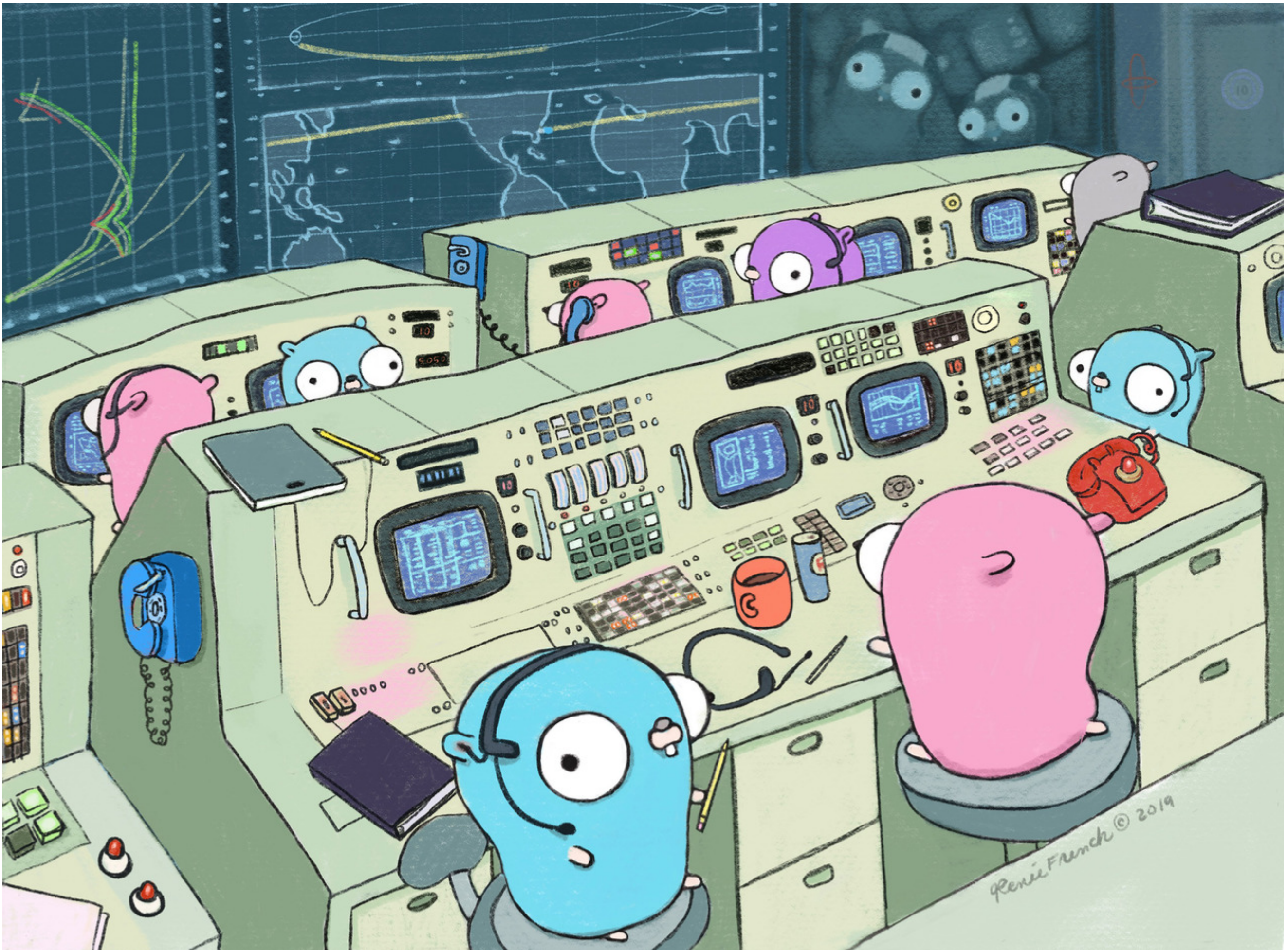
go-kernel Microkernel for Go

Creating & reusing services within a single Go runtime



Peter Mount, Area51.dev & Contributors

CC BY-SA

go-kernel Microkernel for Go

Creating & reusing services within a single Go runtime

Title	go-kernel Microkernel for Go
Subtitle	Creating & reusing services within a single Go runtime
Author	Peter Mount, Area51.dev & Contributors
Copyright	CC BY-SA

