



MESHERY

Meshery Adapters

Adapting to Cloud Native Infrastructure

Design Prologue	3
Guiding Principles for Adapter Design	3
Design Goals	3
Architecture Diagram	4
Adapters	4
Repositories	5
Design Objectives	5
Meshery User Interface	5
Design Objectives	6
Tests	6
Documentation	6
Design Objectives	6
Continuous Integration	7
Development	7
Running the adapter as a container	7
Running the adapter as a process	8
Using gRPCurl to interact with the adapter	8
Refactor: Common (Adapter) Libraries	9
Forming the basis of the common adapter library	9
Refactor: Adapter parametrization	10
Refactor: Adapter configuration	11
Security	11
Misc	11
The structure design for the codebase	11
General	11
The Adapter code	11
MeshKit	12
Module Dependencies	14
Errors	14
The meshery-adapter-library	14
Purpose	14
Overview and usage	14
Package dependencies hierarchy	15

Generating cloud native infrastructure Pattern Components	16
Generating and bundling pattern components at buildtime	16
Generating pattern components at runtime	17
System Flow: cloud native infrastructure Resources	17
System Flow: Kubernetes Native Resources	17
Adapter Component Support	18

Glossary

MeshOps v1 - manifest-based operations.

MeshOps v2 (aka PatternOps) - component-based operations.

Design Prologue

Adapters are a point of pride for Meshery in the fact that Meshery connects to as many cloud native infrastructurees as it does, which is more than any other project / product in the world. Meshery supports management of more cloud native infrastructurees than any other project or product available. That said, Meshery is still egregiously missing support for a few of the more popular cloud native infrastructurees (listed below).

Of the adapters to be completed, each already has some work done toward their completion and are in a similar state. Each adapter is currently in an “alpha” state. They are in this state given that each was created from an adapter template.

Upon completion of the design objectives each adapter should achieve a “stable” state.

Guiding Principles for Adapter Design

- 1. Adapters allow Meshery to interface with the different cloud native infrastructurees, exposing their differentiated value to users.**
cloud native infrastructure projects should be encouraged to maintain their own adapters. Allowing them to expose their differentiated capabilities encourages this.
- 2. Adapters should avoid wheel reinvention, but seek to leverage the functionality provided by cloud native infrastructurees under management.**
This both reduces sustaining costs and improves reliability.

Design Goals

In support of the [Guiding Principles for Adapter Design](#), the following design goals are established. The designs in this specification should result in enabling:

1. cloud native infrastructure-specific logic should be separated from the mechanism by which Meshery communicates with each cloud native infrastructure.
2. For example, when deploying a cloud native infrastructure, seek first to leverage the native deployment methodology supported by the specific cloud native infrastructure (e.g. Kubernetes manifests, Helm chart, Operator...).
 - a. This is caveated by the need to adhere to DRY principle as much as possible, so as to reduce sustaining effort across the adapters.
 - b. Ideally, each adapter is able to leverage a deployment methodology supported by the common adapter library. Commonly, this will be:
 - i. Deployment by Kubernetes Manifest
 - ii. Deployment by Helm Chart
 - iii. Deployment by cloud native infrastructure Operator
 -
 - iv. Deployment by `<service-mesh>ctl`
 1. (There will be the occasion where use of a cloud native infrastructure's go client will be of importance as we go to leverage the same go client in MeshSync. ``istioctl`` is an example of this.)
 - c. Use go clients, not shell where possible.

Architecture Diagram

See the [Meshery Architecture](#) for visuals on how its logical constructs relate to one another.

Adapters

Adapters allow Meshery to interface with the different cloud native infrastructurees. Adapters allow Meshery to interface with the different cloud native infrastructurees, exposing their differentiated value to users. See [Adapters](#) documentation for more information.

As documented in [Extensibility](#), Meshery server communicates with Meshery adapters over gRPC. Each adapter is assigned a specific TCP port, starting from 10000. Upon connection between Meshery server and an adapter, the adapter's list of capabilities will be ingested. An adapter's capabilities are a predefined set of operations which are grouped based on predefined operation types. The predefined operation **types** are:

- Install
- Sample application

- Config
- Validate
- Custom

Adapters establish communication with Kubernetes and a specific type of cloud native infrastructure. Multiple adapters of the same type may be deployed concurrently. Although, this isn't strictly necessary for Mesher to communicate to more than one instance of the same type of cloud native infrastructure. See the [Multiple Adapters](#) guide for more information.

