# Communication

Grafana uses a *bus* to pass messages between different parts of the application. All communication over the bus happens synchronously.

There are three types of messages: *events*, *commands*, and *queries*.

## Events

An event is something that happened in the past. Since an event has already happened, you can't change it. Instead, you can react to events by triggering additional application logic to be run, whenever they occur.

> *Because they happened in the past, event names are written in past tense, such as* `UserCreated`*, and* `OrgUpdated`*.*

### Subscribe to an event

In order to react to an event, you first need to *subscribe* to it.

To subscribe to an event, register an *event listener* in the service's `Init` method:

```
func (s *MyService) Init() error {
    s.bus.AddEventListener(s.UserCreated)
    return nil
}

func (s *MyService) UserCreated(event *events.UserCreated) error {
    // ...
}
```

**Tip:** Browse the available events in the `events` package.

### Publish an event

If you want to let other parts of the application react to changes in a service, you can publish your own events:

```
event := &events.StickersSentEvent {
    UserID: "taylor",
    Count:   1,
}
if err := s.bus.Publish(event); err != nil {
    return err
}
```

## Commands

A command is a request for an action to be taken. Unlike an event's fire-and-forget approach, a command can fail as it is handled. The handler will then return an error.

> *Because we request an operation to be performed, command are written in imperative mood, such as* `CreateFolderCommand`*, and* `DeletePlaylistCommand`*.*

### Dispatch a command

To dispatch a command, pass the `context.Context` and object to the `DispatchCtx` method:

```
// context.Context from caller
ctx := req.Request.Context()
cmd := &models.SendStickersCommand {
    UserID: "taylor",
    Count: 1,
}
if err := s.bus.DispatchCtx(ctx, cmd); err != nil {
  if err == bus.ErrHandlerNotFound {
    return nil
  }
  return err
}
```

> ***Note:*** *`DispatchCtx` will return an error if no handler is registered for that command.*

> ***Note:*** *`Dispatch` currently exists and requires no `context.Context` to be provided, but it's strongly suggested to not use this since there's an ongoing refactoring to remove usage of non-context-aware functions/methods and use context.Context everywhere.*

**Tip:** Browse the available commands in the `models` package.

### Handle commands

Let other parts of the application dispatch commands to a service, by registering a *command handler*:

To handle a command, register a command handler in the `Init` function.

```
func (s *MyService) Init() error {
    s.bus.AddHandlerCtx(s.SendStickers)
    return nil
}

func (s *MyService) SendStickers(ctx context.Context, cmd
*models.SendStickersCommand) error {
    // ...
}
```

> ***Note:*** *The handler method may return an error if unable to complete the command.*

> ***Note:*** *`AddHandler` currently exists and requires no `context.Context` to be provided, but it's strongly suggested to not use this since there's an ongoing refactoring to remove usage of non-context-aware functions/methods and use context.Context everywhere.*

## Queries

A command handler can optionally populate the command sent to it. This pattern is commonly used to implement *queries*.

### Making a query

To make a query, dispatch the query instance just like you would a command. When the `DispatchCtx` method returns, the `Results` field contains the result of the query.

```
// context.Context from caller
ctx := req.Request.Context()
query := &models.FindDashboardQuery{
    ID: "foo",
}
if err := bus.Dispatch(ctx, query); err != nil {
    return err
}
// The query now contains a result.
for _, item := range query.Results {
    // ...
}
```

> ***Note:*** *`Dispatch` currently exists and requires no `context.Context` to be provided, but it's strongly suggested to not use this since there's an ongoing refactoring to remove usage of non-context-aware functions/methods and use context.Context everywhere.*

### Return query results

To return results for a query, set any of the fields on the query argument before returning:

```
func (s *MyService) FindDashboard(ctx context.Context, query
*models.FindDashboardQuery) error {
    // ...
    query.Result = dashboard
    return nil
}
```