Front Page Image

Title	Go 10th anniversary Gopher
Author	Renee French
Copyright	Creative Commons Attribution 3.0

CC BY-SA version 4.0 license

You are free to:

1. **Share** — copy and redistribute the material in any medium or format
2. **Adapt** — remix, transform, and build upon the material or any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

1. **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
2. **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
3. **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

You can read the full license here: https://creativecommons.org/licenses/by-sa/4.0/

Table of Contents

1 Bootstrap
2 Kernel Lifecycle
3 Services
3.1 Naming services
3.2 Dependencies
3.3 Service Dependency Injection
3.4 Initialise & Service Dependencies
3.5 PostInit
3.6 Start & Stop a service
3.7 Run
4 Support Services
4.1 Cron scheduler service
4.1.1 Add a Schedule
4.1.1.1 Custom Schedules
4.1.2 Remove an existing Schedule
4.1.3 Cron specification

go-kernel is a simple microkernel which can run multiple services within the same go runtime. The kernel provides both lifecycle support for those services and dependency management so that services start and stop in the correct order.

1 - Bootstrap

To create an application requires a simple bootstrap which contains the core Service's that are required to run the application.

For example, this site is generated with a microkernel based application and it uses a bootstrap similar to the following:

```
1 package main
2
3 import (
4   "github.com/peter-mount/documentation/tools/hugo"
5   "github.com/peter-mount/documentation/tools/pdf"
6   "github.com/peter-mount/go-kernel"
7   "log"
8 )
9
10 func main() {
11   err := kernel.Launch(
12     &hugo.Hugo{},
13     &pdf.PDF{},
14   )
15   if err != nil {
16     log.Fatal(err)
17   }
18 }
```

Here it defines two services which must be started. Each one is declared by passing an empty instance of the required Service's to the Launch method. The kernel will then deploy each one.

The order they are listed here will be used to determine which one starts first, although this can be affected by any dependencies on services that have already been deployed.

Peter Mount, Area51.dev & Contributors CC BY-SA

2 - Kernel Lifecycle

When the kernel.Launch() function is called from the Bootstrap it starts an instance of the microkernel which then goes through each stage of its lifecycle.

It exits when either the entire sequence has been completed or if a service fails part way though, at which point it will return the error the failed service has returned.

The following is an overview of this sequence. The details of each stage is described in full under the Services section of the documentation, however a Service does not have to implement any of them, they are all optional.

Inject & Init
: This is the initial stage which is split into two parts.

The first part is Injection where dependencies are injected into the service (v1.1.0 or later).

The second part is Init where services can declare they depend on other services by implementing the InitialisableService. They can also define any command line flags they want to use via either mechanism.

Once this stage completes no new services can be added to the kernel.

PostInit
: The second stage is PostInit which allows a service to check it has everything it needs. Usually this is to check that any command line flags are valid or mandatory ones are present.

Start
: The third stage is Start. A service can implement the StartableService interface so that it can perform any initialisation it requires. For example, it could load a configuration file, open a database.

Once a service has started it can be safely used by any dependents.

If a service fails to start the kernel will skip to the Stop stage to stop any previously started service before exiting with the error of the failure.

Run
: When the kernel reaches the Run stage, every service has now been started.

The kernel will then go though every deployed service and if they implement the RunnableService interface it will the Run function. If that function returns an error then the kernel will stop and enter the Stop stage to stop any previously started service before exiting with the error of the failure.

Be aware that it is possible for a single service to never exit from this stage. An example of this is the rest web server service provided by the kernel. As this has to run forever the RunnableService implementation never returns keeping the application running.

For this reason the RunnableService interface is usually used for command line tools. For Server Daemons like a rest service this interface should not be used as you cannot guarantee it will ever be called.

In the majority of purposes, applications would use the worker task Queue instead of implementing Run as it allows for more flexible task scheduling within the application.

Stop
: Stop is the final stage where every service that implements the StoppableService interface that has also passed through the Start stage will have their Stop function called. This allows a service to release any resources it has open.

Note: this will call those services in reverse order. Specifically, the most recently started service will be stopped first. This is done so that services are stopped in the correct sequence.

Also, a StoppableService does not need to implement StartableService. It is perfectly fine to implement just Stop() and not Start().

This stage can be invoked at various stages:

1. When a service's Start() method returns an error
2. When a service's Run() method returns an error
3. When the Run stage completes
4. The process receives a SIGINT(2) or SIGTERM(15) signal

Peter Mount, Area51.dev & Contributors CC BY-SA

3 - Services

A Service is the core entity managed by the microkernel which will provide it with lifecycle support and dependency management.