Repositories

Adapter repositories are created from the same adapter template (from the same code base). Adapters listed in priority order of delivery (links to repositories):

1. [mesher-kuma](#)
2. [mesher-app-mesh](#)
3. [mesher-tanzu-sm](#)
4. [mesher-maesh](#)

While this list reflects the ideal order (the priority order) in which adapters will be delivered, circumstances may change the order in which they are delivered.

Design Objectives

The designs in this specification should result in supporting each of the predefined types of operations.

1. Install
 - a. Install and delete should be functional.
 - b. Multiple deployment configurations should be offered.
 - i. Selecting cloud native infrastructure version
 - ii. Selecting method of deployment (helm, k8s manifests, CLI)
2. Sample application
 - a. Layer5 Image Hub and Istio Book Info should be available in all adapters as sample applications.
 - b. Each cloud native infrastructure's canonical sample application should be supported by the respective adapter.
 - c. Install and delete should be supported for each application.
3. Config
 - a. Support a set of predefined cloud native infrastructure configurations.
4. Validate
 - a. At least 5 validation rules should be delivered per adapter.
5. Custom
 - a. Allow any custom-defined configuration to be sent to the cloud native infrastructure.

Meshery User Interface

Each adapter should have an adapter chip (a button in the Meshery UI).

Design Objectives

Each adapter chip (button) should:

1. allow the user to invoke an ad hoc connectivity test to verify connection between Meshery server and adapter.
2. show the full name of the adapter.
3. show port assignment.
4. include the logo of their respective cloud native infrastructure.
 - a. one full-color icon and the other in grayscale (for the left side navigation menu).
 - b. logos should be in SVG format.

Tests

Each adapter should have unit tests covering its various functions. Ideally, unit tests will have 100% code coverage of adapter functionality, though minimally, must have at least 50% of code coverage.

Before creating a pull request, an adapter should be tested

- as a process ([Running the adapter as a process](#))
- as a container (mesheryctl, [Running the adapter as a container](#))
- in a Kubernetes cluster (e.g. Minikube, KinD, microk8s), see <https://github.com/layer5io/meshery/tree/master/install>

Documentation

Documentation needs to be accounted for with each new function or change of existing project behavior. Each adapter needs to have it's own documentation page, and multiple sites (meshery.io, docs.meshery.io, layer5.io) need to be updated to reflect these changes and the status of these adapters.

Design Objectives

1. Each adapter needs to have it's own documentation page (docs.meshery.io).
 - a. [Meshery Adapter for Kuma](#)
 - b. [Meshery Adapter for App Mesh](#)
 - c. [Meshery Adapter for Tanzu SM](#)

- d. [Meshery Adapter for Maesh](#)
2. Each adapter documentation page should provide users with an overview of each of the adapter's functions and sample apps.
3. Each adapter documentation page should include an architectural diagram of a deployment of the specific cloud native infrastructure.
 - a. The diagram should use the project's color scheme and look similar in design to the other diagrams within the project.
 - b. Google Draw, Google Slides, or Adobe Illustrator (or other) programs may be used to create the diagram. Existing project diagrams may be leveraged.
4. meshery.io and layer5.io/meshery should be updated to reflect adapter status.

Continuous Integration

1. Each adapter should have a CI workflow that builds and passes all checks.
2. All code should be checked using GolangCI-Lint (<https://github.com/golangci/golangci-lint>), see <https://github.com/layer5io/meshery> and <https://github.com/layer5io/meshery-consul> for example configurations.
3. CI workflows should build and push Docker images to Docker Hub
4. Security scanning

Development

Development follows the usual fork-and-pull request workflow described [here](#), see also [GitHub Process](#). On forking GitHub deactivates all workflows. It is safe and good practice to activate them such that the code is validated on each push. This requires that branches filter for "on push" is set to '**' to be triggered also on branches containing '/' in their name. The actions are parameterized using secrets (see [Build & Release Strategy](#)). The Docker image is only built and pushed to Docker Hub if a tag is pushed and the corresponding authentication information is configured. The only secret that should be set in each fork is GO_VERSION, specified in [Build & Release Strategy](#), otherwise, the corresponding action's default version is used.

Each commit has to be signed off, see [Contributing Overview](#).

Running the adapter as a container

Testing your local changes running as a container can be accomplished in two ways:

1. **Define the adapter's address in the UI:** Unless the running container is named as specified in the docker-run target in the Makefile, the container has to be removed manually first. Then, run `make docker` followed by `make docker-run`. Then, connect to the adapter in the UI in "Settings>cloud native infrastructures" using `localhost:<port>` if the meshery server is running as a binary, or `<docker IP address>:<port>` if it is running as a docker container.
2. **Using mesheryctl:** In `~/.meshery/meshery.yaml`, change the tag specifying the image of the adapter to "latest". Run `make docker`, followed by `mesheryctl system start --skip-update`. This assumes `mesheryctl system start` has been executed at least once before.

