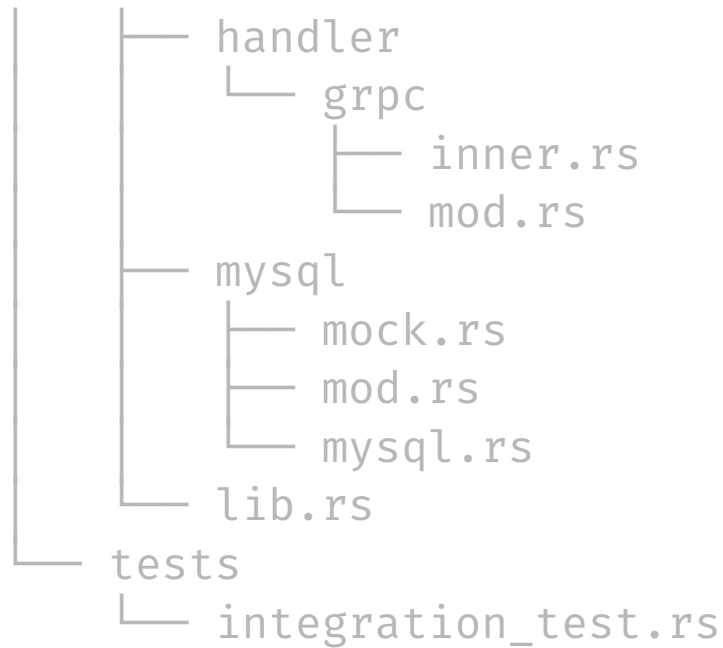
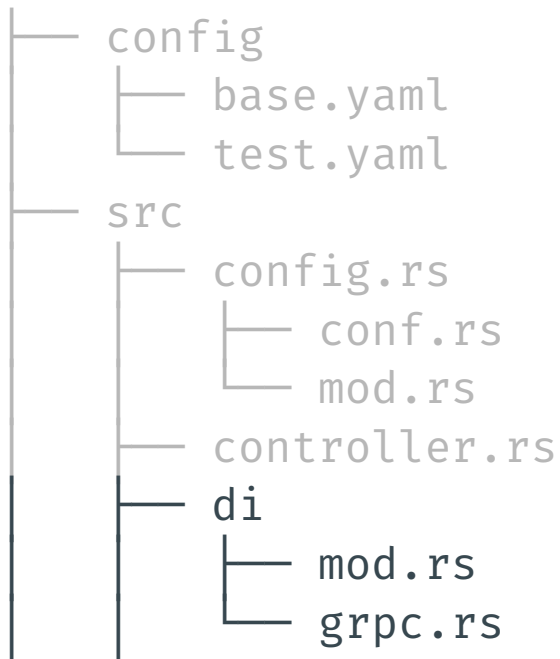
# The tonic Server POV :)

```
pub struct GrpcService {  
    handler: GrpcHandler,  
}  
  
impl IGrpcService for GrpcService {  
    async fn start(&self) -> Result<()> {  
        . . .  
        Server::builder()  
            .add_service(AppServiceServer::new(self.handler.to_owned()))  
            .await?;  
        . . .  
    }  
}
```

# Handler



# DI Module





# Final injector implementation

```
pub fn dependency_injector() -> Result<Injector> {  
    let mut injector = Injector::builder();  
    injector.provide(ConfigProvider.singleton()  
        .with_interface::<dyn IConfig>());  
    injector.provide(LocalControllerProvider.singleton()  
        .with_interface::<dyn ILocalController>(),  
    );  
    injector.provide(MysqlProvider::default());  
    . . .  
    Ok(injector.build())  
}
```

# gRPC DI module

```
pub fn module() -> runtime_injector::Module {  
    define_module! {  
        services = [  
            GrpcServiceProvider.singleton(),  
            GrpcHandlerProvider.singleton(),  
        ],  
        interfaces = {  
            dyn AppService = [ GrpcHandlerProvider.singleton() ],  
            dyn IGrpcService = [ GrpcServiceProvider.singleton() ],  
        },  
    }  
}
```

# Final v2.0 injector implementation

```
pub fn dependency_injector() -> Result<Injector> {  
    let mut injector = Injector::builder();  
    injector.provide(ConfigProvider.singleton()  
        .with_interface::<dyn IConfig>());  
    injector.provide(LocalControllerProvider.singleton()  
        .with_interface::<dyn ILocalController>(),  
    );  
    injector.provide(MysqlProvider::default());  
    injector.add_module(grpc::module());  
    Ok(injector.build())  
}
```

