Dependency Injection in EdgeX Core Services

Geneva Planning Face-to-Face
Michael Estrin
November 2019

DELL Technologies

About Me

- Michael Estrin (m.estrin@dell.com).
- Software System Senior Principal Engineer at Dell Technologies.
- Part of the IoT Solutions Division's Platform Development Team.
- My team is responsible for
  - Dell's ongoing contributions to EdgeX.
  - Internal innovation projects built using EdgeX.
- My contributions to EdgeX
  - Common extensible bootstrapping package.
  - Simple dependency-injection container.
  - Helm Chart and Kustomize-based manifests to run EdgeX in Kubernetes.
  - Environment variable override support.
  - Refactored system management agent and Docker executor implementations.

I do not intend to take questions during this presentation.

I'm here all week and generally available on Slack.

I am happy to have individual conversations to answer questions or concerns – just seek me out.

Having said that, I'm presenting on a complex topic tailored towards code contributors.

Please speak up in the moment if I lose you.

Introduction to Dependency Injection

Disclaimer

- This is a *simplified introduction* to dependency injection.
- It is not a comprehensive overview (for that, refer to the literature).
- Basic conceptual foundation required to understand how dependency injection is being implemented in the EdgeX core services.

Dependency Injection Explained to a 5-Year-Old

“When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

“What you should be doing is stating a need, ‘I need something to drink with lunch,’ and then we will make sure you have something when you sit down to eat.”

- John Munsch

A Less Complicated Explanation

Dependency Injection is where one object supplies the dependencies of another object.

Given:

```
type Foo struct{}

func NewFoo() *Foo {
    return &Foo{}
}

type Bar struct{ foo *Foo }
```

Do this:

```
func NewBar(foo *Foo) *Bar {
    return &Bar{foo: foo}
}

func main() {
    bar := NewBar(NewFoo())
    /* ... */
```

Not this:

```
func NewBar() *Bar {
    return &Bar{foo: NewFoo()}
}

func main() {
    bar := NewBar()
    /* ... */
```

Pros and Cons of Dependency Injection

- Advantages
  - Reduces knowledge required when working on a piece of code by expressing dependencies explicitly.
  - Test code in isolation of its dependencies.
  - Quickly and reliably test situations that are otherwise difficult or impossible to test.
  - Decreases the coupling between a client and its dependency.
- Disadvantages
  - Can make code difficult to trace/read because it separates behavior from construction.
  - Typically requires more up-front analysis and development effort.
  - Encourages dependence on a dependency injection framework.

Dependency Injection != Dependency Inversion

- Dependency Injection
  - Instead of finding or making the things it needs, an object gets them from its collaborators.
- Dependency Inversion
  - One of Uncle Bob Martin's S.O.L.I.D. principles.
  - Uses shared abstractions to decouple dependencies between high-level and low-level layers.
- Often confused, these are related (both are expressions of Inversion of Control) but not the same.

Dependency Injection Frameworks

What is in a typical Dependency Injection Framework?

- A container construct capable of composing object graphs.
- The ability to automatically compose an object graph from maps between abstractions and concrete implementations (typically annotation-driven auto-wiring).
- Configuration-driven via one or more mechanisms for mapping abstractions to concrete implementations (i.e. in code, via code-generation, external XML, auto-registration, etc.).

DI Framework: Google's Wire

- Compile-time code generation
  - Embed dependencies in code; i.e. `wire.Build([dependency],…)`
  - Run the wire tool to generate code; catches missing dependencies.
  - Compile result.
- "Contract complete" but still under active development; designated as "beta."
- Requires extra step between writing code and compilation.
- Does not leverage runtime reflection.
- Idiomatic – uses generated code.

DI Framework: Uber's Dig, Facebook's Inject

- Both use reflection.
  - Runtime overhead.
  - Missing dependencies may cause runtime failure.
- Dig is a supported project.
- Inject has been archived.

Why Not Use a Third-Party DI Framework?

- Adds an external third-party dependency to the project.
- Inherent learning curve; adoption can be costly.
- Ideological issues –some are not idiomatic or in-line with Go's philosophies.
- **Frankly, they are complicated and do more than we need.**

Dependency Injection in Core Services

Goals in Adding Dependency Injection to Core

- Simplest implementation possible.
- Make dependencies explicit.
- Eliminate service-level global variables.
- Structure code to be more easily unit-testable.
- Encourage decomposition of monolithic service packages to service-specific sub-packages.