Running the adapter as a process

Another way to test your local changes is to run the adapter as a process. To do this, clone the meshery repository, and start meshery using `make run-local-cloud`. Start the adapter from your IDE, or by executing `make run`. Then, in the meshery interface, add the adapter using "localhost:<PORT>".

Using gRPCurl to interact with the adapter

As the adapter is exposing a gRPC-API, [gRPCurl](#) can be used to interact with it. [gRPCurl](#) is a command-line tool that lets you interact with gRPC servers. It's basically curl for gRPC servers. Nic Jackson explains it really well in [this video](#). If you're using gRPCurl, you don't need to start meshery at all.

Start your adapter, preferably in debug mode, from your favorite IDE. In the following examples, it is assumed to run on "localhost:10002". It is also assumed that gRPC reflection is enabled in the adapter (by importing `google.golang.org/grpc/reflection` in `main.go` and registering the server "s" using `reflection.Register(s)`). If reflection is not enabled, add `-import-path ./meshes/ -proto meshops.proto` to all gRPCurl commands, which assumes that "meshops.proto" has been downloaded using `make protoc-setup`.

Some examples (`--plaintext` means no TLS when connecting to server):

Get a list of all services:

```
grpcurl --plaintext localhost:10002 list
```

Get a list of all functions in meshes.MeshService

```
grpcurl --plaintext localhost:10002 list meshes.MeshService
```

Get all supported operations:

```
grpcurl --plaintext localhost:10002 meshes.MeshService.SupportedOperations
```

Describe meshes.CreateMeshInstanceRequest and get a message template:

```
grpcurl -plaintext -msg-template localhost:10002 describe \
  meshes.CreateMeshInstanceRequest
```

yields a message template:

```
{
```



```
"k8sConfig": "",
"contextName": ""
}
```

k8sConfig takes a base64-encoded k8s configuration, all on one line. This can be achieved using `kubectrl config view | base64 -w 0 -`. Save the message in a file, e.g. message.json. Then, create a new k8s-client in the adapter using:

```
cat message.json | grpcurl --plaintext -d @ \
localhost:10002 meshes.MeshService.CreateMeshInstance
```

Now, you can use `meshes.MeshService.ApplyOperation` to apply operations, like deploying a cloud native infrastructure or a sample application.

Refactor: Common (Adapter) Libraries

- See <https://github.com/layer5io/meshkit> (**note:** this was renamed from layer5/gokit on 24.10.20)
- See <https://github.com/layer5io/meshery-adapter-library>,
 - see [The meshery-adapter-library \(documentation\)](#)

The current architecture of adapters is such that they each have some redundancy between them. A subset of their functionality, functions universal to all adapters, are candidates for centralization into a shared adapter package. Examples of this redundancy:

1. Downloading a cloud native infrastructure release from GitHub and provisioning to Kubernetes
2. Installing a cloud native infrastructure using Helm
3. Splitting YAML file (?)
4. ...

The creation of a central, shared package with common functions, and the subsequent displacement of those then redundant functions in the existing adapters is something that can be done before or after the adapters above are created.

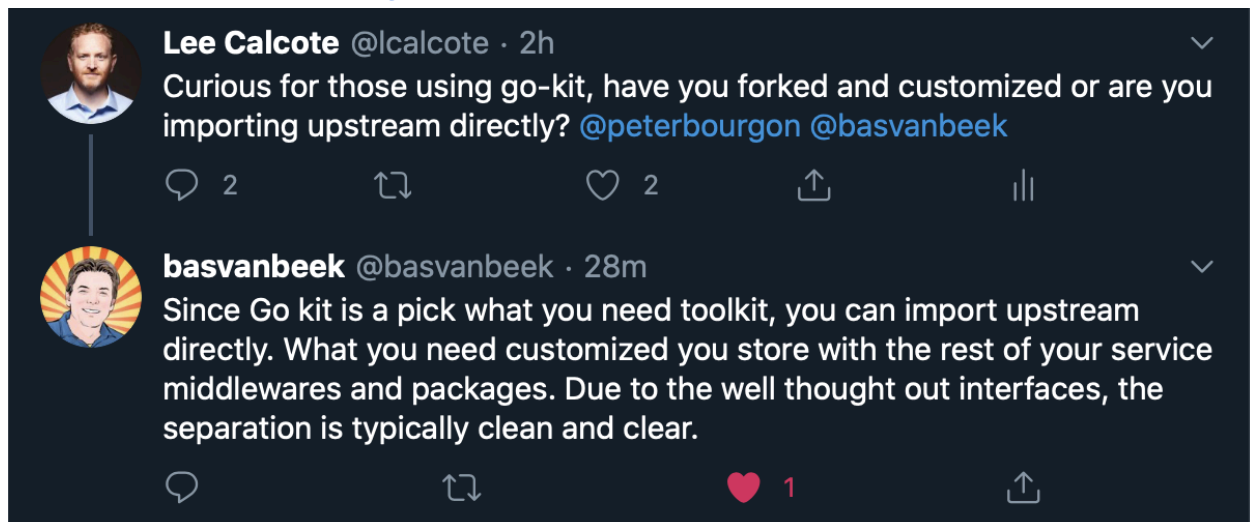
Forming the basis of the common adapter library