When a service is deployed it becomes managed by the kernel and follow a strict lifecycle:

Lifecycle	Description	Function
Inject	Resolve dependencies on other services, Declare command line flags	field injection
Init		Init(*Kernel) error
PostInit	Allow checks after init, e.g. command line arguments are correct	PostInit() error
Start	Allows a service to start, open files, databases etc.	Start()
Run	Allows a service to perform a task. This is deprecated as tasks should be performed using the task worker queue instead.	Run() error
Stop	Allows a service to free any resources as it shuts down.	Stop()

The details of each lifecycle stage is described in full in the following sections, however every one of them is optional as a service does not require any of them to be implemented for a service to be deployed.

The Inject lifecycle was introduced in V1.1.0 and is an alternative to Init although both can be used under certain cirumstances.

v1.1.0

As of version 1.1.0 a service can be any struct. This struct would, when deployed within the kernel become a singleton instance available for injection into other services.

Lifecycle handling can be performed by means of the lifecycle interfaces described below.

Before v1.1.0

Before version 1.1.0 a service consisted of a struct type which implemented the Service interface and optionally any of the lifecycle interfaces.

The Service interface required the Name() function which provided the unique name for this service:

```
1 type Service interface {
2     Name() string
3 }
```

The Name() function must return a unique identifier for your service. Usually this can be the services name, just as long as it's unique with any other services deployed within the kernel.

For Example, a bare minimal Service can be defined

```
1 type MyService struct {
2 }
3 
4 func (s *MyService) Name() string {
5     return "MyService
6 }
```

With this basic definition this new service can now be deployed into the kernel. As this example has no lifecycle functions defined the kernel will just deploy it and make it available to any service that requires access to it.

Peter Mount, Area51.dev & Contributors CC BY-SA

3.1 - Naming services

When a service is deployed, it is deployed using a unique name. It is this name which indicates if the service has already been deployed when dependency injection is performed.

As of version 1.1.0 a service can be any struct so by default it would be assigned a unique name based on it's package path and type name providing a unique name for it's type.

Alternatively the service can have the `Name() string` function which will provide the service's name.

For applications using v1.1.0 or later it's advised to not implement Name() and leave the naming to the Kernel. The Name() function will still be supported as it has uses for specialist use-cases.

Before v1.1.0

In kernel's prior to v1.1.0 the Name() function was mandatory and was the only means of providing a unique name to a service.

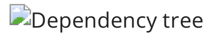
Peter Mount, Area51.dev & Contributors CC BY-SA

3.2 - Dependencies

A service can have dependencies on other services. During startup, when a service declares it has a dependency then that service will be started before it's dependent. This ensures that a services' functionality is in a valid state during the lifetime of the application.

Dependencies are declared in a service by either injection or in the Init lifecycle phase.

The the following example we have an application with 4 services. Service A depends on B & C, B on C and C with D. This relationship is shown with the black arrows.


The red arrows show dependencies which are invalid as they create circular dependency which is described below.

Circular dependencies

The kernel does not allow circular dependencies.

In the example above we have four services called A, B, C and D. The black arrows show the valid dependencies but the red arrows show illegal dependencies - that is, a dependency that cannot be made as it would create a circular dependency.

The reason this is an issue is that, when a circular dependency exists the kernel cannot determine the correct starting order.

e.g. if C depends on A, but A indirectly depends on C via B which one should start first?

If this situation happens during the injection or init lifecycle phases the kernel itself will stop returning an error.

Peter Mount, Area51.dev & Contributors CC BY-SA

3.3 - Service Dependency Injection

The first lifecycle stage a service goes through is injection. Here a service can declare any other services it has a dependency on or any command line flags it requires by use of field tags within its structure.

For example, here we have a simple service showing the possible combinations available by the Injection lifecycle:

```
1 type Example struct {
2   config  *conf.Config `kernel:"inject"`
3   worker  task.Queue   `kernel:"worker"`
4   _       *PostCSS     `kernel:"inject"`
5   server  *bool        `kernel:"flag,s,Run hugo in server mode"`
6 }
```

Service dependencies

A service can declare a dependency against another service by including th e `kernel:"inject"` tag against a field of the type of the service.