Our Simple Container Implementation

```
// Get defines the contract use to retrieve a service instance.
type Get func(serviceName string) interface{}

// ServiceConstructor defines the contract for a function/closure to create a service.
type ServiceConstructor func(get Get) interface{}

// ServiceConstructorMap maps a service name to a function/closure to create that service.
type ServiceConstructorMap map[string]ServiceConstructor

// Container is a receiver that maintains a list of services, their constructors, and their constructed instances in a
// thread-safe manner.
type Container struct {...}

// NewContainer is a factory method that returns an initialized Container receiver struct.
func NewContainer(serviceConstructors ServiceConstructorMap) *Container {...}

// Set updates its internal serviceMap with the contents of the provided ServiceConstructorMap.
func (c *Container) Update(serviceConstructors ServiceConstructorMap) {...}

// get looks up the requested serviceName and, if it exists, returns a constructed instance.  If the requested service
// does not exist, it panics.  Get wraps instance construction in a singleton; the implementation assumes an instance,
// once constructed, will be reused and returned for all subsequent get(serviceName) calls.
func (c *Container) get(serviceName string) interface{} {...}

// Get wraps get to make it thread-safe.
func (c *Container) Get(serviceName string) interface{} {...}
```

A Simple Example

```
const (
    fooServiceName = "foo"
    barServiceName = "bar"
)

type foo struct{ FooMessage string }

func NewFoo(m string) *foo { return &foo{FooMessage: m} }

type bar struct {
    BarMessage string
    Foo        *foo
}

func NewBar(m string, foo *foo) *bar { return &bar{BarMessage: m, Foo: foo} }

func main() {
    dic := di.NewContainer(
        di.ServiceConstructorMap{
            fooServiceName: func(get di.Get) interface{} { return NewFoo("fooMessage") },
            barServiceName: func(get di.Get) interface{} { return NewBar("barMessage", get(fooServiceName).(*foo)) },
        },
    )
    o := dic.Get(barServiceName).(*bar)
    fmt.Println(o.BarMessage, o.Foo.FooMessage) /* "barMessage fooMessage" */
}
```

Pros and Cons of Our Simple Container

- Advantages
  - Small (<100 loc) easy to understand implementation.
    - Abstract factory (provides concrete implementations of abstractions).
    - Supports recursive dependency usage.
    - Dependencies are instantiated only when needed.
    - Dependencies are instantiated once and only once.
  - No support for auto-wiring.
  - No separate code generation step or runtime reflection required.
- Disadvantages
  - Somewhat steep learning curve (difficult for uninitiated to understand).
  - Discontinuity between initialization and usage of container.

Quick Digression – Common Bootstrap Package

- Integrated into core services (all but security).
- Original Goals
  - Eliminate code duplication -- consolidate common code from each service's init.go into a single extensible package.
  - Unified approach to concurrency, go routine shutdown and clean up.
  - Load configuration from file once outside of retry loop.
  - Improved encapsulation.
  - Provide a path forward to remove the global variables currently defined in each core service's init.go and referenced directly throughout the service's implementation.
  - Provide a mature pattern for possible adoption by other major EdgeX components.

DI Interaction with the Common Bootstrap Package

- Our simple container holds dependencies defined during bootstrap/initialization and makes them available for injection to controller functions.
- Limited container usage.
  - Pass individual dependencies (not the container) to controller functions as parameters.
  - Controller functions pass individual dependencies as explicit parameters down call chain.
  - Encourages creation of objects (receivers encapsulating behavior via methods expressing a contract) that dependencies can be injected via a factory function.
- Addressed Golang's strong typing using helper functions that provide naming and type-asserting getters.

Caution!

Code Ahead

Code: System Management

```
func main() {
   startupTimer := startup.NewStartUpTimer(internal.BootRetrySecondsDefault, internal.BootTimeoutSecondsDefault)

   // …some flag-related code…

   configuration := &agentConfig.ConfigurationStruct{}
   dic := di.NewContainer(di.ServiceConstructorMap{
      container.ConfigurationName: func(get di.Get) interface{} {
         return configuration
      },
      bootstrapContainer.ConfigurationInterfaceName: func(get di.Get) interface{} {
         return get(container.ConfigurationName)
      },
   })
   httpServer := httpserver.NewBootstrap(agent.LoadRestRoutes(dic))
   bootstrap.Run(
      configDir,
      profileDir,
      internal.ConfigFileName,
      useRegistry,
      clients.SystemManagementAgentServiceKey,
      configuration,
      startupTimer,
      dic,
      []interfaces.BootstrapHandler{
         agent.BootstrapHandler,
         httpServer.BootstrapHandler,
         message.NewBootstrap(clients.SystemManagementAgentServiceKey, edgex.Version).BootstrapHandler,
      })
}
```

Code: Router Setup / Calls to Controller Functions

```
func LoadRestRoutes(dic *di.Container) *mux.Router {
   r := mux.NewRouter()
   b := r.PathPrefix("/api/v1").Subrouter()

   /* … */

   b.HandleFunc("/metrics/{services}", func(w http.ResponseWriter, r *http.Request) {
      metricsHandler(w, r, bootstrapContainer.LoggingClientFrom(dic.Get), container.MetricsFrom(dic.Get))
   }).Methods(http.MethodGet)

   /* … */

   return r
}
```