Should we fork [go-kit](#) as the basis of the Layer5's meshkit? Layer5 needs to customize some of go-kit's packages.

If we do fork...	If we do not fork...
- Incur sustaining overhead	+ Avoid sustaining overhead
- Challenging to update from upstream go-kit	+ Easy to update from upstream go-kit

- Learn Layer5's meshkit as a mandatory for adapter development	- Learn upstream go-kit as a mandatory for adapter development
+ Can customize the packages	- Cannot customize the packages
- Performance overhead due to wide scoped packaging	- Issues with versioning, the control is with the upstream.
Test cases maintained	Have to write test cases and coverage for every feature we implement.
Modified functions behave differently than expected, this might be quite confusing.	+ Functions behave as expected
Changes are implemented in the fork, not a clear separation, developers have to go through change history/diff to figure out what has changed.	+ Clear separation between go-kit and Layer5 customization
If we think we need to fork because the library doesn't support e.g. injection of necessary custom functionality, then maybe another library should be considered.	+ Well designed library with clean extension points.
Tight coupling to go-kit	+ Loose coupling

See [this tweet](#) from one of the [go-kit](#) maintainers -



Lee Calcote @lcalcote · 2h
Curious for those using go-kit, have you forked and customized or are you importing upstream directly? @peterbourgon @basvanbeek

basvanbeek @basvanbeek · 28m
Since Go kit is a pick what you need toolkit, you can import upstream directly. What you need customized you store with the rest of your service middlewares and packages. Due to the well thought out interfaces, the separation is typically clean and clear.

Refactor: Adapter parametrization

Improving our adapter and server design to account for tracking cloud native infrastructure Version #. This attribute would be sourced within the adapter and displayed in Meshery server.

E.g. using Viper, where the default version is defined in the adapter code, but can be overridden through command line arguments or environment variables. Depending on a specific mesh, this could mean several parameters, e.g. for Consul Helm chart version (0.24.1) as well as specific version of the mesh binary (e.g. 1.8.0, 1.8.1).

Mesh parameterization could be exposed through a new service function MeshInfo, returning some fixed fields (name etc) and a map for arbitrary data.

Related: SupportedOperations might return a list with supported parameters/flags with default values, that can be used in ApplyOperation, and the UI.

Refactor: Adapter configuration

Using Viper (<https://github.com/spf13/viper>) by default.

Security

- code security check using gosec (golangci-lint)
- container security scanning in CI/CD?
- secure RPC traffic
- handling kubeconfig
- handling secrets

Misc

(don't really know where to put points mentioned here, move them to appropriate section later)

- **kubeconfig**. kubeconfig with the correct context is sent to the adapter using the CreateMeshInstance call. Is it also written back to the filesystem?

The structure design for the codebase

General

Circular dependencies between packages should be avoided.

Related presentation(s): [Common Libraries](#)

The Adapter code

The code hierarchy is divided into several layers. This is based on domain-driven-design architecture.

- The top most layer is the **`Service`** layer, which provides the server specific implementations and configurations.
- The middle layer will be the **`Middleware`** layer, which holds all the dependencies and custom integrations to support the application. Middleware is also known as a decorator.
- The bottom most layer is the **`Handler`** layer, which would have the core business/adapter logic code implementations.

MeshKit

The code hierarchy is pluggable and independent from one another. There can be **N** number of packages depending upon the use case.

- **`errors/`** - holds the implementations and the error handlers and error codes which are used across projects.
- **`logger/`** - holds the implementations of logging handler and custom attributes to add if any.
- **`utils/`** - holds all the utility functions that are specific to mesher projects and are to be used generically across all of them.
- **`tracing/`** - holds the implementations of tracing handlers with different tracing providers like jaeger, newrelic, etc.

Each package inside a meshkit is a handler interface implementation, the implementation could be from any third-party packages or the [go-kit](#).