In the above example we have the field config which has the type `*conf.Config`. As that field has the tag then the kernel will inject that service's pointer into that field. When the service starts, it will be started before this one.

Unreferenced Service dependencies

An unreferenced dependency is a dependency on another service, but we won't call that service directly. An example of this is a webserver - we need the service to be running whilst this service is running, but we won't call the service directly, e.g. we might make an HTTP connection to it but not via code.

In the above example, the PostCSS service is one of these. We want that service to be deployed however we won't be using it directly so the field name is `_`. This ensures that service is deployed but not injected into the struct.

Worker Task Queue

The worker task queue is a kernel service allowing for tasks to be queued and prioritised for execution. It's injected using the `kernel:"worker"` tag.

Command line flags

Command line flags can be injected using the `kernel:"flag"` tag. This is a composite tag consisting of up to 4 comma separated fields. The first field is always `flag`.

To inject a flag, simply create a field with its type being one of: *bool, *string, *int, *int64 or *float64. Any other type will cause the service to fail.

The tag takes the following format: `flag,{name,{description,{default}}}`

name
The name of this flag. It will have '-' prepended to it when used on the command line.
If absent or "" then this will take the name of the field being tagged.

description
The description for this flag, used when listing the available flags on the command line.
If absent or "" then this will take the value used for the flag name.

default
The default value for this flag.
If absent or "" then this will take a suitable default for the type: "" for string, false for boolean, 0 for integers or 0.0 for floats.

Peter Mount, Area51.dev & Contributors CC BY-SA

3.4 - Initialise & Service Dependencies

The second lifecycle stage a service goes through is init. Here a service can declare any other services it has a dependency on or any command line flags it requires.

> Since v1.1.0 the Init lifecycle has been replaced by Injection for most purposes.
>
> There are some use-cases where Init is still useful but for most purposes you will find Injection is neater and easier to maintain with less boilerplate code.

To do this the service needs to implement the InitialisableService interface:

```
type InitialisableService interface {
  Init(*Kernel) error
}
```

If an error is returned from this Init() method the Kernel will exit immediately returning that error.

> A service must never call a function in another service from inside the Init() method, nor create any external resources like go routines or open files.
>
> Doing so could call a service which has not yet been initialised and leave resources open if the kernel exits due to an error.

Command line flags

To add a simple flag to a service we simply use the flag package to create the flag from within the Init function.

```
package hugo

import (
    "flag"
    "github.com/peter-mount/documentation/tools/util"
    "github.com/peter-mount/go-kernel"
)

// Hugo runs hugo
type Hugo struct {
    server *bool // true to run Hugo in server mode
}

func (h *Hugo) Name() string {
    return "hugo"
}

func (h *Hugo) Init(_ *kernel.Kernel) error {
    h.server = flag.Bool("s", false, "Run hugo in server mode")
    return nil
}
```

Once the kernel has completed the Init stage we can then reference the flag's value as it would be set with the command line flag from the command line.

Peter Mount, Area51.dev & Contributors CC BY-SA

Service dependencies

A service can declare a dependency against another service by using the Kernel instance passed to the Init method. The passed instance is only valid for this call, and you should never store it or attempt to use it outside of the Init lifecycle stage.

There are two types of dependencies. The most common one is where you want to use one service from another. To do this you need to use the AddService() function in the Kernel instance passed to Init().

```
func (k *Kernel) AddService(s Service) (Service, error)
```

This function accepts a single instance of the service you require. The function will then either return the instance the service that's been deployed in the kernel which you can then cast and store for later use.

If an error occurs then AddService() will return that error. You must exit the Init() function immediately, returning that error.

For example:

```
 1 package hugo
 2 
 3 import (
 4     "context"
 5     "github.com/peter-mount/go-kernel"
 6 )
 7 
 8 type Webserver struct {
 9     config *Config // Config
10 }
11 
12 func (w *Webserver) Name() string {
13     return "webserver"
14 }
15 
16 func (w *Webserver) Init(k *kernel.Kernel) error {
17     service, err := k.AddService(&Config{})
18     if err != nil {
19         return err
20     }
21     w.config = service.(*Config)
22 
23     return nil
24 }
```

The instance you pass to AddService() may not be the same one returned if that service already exists in the kernel.

The key here is the string the Name() function returns. As that must be unique it is used as the unique identifier within the kernel for the service.