Code: type assertion helpers

```go
// LoggingClientInterfaceName contains the name of the logger.LoggingClient implementation in the DIC.
var LoggingClientInterfaceName = di.TypeInstanceToName((*logger.LoggingClient)(nil))

// LoggingClientFrom helper function queries the DIC and returns the logger.loggingClient implementation.
func LoggingClientFrom(get di.Get) logger.LoggingClient {
   return get(LoggingClientInterfaceName).(logger.LoggingClient)
}
```

Code: Metrics Controller Function

```
// metricsHandler implements a controller to execute a metrics request.
func metricsHandler(
   w http.ResponseWriter,
   r *http.Request,
   loggingClient logger.LoggingClient,
   metricsImpl interfaces.Metrics) {

   loggingClient.Debug("retrieved service names")

   vars := mux.Vars(r)
   pkg.Encode(metricsImpl.Get(strings.Split(vars["services"], ","), r.Context()), w, loggingClient)
}
```

Code: inside bootstrap.Run()

```
dic.Update(di.ServiceConstructorMap{
   container.ConfigurationInterfaceName: func(get di.Get) interface{} {
      return config
   },
   container.LoggingClientInterfaceName: func(get di.Get) interface{} {
      return loggingClient
   },
   container.RegistryClientInterfaceName: func(get di.Get) interface{} {
      return registryClient
   },
})
```

Code: inside service's init.go

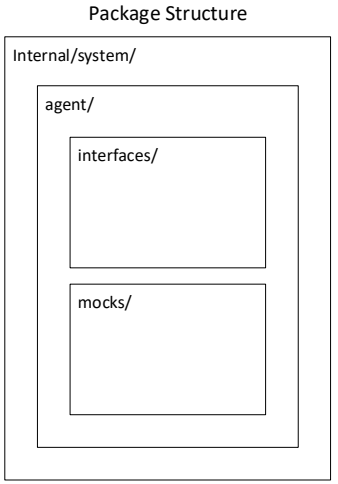
```
// add dependencies to container
dic.Update(di.ServiceConstructorMap{
   container.GeneralClientsName: func(get di.Get) interface{} {
      return clients.NewGeneral()
   },
   container.MetricsInterfaceName: func(get di.Get) interface{} {
      logging := bootstrapContainer.LoggingClientFrom(get)
      switch configuration.MetricsMechanism {
      case direct.MetricsMechanism:
         return direct.NewMetrics(logging, container.GeneralClientsFrom(get), bootstrapContainer.RegistryFrom(get),
            configuration.Service.Protocol)
      case executor.MetricsMechanism:
         return executor.NewMetrics(executor.CommandExecutor, logging, configuration.ExecutorPath)
      default:
         panic("unsupported metrics mechanism " + container.MetricsInterfaceName)
      }
   },
   container.OperationsInterfaceName: func(get di.Get) interface{} {
      return executor.NewOperations(executor.CommandExecutor, bootstrapContainer.LoggingClientFrom(get), configuration.ExecutorPath)
   },
   container.GetConfigInterfaceName: func(get di.Get) interface{} {
      logging := bootstrapContainer.LoggingClientFrom(get)
      return getconfig.New(getconfig.NewExecutor(container.GeneralClientsFrom(get), bootstrapContainer.RegistryFrom(get), logging,
         configuration.Service.Protocol), logging)
   },
   container.SetConfigInterfaceName: func(get di.Get) interface{} {
      return setconfig.New(setconfig.NewExecutor(bootstrapContainer.LoggingClientFrom(get), configuration))
   },
})
```

DI and Unit Testing

- When an object is given its dependencies (instead of instantiating them itself), it's much easier to provide substitutions to facilitate unit testing.
- Global variables are hidden dependencies and complicate writing unit tests (monkey patching anyone?). Eliminating dependencies on global variables in favor of injected dependencies simplifies unit test implementation.
- Fuji effort to implement "Operator" (i.e. Command) pattern.
  - Isolate implementation from global variables.
  - Make code more easily testable.
  - DI and subsequent elimination of service-level global variables eliminates need for "operator."

