Backend

This directory contains the code for the Grafana backend. This document gives an overview of the directory structure, and ongoing refactorings.

For more information on developing for the backend:

- Backend style guide
- Architecture

Central folders of Grafana's backend

folder	description
/pkg/api	HTTP handlers and routing. Almost all handler funcs are global which is something we would like to improve in the future. Handlers should be associated with a struct that refers to all dependencies.
/pkg/cmd	The binaries that we build: grafana-server and grafana-cli.
/pkg/components	A mix of third-party packages and packages we have implemented ourselves. Includes our packages that have out-grown the util package and don't naturally belong somewhere else.
/pkg/infra	Packages in infra should be packages that are used in multiple places in Grafana without knowing anything about the Grafana domain.
/pkg/services	Packages in services are responsible for persisting domain objects and manage the relationship between domain objects. Services should communicate with each other using DI when possible. Most of Grafana's codebase still relies on global state for this. Any new features going forward should use DI.
/pkg/tsdb	All backend implementations of the data sources in Grafana. Used by both Grafana's frontend and alerting.
/pkg/util	Small helper functions that are used in multiple parts of the codebase. Many functions are placed directly in the util folders which is something we want to avoid. Its better to give the util function a more descriptive package name. Ex `errutil`.

Central components of Grafana's backend

package	description
/pkg/bus	The bus is described in more details under Communication
/pkg/models	This is where we keep our domain model. This package should not depend on any package outside standard library. It does contain some references within Grafana but that is something we should avoid going forward.
/pkg/registry	Package for managing services.
/pkg/services/alerting	Grafana's alerting services. The alerting engine runs in a separate goroutine and shouldn't depend on anything else within Grafana.
/pkg/services/sqlstore	Currently where the database logic resides.

/pkg/setting	Anything related to Grafana global configuration should be dealt with in this package.

Dependency management

Refer to UPGRADING_DEPENDENCIES.md.

Ongoing refactoring

These issues are not something we want to address all at once but something we will improve incrementally. Since Grafana is released at a regular schedule the preferred approach is to do this in batches. Not only is it easier to review, but it also reduces the risk of conflicts when cherry-picking fixes from main to release branches. Please try to submit changes that span multiple locations at the end of the release cycle. We prefer to wait until the end because we make fewer patch releases at the end of the release cycle, so there are fewer opportunities for complications.

Global state

Global state makes testing and debugging software harder and it's something we want to avoid when possible. Unfortunately, there is quite a lot of global state in Grafana.

We want to migrate away from this by using the `inject` package to wire up all dependencies either in `pkg/cmd/grafana-server/main.go` or self-registering using `registry.RegisterService` ex https://github.com/grafana/grafana/blob/main/pkg/services/cleanup/cleanup.go#L25.

Limit the use of the init() function

Only use the init() function to register services/implementations.

Settings refactoring

The plan is to move all settings to from package level vars in settings package to the setting.Cfg struct. To access the settings, services and components can inject this setting.Cfg struct:

Cfg struct Injection example

Reduce the use of GoConvey

We want to migrate away from using GoConvey. Instead, we want to use stdlib testing, because it's the most common approach in the Go community and we think it will be easier for new contributors. Read more about how we want to write tests in the style guide.

Refactor SqlStore

The `sqlstore` handlers all use a global xorm engine variable. Refactor them to use the `SqlStore` instance.

Avoid global HTTP handler functions

Refactor HTTP handlers so that the handler methods are on the HttpServer instance or a more detailed handler struct. E.g (AuthHandler). This ensures they get access to HttpServer service dependencies (and Cfg object) and can avoid global state.

Date comparison

Store newly introduced date columns in the database as epochs if they require date comparison. This permits a unified approach for comparing dates against all the supported databases instead of handling dates differently for each database. Also, by comparing epochs, we no longer need error pruning transformations to and from other time zones.

Avoid use of the simplejson package

Use of the `simplejson` package (`pkg/components/simplejson`) in place of types (Go structs) results in code that is difficult to maintain. Instead, create types for objects and use the Go standard library's `encoding/json` package.

Provisionable*

All new features that require state should be possible to configure using config files. For example:

- Data sources
- Alert notifiers
- Dashboards

Today its only possible to provision data sources and dashboards but this is something we want to support all over Grafana.

Use context.Context "everywhere"

The package context should be used and propagated through all the layers of the code. For example the `context.Context` of an incoming API request should be propagated to any other layers being used such as the bus, service and database layers. Utility functions/methods normally doesn't need `context.Context` To follow best practices, any function/method that receives a context.Context argument should receive it as its first argument.

To be able to solve certain problems and/or implement and support certain features making sure that `context.Context` is passed down through all layers of the code is vital. Being able to provide contextual information for the full life-cycle of an API request allows us to use contextual logging, provide contextual information about the authenticated user, create multiple spans for a distributed trace of service calls and database queries etc.

Code should use `context.TODO` when it's unclear which Context to use or it is not yet available (because the surrounding function has not yet been extended to accept a `context.Context` argument).

More details in Services, Communication and Database.

Original design doc.