As such the lookup follows the following rules:

- If it already exists then the existing entry will be returned.
- If it does not exist then the kernel will perform the following in sequence:
    1. If the new Service implements InitialisableService then it's Init() function will be called so that it can add its own dependencies which will then deploy before it.
    2. The new service is finally added to the kernel and the instance you passed to the function will be returned.

If the kernel has a service with the same name defined but of a different type then the cast will cause a panic stopping the kernel.

Peter Mount, Area51.dev & Contributors CC BY-SA

Unreferenced Service dependencies

The Kernel instance has a second function available, DependsOn(). This is rarely used but is a convenience function where you declare that you depend on one or more services to exist but don't actually want a reference to them.

```
func (k *Kernel) DependsOn(services ...Service) error
```

For example, you might have a service that requires a webserver to be running, but you don't need to directly link to it as you would be making http calls to it instead.

```
1 package pdf
2
3 import (
4     "github.com/peter-mount/documentation/tools/hugo"
5     "github.com/peter-mount/go-kernel"
6 )
7
8 // PDF tool that handles the generation of PDF documentation of a "book"
9 type PDF struct {
10     config *hugo.Config // Config
11     chromium *hugo.Chromium // Chromium browser
12 }
13
14 func (p *PDF) Name() string {
15     return "PDF"
16 }
17
18 func (p *PDF) Init(k *kernel.Kernel) error {
19     service, err := k.AddService(&hugo.Config{})
20     if err != nil {
21         return err
22     }
23     p.config = service.(*hugo.Config)
24
25     service, err = k.AddService(&hugo.Chromium{})
26     if err != nil {
27         return err
28     }
29     p.chromium = service.(*hugo.Chromium)
30
31     // We need a webserver & must run after hugo
32     return k.DependsOn(&hugo.Webserver{}, &hugo.Hugo{})
33 }
```

Here we depend on two services which we store a reference to use them, but we also require two others to be deployed and started before this service.

If an error occurs then DependsOn() will return that error. You must exit Init() returning that error.

Peter Mount, Area51.dev & Contributors CC BY-SA

3.5 - PostInit

After the Init stage the kernel enters PostInit. Here the command flags have been parsed so any reference to them are now valid. Any Service's which implements the PostInitialisableService service will have their PostInit() function called so that they can check they are in a valid state.

```
1 type PostInitialisableService interface {
2     PostInit() error
3 }
```

This stage is provided to allow services to stop the kernel if they are in an invalid state before any Service has been started.

For example, if the service created command line flags then it can check that they are valid.

A service must never call a function in another service from inside the PostInit() method, nor create any external resources like go routines or open files.

Doing so could call a service which has not yet been initialised and leave resources open if the kernel exits due to an error.

Example: Checking command line flags are valid failing if it's not been set

```
 1 type Config struct {
 2     configFile *string
 3 }
 4 
 5 func (a *Config) Name() string {
 6     return "Config"
 7 }
 8 
 9 func (a *Config) Init(k *kernel.Kernel) error {
10     a.configFile = flag.String("c", "", "The config file to use")
11 
12     return nil
13 }
14 
15 func (a *Config) PostInit() error {
16     if *a.configFile == "" {
17         return fmt.Errorf("No default config defined, provide with -c")
18     }
19 
20     return nil
21 }
```


Peter Mount, Area51.dev & Contributors CC BY-SA

3.6 - Start & Stop a service

After the Init and PostInit stages have completed the deployment order of Service's in the kernel is confirmed and the kernel then enters the Start stage.

> Both Start() and Stop() functions are optional. It is perfectly valid for a service to have only one implemented. For example a service may implement Start() and not have a Stop(). Likewise, some services only implement Stop()

Starting a service

During this stage, the kernel checks each Service in the deployment order and checks to see if it implements the StartableService interface. If it does then it calls the services Start() function to allow the service to initialise itself, creating any resources it requires.

```
1 type StartableService interface {
2     Start() error
3 }
```

For example a service can open an external Database, load a configuration file or create a Webserver.

If a service has internal types that require initialisation like maps, then those maps must be initialised with a Start() method.

> A service can call any dependency from within the Start() function as those dependencies have already been started.

If an error occurs then the Start() method must close any resources it has already created and then return an error. The kernel will then enter the Stop stage to stop any service that has already started then return that error.

Stopping a service

Once the kernel has started a service in the Start stage, it also checks to see if the service implements the StoppableService interface. If it does it will add that service to an internal stop list which contains those services which have been started and require stopping during shutdown.