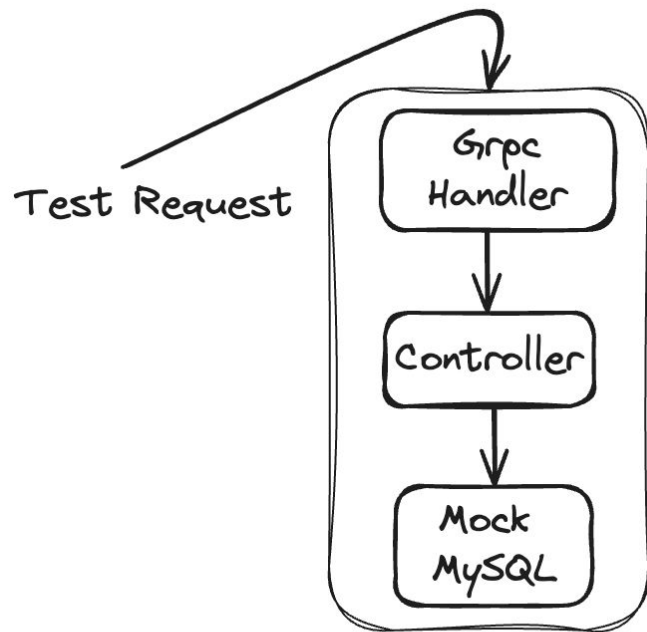
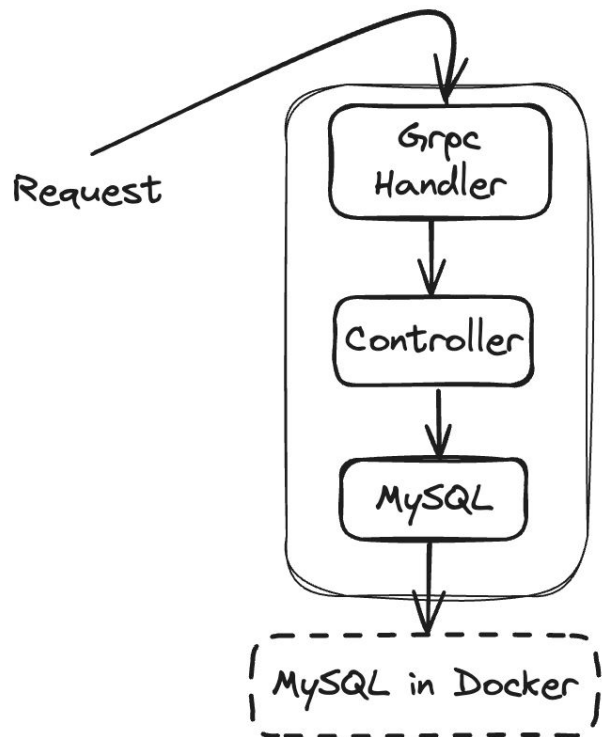
# main.rs

```
pub async fn run() -> Result<()> {  
    let injector = dependency_injector()?;  
    let grpc_service: Svc<dyn IGrpcService> = injector.get()?;  
  
    match try_join!(grpc_service.start()) {  
        Err(err) => die(err),  
        _ => {}  
    };  
    Ok(())  
}
```

# integration\_test.rs

```
#[tokio::test]
async fn test_local_ping() {
    env::set_var("ENV", "test");
    let di = dependency_injector().unwrap();
    let (serve_future, mut client) = server_and_client_stub(di).await;
    let request_future = async {
        let response = client.ping(PingRequest{}).await.unwrap().into_inner();
        assert_eq!(response, PingResponse {});
    };
}
```

# The Architecture revisited



# The Goal

- A service that has swappable components
- Allows integration tests
- D from SOLID: Dependency inversion principle
  - Depend upon abstractions, not concretes

<https://github.com/luchev/rust-scaffold>





# Credits

- [https://github.com/TehPers/runtime\\_injector](https://github.com/TehPers/runtime_injector)
- <https://romannurik.github.io/SlidesCodeHighlighter/>

# Providing constants

```
pub fn module() -> runtime_injector::Module {
    let (sender, receiver) = channel::<Command>(5);
    define_module! {
        services = [
            SwarmControllerProvider.singleton(),
            SwarmProvider.singleton(),
            constant(Mutex::new(sender)),
            constant(Mutex::new(receiver)),
        ],
        interfaces = {
            dyn IRemoteController = [ RemoteControllerProvider.singleton() ],
            dyn IRemote = [ RemoteProvider.singleton() ],
        },
    }
}
```

# Static (compile time) vs Dynamic DI

- Compile time
  - Shaku
    - <https://github.com/AzureMarker/shaku>
  - Teloc
    - <https://github.com/p0lunin/teloc>
- Dynamic
  - Runtime\_injector
    - [https://github.com/TehPers/runtime\\_injector](https://github.com/TehPers/runtime_injector)

# Async

- Supports async constructors
  - <https://github.com/udoprogram/async-injector>

# Macro-heavy

- More macro magic and no boilerplate code
  - <https://github.com/azureblaze/lockjaw>
  - <https://github.com/nicolascotton/nject>