How are we tracking adapter version #?

The adapter version is tracked in the Service layer described above. There is an object corresponding to this layer that holds the adapter version. The value of this version would come from the config.

layer5/meshkit (**L-1** in the figure below) should only contain general utility code that can also be used in code not (directly) dealing with adapters. If there exists such code or in the future? This means that layer5/meshkit should maybe not contain k8s etc code and imports, as these are used exclusively in adapters.

How does the adapter

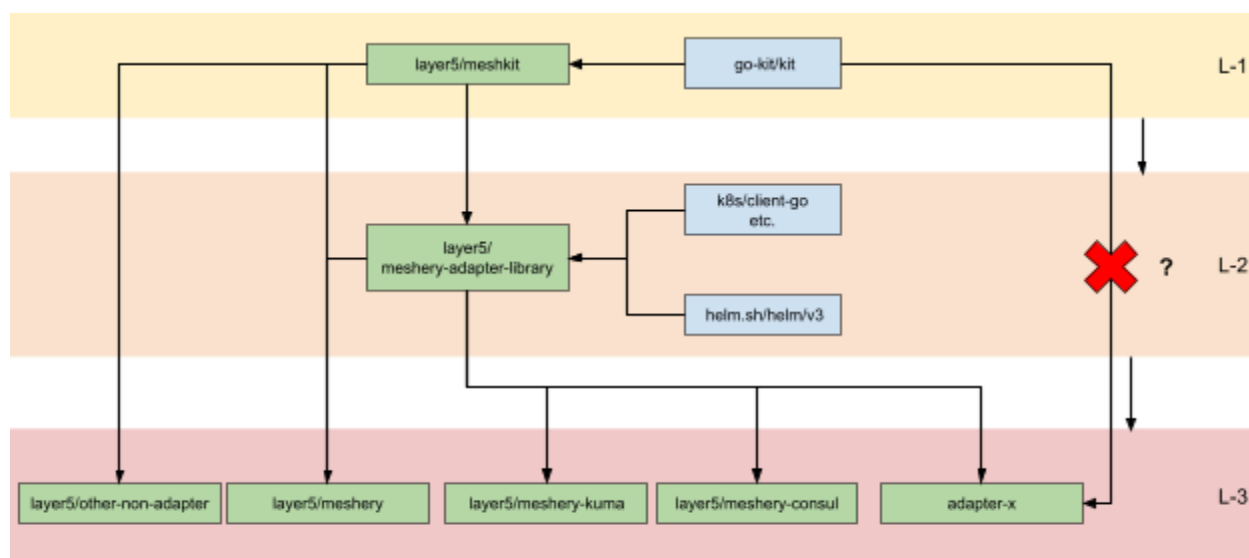
What goes into the common adapter library and what is cloud native infrastructure specific?

The following would go under adapter library (**L-2**):

- Implementations of the smi tooling library
- Streaming handlers
- Protocol buffers and client handlers
- Deployment implementations like helm, sample applications, etc

cloud native infrastructure specifics are (L-3):

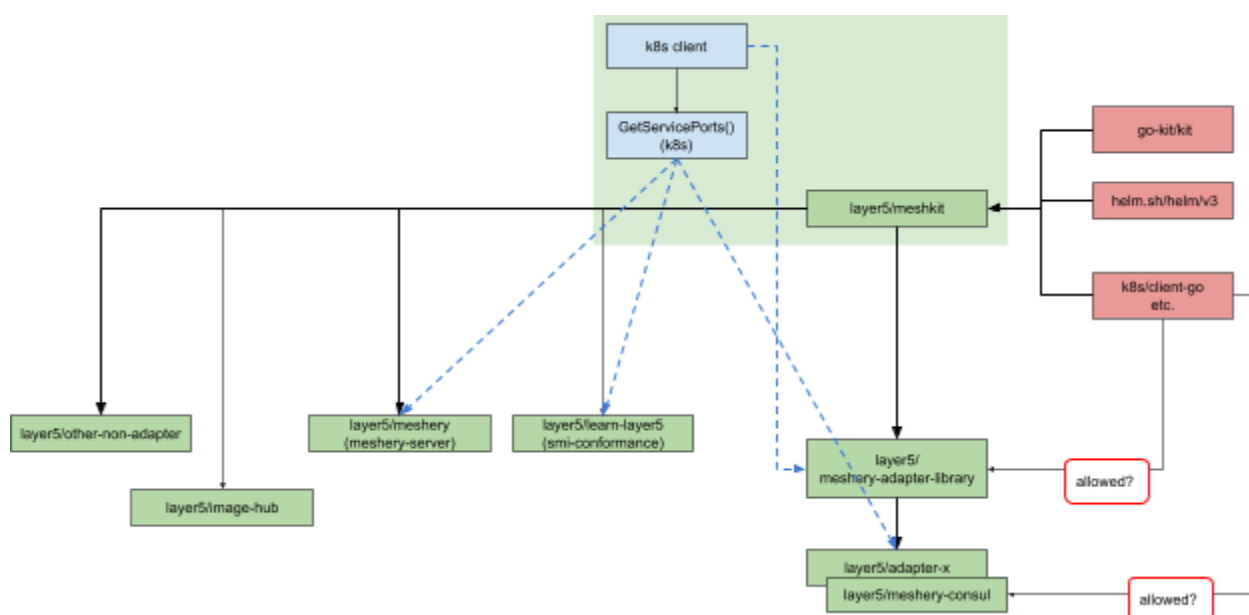
- Deploy and delete operation logics
- Configurations, eg: port number, labels, annotations, etc



