Services

A Grafana *service* encapsulates and exposes application logic to the rest of the application, through a set of related operations.

Grafana uses Wire, which is a code generation tool that automates connecting components using dependency injection. Dependencies between components are represented in Wire as function parameters, encouraging explicit initialization instead of global variables.

Even though the services in Grafana do different things, they share a number of patterns. To better understand how a service works, let's build one from scratch!

Before a service can start communicating with the rest of Grafana, it needs to be registered with Wire, see `ProvideService` factory function/method in the service example below and how it's being referenced in the wire.go example below.

When Wire is run it will inspect the parameters of `ProvideService` and make sure that all it's dependencies has been wired up and initialized properly.

Service example:

```go
package example

// Service service is the service responsible for X, Y and Z.
type Service struct {
    logger   log.Logger
    cfg      *setting.Cfg
    sqlStore *sqlstore.SQLStore
}

// ProvideService provides Service as dependency for other services.
func ProvideService(cfg *setting.Cfg, sqlStore *sqlstore.SQLStore) (*Service, error) {
    s := &Service{
        logger:     log.New("service"),
        cfg:        cfg,
        sqlStore:   sqlStore,
    }

    if s.IsDisabled() {
        // skip certain initialization logic
        return s, nil
    }

    if err := s.init(); err != nil {
        return nil, err
    }
```

```
    return s, nil
}

func (s *Service) init() error {
    // additional initialization logic...
    return nil
}

// IsDisabled returns true if the service is disabled.
//
// Satisfies the registry.CanBeDisabled interface which will guarantee
// that Run() is not called if the service is disabled.
func (s *Service) IsDisabled() bool {
    return !s.cfg.IsServiceEnabled()
}

// Run runs the service in the background.
//
// Satisfies the registry.BackgroundService interface which will
// guarantee that the service can be registered as a background service.
func (s *Service) Run(ctx context.Context) error {
    // background service logic...
    <-ctx.Done()
    return ctx.Err()
}
```

wire.go

```
// +build wireinject

package server

import (
    "github.com/google/wire"
    "github.com/grafana/grafana/pkg/example"
    "github.com/grafana/grafana/pkg/services/sqlstore"
)

var wireBasicSet = wire.NewSet(
    example.ProvideService,

)

var wireSet = wire.NewSet(
    wireBasicSet,
    sqlstore.ProvideService,
)
```

```go
var wireTestSet = wire.NewSet(
    wireBasicSet,
)

func Initialize(cla setting.CommandLineArgs, opts Options, apiOpts api.ServerOptions) (*Serv
    wire.Build(wireExtsSet)
    return &Server{}, nil
}

func InitializeForTest(cla setting.CommandLineArgs, opts Options, apiOpts api.ServerOptions,
    wire.Build(wireExtsTestSet)
    return &Server{}, nil
}
```

Background services

A background service is a service that runs in the background of the lifecycle between Grafana starts up and shutdown. If you want a service to be run in the background your Service should satisfy the `registry.BackgroundService` interface and add it as argument to the ProvideBackgroundServiceRegistry function and add it as argument to `NewBackgroundServiceRegistry` to register it as a background service.

You can see an example implementation above of the Run method.

Disabled services

If you want to guarantee that a background service is not run by Grafana when certain criteria is met/service is disabled your service should satisfy the `registry.CanBeDisabled` interface. When the service.IsDisabled method return false Grafana would not call the service.Run method.

If you want to run certain initialization code if service is disabled or not, you need to handle this in the service factory method.

You can see an example implementation above of the IsDisabled method and custom initialization code when service is disabled.

Run Wire / generate code

When running `make run` it will call `make gen-go` on the first run. `gen-go` in turn will call the wire binary and generate the code in wire_gen.go and wire_gen.go. The wire binary is installed using bingo which will make sure to download and install all the tools needed, including the Wire binary at using a specific version.

OSS vs Enterprise

Grafana OSS and Grafana Enterprise shares code and dependencies. Grafana Enterprise might need to override/extend certain OSS services.

There's a wireexts_oss.go that has the `wireinject` and `oss` build tags as requirements. Here services that might have other implementations, e.g. Grafana Enterprise, can be registered.

Similarly, there's a wireexts_enterprise.go file in the Enterprise source code repository where other service implementations can be overridden/be registered.

To extend oss background service create a specific background interface for that type and inject that type to ProvideBackgroundServiceRegistry instead of the concrete type. Then add a wire binding for that interface in wireexts_oss.go and in the enterprise wireexts file.

Methods

Any public method of a service should take `context.Context` as its first argument. If the method calls the bus, other services or the database the context should be propagated, if possible.