During shutdown, the stop list is called in reverse order so that the last service started will be stopped first.

```
1 type StoppableService interface {
2     Stop()
3 }
```

> A service can call any dependency within the Stop() function as those dependencies are still running and will be stopped after this service has exited it's Stop() function.

> A service cannot create any new resources or go routines from inside the Stop() function. Attempting to do so may silently fail, especially if the kernel is being stopped due to a SIGINT or SIGTERM signal.

Peter Mount, Area51.dev & Contributors CC BY-SA

3.7 - Run

Between the Start and Stop stages is Run. In this stage the kernel runs through each deployed service in deployment order and if that Service implements the Runnable interface then it's Run() function is invoked.

```
1 type RunnableService interface {
2     Run() error
3 }
```

If that function returns an error then the kernel stops and enters the Stop stage. Otherwise, it continues with the next deployed service. When all services are checked it then enters the Stop stage.

Run patterns

There are two patterns for the use of this interface.

Command line tools

In this pattern you write the components of your tool as Service's with each component having dependencies to the various Service's it requires to perform some task.

Then, each one of those components implements the Run() function. It can then test to see if it should actually do something (e.g. a command line flag) and just return nil if it should not do anything, otherwise perform its task.

Once all Service's implementing Run() have done their task the tool can then shutdown.

An example of this is the tool that generates this actual document. It consists of a group of RunnableService's which perform various tasks:

1. Generates any dynamic pages like indices,
2. Run's hugo to generate the actual html site,
3. Run's chromium in headless mode to generate each PDF document based on the site.

Daemon

In the Daemon pattern only one Service implements RunnableService. When it's Run() function is called it never returns keeping the kernel and the application alive.

Dependent services might extend the Daemon service by calling a registration function in the Daemon from within their Start() function.

For example the daemon implements a webserver so the dependent services register handlers against specific paths within the website.

An example of this is the rest package included with the kernel. This implements a Webserver Service where you can register handlers against paths served by the server and implement REST actions which are provided by dependent Service's.

Peter Mount, Area51.dev & Contributors CC BY-SA

4 - Support Services

The microkernel comes with a suite of support Service's which implement common functionality provided by common third party libraries.

4.1 - Cron scheduler service

The CronService integrates the gopkg.in/robfig/cron.v2 scheduler into the microkernel.

To add the scheduler simply add it as a dependency to any of your own Service's:

Adding dependency

```
package example

import (
  "github.com/peter-mount/go-kernel"
  "github.com/peter-mount/go-kernel/cron"
)

type Example struct {
  cron *cron.CronService
}

func (p *Example) Name() string {
  return "Example"
}

func (p *Example) Init(k *kernel.Kernel) error {
  service, err := k.AddService(&cron.CronService{})
  if err != nil {
    return err
  }
  p.cron = service.(*cron.CronService)

  return nil
}
```


Peter Mount, Area51.dev & Contributors CC BY-SA

4.1.1 - Add a Schedule

Schedule a function

To have a function invoked according to a cron specification, simply call the AddFunc() function of this Service.

For example, to invoke a function in your own service every hour on the half hour you can implement:

```
func (p *Example) Start() error {
    id, err := p.cron.AddFunc( "0 30 * * * *", p.tick )
    if err != nil {
        return err
    }
    p.tickId = id

    return nil
}

func (p *Example) Stop() {
    p.cron.Remove(p.tickId)
}

// tick is called every hour on the half hour
func (p *Example) tick() {
    // Do something
}
```

Schedule a Job

As an alternative, you can add a type instead of a function as long as that type implements the Job interface.

```
type Job interface {
    Run()
}
```

To do this use the AddJob() function instead of AddFunc.

```
type MyJob struct {
}

// Run will be called every hour on the half hour
func (j *MyJob) Run() {
    // Do something
}

func (p *Example) Start() error {
    id, err := p.cron.AddJob( "0 30 * * * *", &MyJob{} )
    if err != nil {
        return err
    }
    p.tickId = id

    return nil
}

func (p *Example) Stop() {
    p.cron.Remove(p.tickId)
}
```


Peter Mount, Area51.dev & Contributors CC BY-SA

4.1.1.1 - Custom Schedules

The underlying library supports a Schedule interface which allows for custom schedules to be implemented:

```
type Schedule interface {
    // Next returns the next activation time, later than the given time.
    // Next is invoked initially, and then each time the job is run.
    Next(time.Time) time.Time
}
```

You can then create a type which implements this interface and, every time Next() is called return the time.Time when that job will execute.