Legend:



Module A can be imported into module B
modules from "higher up" can be imported "downwards"



Module Dependencies

TODO: Overview over module dependencies and hierarchy, e.g. similar to the diagram above, or as a table. This helps to decide where (common) code should be located. Also, it helps to avoid (and detect) circular dependencies between modules, which has to be avoided.

Errors

The custom error object that has been planned consists of several attributes that makes the error much informative and yet easier to maintain across projects. See the [Messaging System and Notification Center](#) document.

The meshery-adapter-library

This section contains a high level overview of the meshery-adapter-library, its purpose and architecture. For details, the reader is referred to the documentation and the code in the repository.

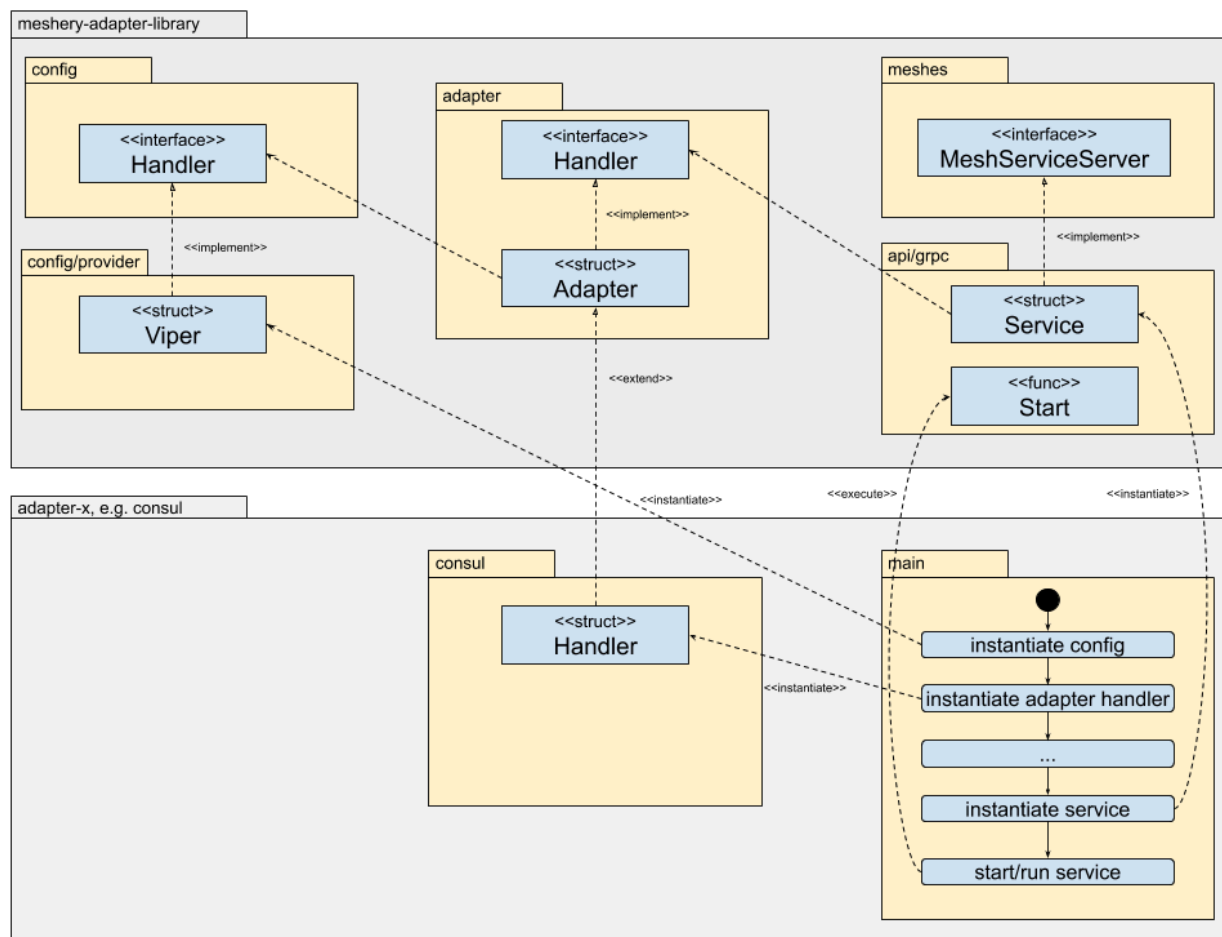
Purpose

The main purpose of the meshery-adapter-library is to

- provide a set of interfaces, some with default implementations, to be used and extended by adapters.
- implement common cross cutting concerns like logging, errors, and tracing
- provide a mini framework implementing the gRPC server that allows plugging in the mesh specific configuration and operations implemented in the adapters.
- provide middleware extension points

Overview and usage

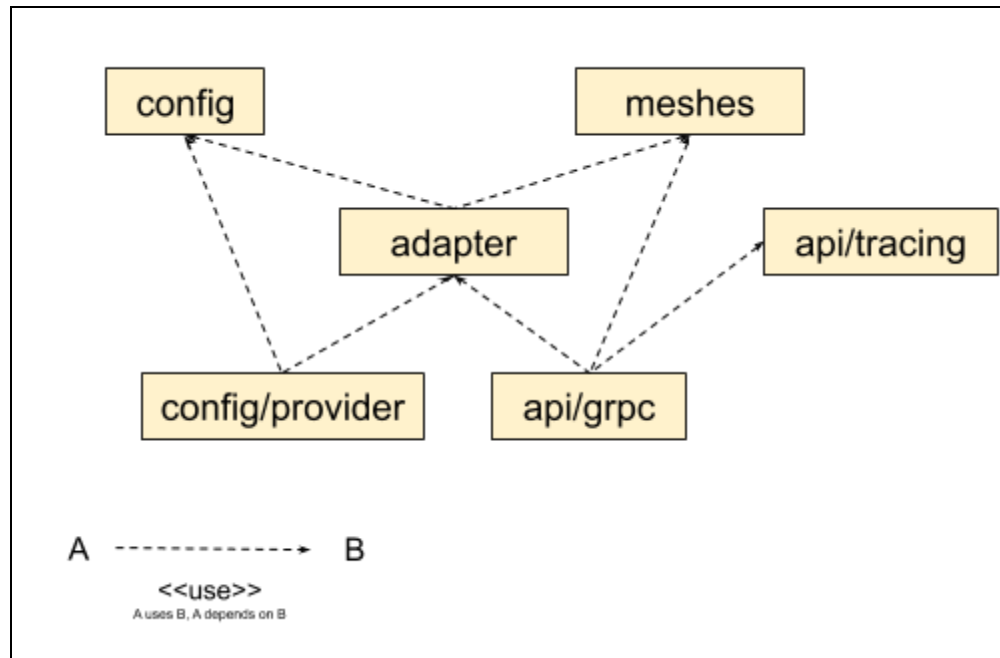
The library consists of interfaces and default implementations for the main and common functionality of an adapter. It also provides a mini-framework that runs the gRPC adapter service, calling the functions of handlers injected by the adapter code. This is represented in an UML-ish style in the figure below. The library is used in the Consul adapter, and others will follow.



Package dependencies hierarchy

A clear picture of dependencies between packages in a module helps avoid circular dependencies (import cycles), understand where to put code, design coherent packages etc.

Referring to the figure below, the packages `config` and `meshes` (which contains the adapter service proto definition) are at the top of the dependency hierarchy and can be used by any other package. Thinking in layers (L), `config` would be in the top layer, L1, `adapter` in L2, and `config/provider` in L3. Packages can always be imported and used in lower layers.



Generating cloud native infrastructure Pattern Components

For more context on cloud native infrastructure patterns and their components, see:

[Meshery and Service Mesh Patterns](#)

Every adapter should have OAM components for each cloud native infrastructure version specific resource so that it can register its capabilities with the Meshery server and operations can be performed using patterns.