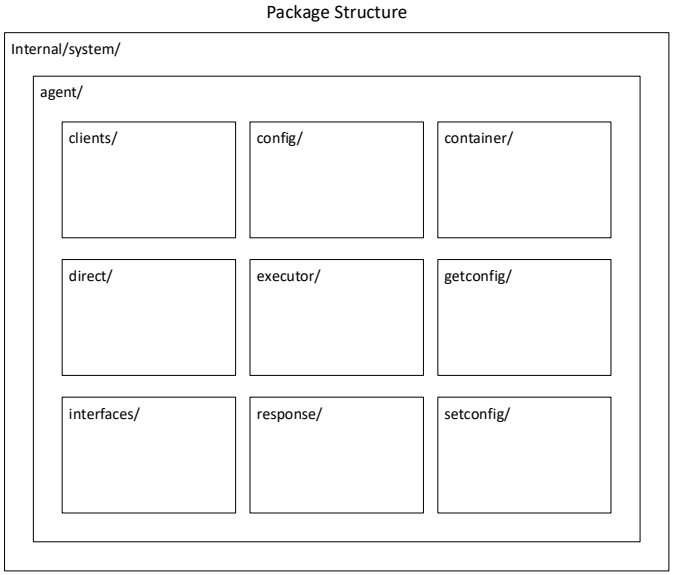
Impact on Core: System Management Agent (Before)

- Single service-level package; no encapsulation.
- Dependence on service-level global variables.
- No upfront definition of long-lived service-level dependencies; instantiated repeatedly as needed.
- Entire service implementation in agent package.



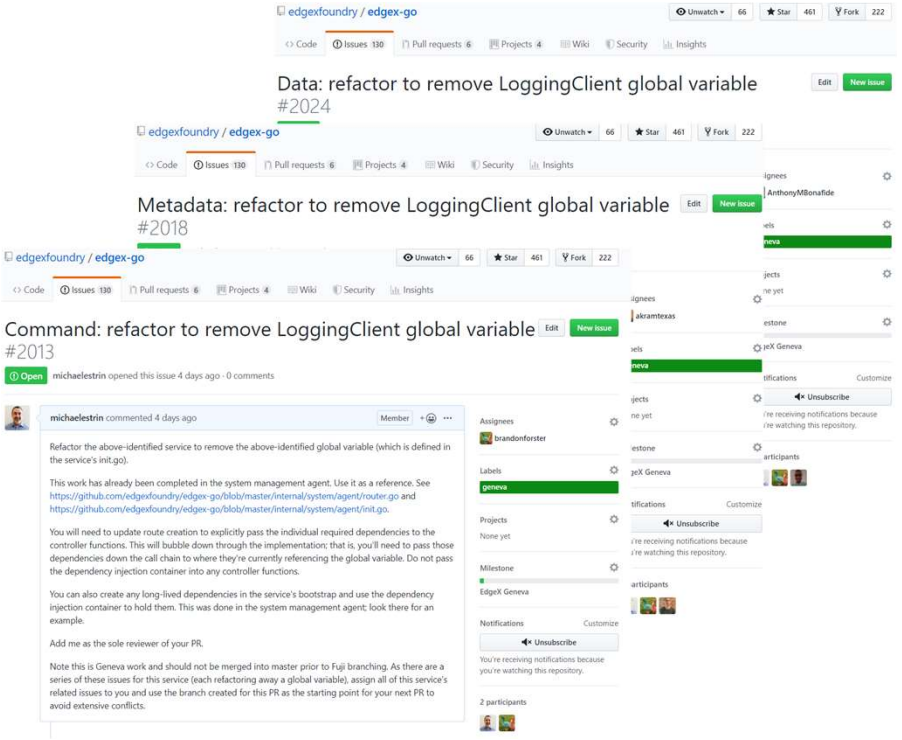
Impact on Core: System Management Agent (After)

- Multiple cohesive service-level sub-packages; encapsulation.
- No service-level global variables.
- Upfront definition of long-lived service-level dependencies; instantiated once and reused as needed.
- agent package contains only:
  - Service bootstrap (init.go).
  - Routing and controller functions (router.go).
  - Health check implementation (services.go).



Refactoring Effort is Ongoing for Geneva

- system management and notifications refactoring is complete.
- command, data, logging, metadata, and scheduler all pending simple refactoring to remove service-specific global variables.
- security-proxy-setup, security-secrets-setup, and security-secretstore-setup to first be converted to use common bootstrap package, then refactored to remove service-specific global variables.



“That’s all folks”

Thank you for your attention.

Again, I am happy to have individual conversations to answer questions, concerns, or to dive deeper into the implementation details – just seek me out.

Michael Estrin
m.estrin@dell.com
@michaelestrin in EdgeX Foundry Slack
github.com/michaelestrin
linkedin.com/in/michaelestrin

References

References

- Literature
  - Donovan, Alan A. A., Kernighan, Brian W. *The Go Programming Language*. Addison-Wesley, 2015.
  - Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.
  - Martin, Robert C. *Clean Architecture.* Prentice Hall, 2018.
  - Scott, Corey. *Hands-On Dependency Injection in Go*. Packt, 2018.
- Web
  - https://en.wikipedia.org/wiki/Dependency_injection
  - https://en.wikipedia.org/wiki/Inversion_of_control
  - https://github.com/facebookarchive/inject
  - https://github.com/google/wire
  - https://github.com/uber-go/dig


DELL Technologies