To schedule the Job you then use the Schedule() function in the Service:

```
type MySchedule {
}

func (s *MySchedule) Next(t time.Time) time.Time {
    return t.Add(time.Second*5)
}

type MyJob struct {
}

// Run will be called every 5 seconds
func (j *MyJob) Run() {
    // Do something
}

func (p *Example) Start() error {
    id, err := p.cron.AddJob( &MySchedule{}, &MyJob{} )
    if err != nil {
        return err
    }
    p.tickId = id

    return nil
}

func (p *Example) Stop() {
    p.cron.Remove(p.tickId)
}
```

The underlying library provides several Schedule implementations including ConstantDelaySchedule and SpecSchedule.

Peter Mount, Area51.dev & Contributors CC BY-SA

4.1.2 - Remove an existing Schedule

The AddFunc, AddJob and Schedule functions return two values, an EntryID and an error. If error is nil then the EntryID can be stored and used to cancel the Schedule at a later time.

```
func (p *Example) Start() error {
    id, err := p.cron.AddFunc("0 30 * * * *", p.tick)
    if err != nil {
        return err
    }

    // Pointless but cancels the timer
    p.cron.Remove(id)
    return nil
}

// tick is called every hour on the half hour
func (p *Example) tick() {
    // Do something
}
```


Peter Mount, Area51.dev & Contributors CC BY-SA

4.1.3 - Cron specification

The following describes a Cron specification. This is copied from the underlying libraries documentation.

A cron expression represents a set of times, using 6 space-separated fields.

Field name	Required	Allowed values	Allowed special characters
Seconds	No	0-59	* / , -
Minutes	Yes	0-59	* / , -
Hours	Yes	0-23	* / , -
Day of month	Yes	1-31	* / , - ?
Month	Yes	1-12 or JAN-DEC	* / , -
Day of week	Yes	0-6 or SUN-SAT	* / , - ?

Note: Month and Day-of-week field values are case-insensitive.

Special Characters

***** The asterisk indicates that the cron expression will match for all values of the field.

e.g., using an asterisk in the 5th field (month) would indicate every month.

/ Slashes are used to describe increments of ranges.

For example 3-59/15 in the 1st field (minutes) would indicate the 3rd minute of the hour and every 15 minutes thereafter.

The form "*\/..." is equivalent to the form "first-last/...", that is, an increment over the largest possible range of the field.

The form "N/..." is accepted as meaning "N-MAX/...", that is, starting at N, use the increment until the end of that specific range.

It does not wrap around.

, Commas are used to separate items of a list. For example, using "MON,WED,FRI" in the 5th field (day of week) would mean Mondays, Wednesdays and Fridays.

- Hyphens are used to define ranges.

For example, 9-17 would indicate every hour between 9am and 5pm inclusive.

? Question mark may be used instead of '*' for leaving either day-of-month or day-of-week blank.

Predefined schedules

You may use one of several pre-defined schedules in place of a cron expression.

Entry	Description	Equivalent Cron entry
@yearly (or @annually)	Run once a year, midnight, Jan. 1st	0 0 0 1 1 *
@monthly	Run once a month, midnight, first of month	0 0 0 1 * *
@weekly	Run once a week, midnight on Sunday	0 0 0 * * 0
@daily (or @midnight)	Run once a day, midnight	0 0 0 * * *
@hourly	Run once an hour, beginning of hour	0 0 * * * *

Intervals

You may also schedule a job to execute at fixed intervals. This is supported by formatting the cron spec like this:

```
1 @every &lt;duration&gt;
```

where "duration" is a string accepted by time.ParseDuration.

For example, "@every 1h30m10s" would indicate a schedule that activates every 1 hour, 30 minutes, 10 seconds.

> The interval does not take the job runtime into account.
>
> For example, if a job takes 3 minutes to run, and it is scheduled to run every 5 minutes, it will have only 2 minutes of idle time between each run.

Peter Mount, Area51.dev & Contributors CC BY-SA

Time zones

By default, all interpretation and scheduling is done in the machine's local time zone. The time zone may be overridden by providing an additional space-separated field at the beginning of the cron spec, of the form "TZ=Asia/Tokyo"

> Be aware that jobs scheduled during daylight-savings leap-ahead transitions will not be run!

Peter Mount, Area51.dev & Contributors CC BY-SA