Generating and bundling pattern components at buildtime

- The CI process of each adapter should generate pattern components as new versions of a cloud native infrastructure are released and upload them to the adapter repo.
- They would then be packed in the adapter's container image during the build process. This will ensure that the adapter is ready to use out of the box in an air gapped environment.
- Tentatively, this is being done using bash scripts in the CI process. However, in the long term we'll be generating components by using adapter's runtime generation capabilities in CI as well. This will ensure components have a single source of origin.

Generating pattern components at runtime

System Flow: cloud native infrastructure Resources

Adapters facilitate generation of pattern components on-demand. Integrated in each adapter are generic component generation utilities imported from MeshKit.

- A user would only specify the appropriate cloud native infrastructure version and then summon the operation for generating components.
- Adapter developers would ideally recognize and specify reliable sources for any of those manifests in the adapter's code. Meshkit would take care of fetching them.
- Also adapter developers would be having options for specifying jsonpath filters to be used with the json schema utility. If they provide none, default filters based upon observation from various meshes would be used. Sample jsonpath filter which tends to apply over multiple cloud native infrastructure CRDs:

- `$[?(@.kind=="CustomResourceDefinition" && @.spec.names.kind=="$t")].openAPIV3Schema.properties.spec`

- What all manifests are we [fetching and their source](#):

- CRDs: GH repo
 - a. Helm charts:
 1. Helm repo
 2. GH repo
 - b. What else?

NOTE: We would also keep GH release bundles as a universal fallback method for the above resources.

- c. cloud native infrastructure resources and K8s native resources:
 1. Fetch the json schema utility on demand
 2. Provide it cloud native infrastructure CRDs
 3. Extract OpenAPI schema from those CRDs using appropriate jsonpath filters.
 4. Extract other metadata for generating OAM workloads using some other jsonpath filters.

System Flow: Kubernetes Native Resources

Use Case: Extracting metadata for pattern components from Kubernetes

Primary Actor: Mesher Server

Scope: A single Kubernetes cluster

Level: System-wide setting

Story: ashishjaitiwari15112000@gmail.com you can regurgitate the user story here.

Preconditions:

1. Mesher has appropriate credentials to a Kubernetes cluster.

2.

Acceptance Tests:

1. The following Kubernetes objects are registered as capabilities in Meshery.
2. Each capability is associated with the respective Kubernetes context.
3. The Kubernetes context has a version # (e.g. v1.22.0); consequently, every registered capability has an associated version number.

Behaviors:

1. Meshery should register duplicative capabilities of same object type and K8s version # between contexts. Deletion of one context and its registered capabilities should have no effect on another connected context.
2. Meshery Server does not extract cloud native infrastructure pattern components - only Kubernetes pattern components.
3. Upon successful connection to Kubernetes context, but on failure to generate components, display Warning level event in Meshery UI Notification Center.
 - a. Include probable cause of insufficient service account / cert permissions.

Triggers:

1. Initial connection to Kubernetes cluster. Could be on Meshery Server boot or on user upload of one or more contexts.
2. Does not trigger when ad hoc connectivity tests are run.

Basic flow (actor: meshery server):

1. Upon initial connection to a Kubernetes context, retrieve K8s objects (json documents) from kube-api using client-go.
2. Use MeshKit function to create schemas and definitions from the json documents.
3. Use `core.RegisterWorkload` function to register these workloads in Meshery Server.
- d. Native Meshery resource: ?

Adapter Component Support

MESH	MANIFESTS	RETRIEVAL MECHANICS	RUNTIME	BUILDTIME	MANIFESTS	EARLIEST VERSION	RANGE OF VERSIONS	# OF VERSIONS
Istio	GH	Using Manifests in meshkit function	✓	✓	✓	1.6.0	1.6.0 - 1.11.4	50
OSM	GH	Using helm in meshkit function	✓	✓	✓	0.9.2	0.9.2 - 0.10.0	3
Kuma	GH	Using helm in meshkit function	✓	✓	✓	1.2.2	1.2.2, 1.3.0-1.3.1	3
Linkerd	GH	Using crds from github repo	✓	✓	✓	2.10.2	2.10.2 - 2.11.0	2
Consul	GH	Using crds from github repo	✓	✓	✓	1.8.4	1.8.4-1.10.0	10
Traefik Mesh	GH	Using helm in meshkit function	✓	✓	✓	?	?	?

App Mesh	GH	Using crds from github repo	✓	✓	✓	1.4.1	1.4.1	1
NGINX	GH	Using helm in meshkit function	✓	✓	✓	1.2.0	1.2.0-1.2.1	2
Cilium								
NSM	GH	?	✗	✗	✓	?	?	?
Citrix	?	-	-	-	-	-	-	-
VMware Tanzu	?	-	-	-	-	-	-	-
Octarine	N/A	-	-	-	-	-	-	-