Carthage Core Documentation

Sam Hartman

May 14, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

Carthage is an **Infrastructure as Code (IAC)** framework. Carthage provides models for infrastructure concepts such as `machines`, `networks`, and domains or groups of machines. There are concrete implementations of these models including containers and virtual machines.

Carthage allows experts to quickly construct infrastructure from a Carthage layout. Infrastructure can be real, virtual, or a mixture. Often the same layout is used to produce both real and virtual infrastructure. In the core of Carthage, when we have had to choose between power and efficiency for experts or making things easy for beginners, we have chosen to empower experts. Carthage evolved in part out of frustrations with other IAC frameworks. On the surface these other frameworks were easy to understand, but they lacked the power to express real world environments. We found ourselves writing a tool to compile domain-specific models into inputs for these other frameworks. Rather than combining the complexity of our precompiler with the limitations imposed by other systems, we focus on providing a flexible, powerful framework.

While parts of Carthage are expert tools, Carthage works to keep simple tasks simple. We strive to make it easy to make simple changes to layouts. We also strive to allow complexity to be compartmentalized. It might take a Carthage expert to design a reusable template for describing networking for a complex layout that can be deployed both on virtual hardware and on real switches. However, anyone who knows Ansible or some other supported devops tool can contribute to a Carthage application. Adding an Ansible role or playbook to a Carthage machine is easy.

Many IAC systems focus on building containers and micro-services. By focusing on these environments, significant simplicity is gained. Some of the Carthage use cases focus on modeling existing architectures that are not micro service based. Many Carthage layouts do involve at least some portion that is containerized or micro service based. However Carthage can also model other architectural approaches.

Carthage permits layouts to be described in a declarative manner when that makes sense. There are many advantages to declarative descriptions: it is possible to introspect the description, and even to compare the state of real hardware or a cloud environment to the description. However the real world is rarely that simple. As an example, a layout may wish to create a machine for each developer in an active directory group. So as part of building the layout, Carthage needs to query the directory server. Such a process cannot be fully declarative. Things get even more complex when the same layout is responsible for building and maintaining the directory server itself. Supporting such configurations is a design goal of Carthage.

1.1 Testing Use Case

One of the motivating applications for Carthage is to provide a realistic test environment for a distributed product that includes hosted and cloud components. In this mode, all machines are realized in virtual environments (either as containers or VMs).

The goal is to test the product as well as the IAC infrastructure used to install and ship hosted components along with infrastructure to maintain the cloud service.

The testing environment needs to be entirely isolated from the production environment and cloud services.

To accomplish this, a Carthage test layout is constructed. This layout starts by building initial OS images. Then it bootstraps some of the components from the IAC layouts used to install real hardware, re-targeted at virtual environments. This is used to set up the cloud services. IAC code used to maintain the production services is targeted to set up the provisioning and inventory cloud services. Data is copied in from an export provided by the real cloud service. The data is massaged to account for a few differences where the test environment does not fully replicate real hardware. (As an example, connections to the test network are more uniform than connections to networks around the world.)

Then the production IAC code is run in the virtual environment to bring up and provision virtual analogues of real equipment and cloud services. Tests are run against these systems and the results reported.

Several features emerge from this test case:

- Carthage supports a multi-stage layout. Until the virtual instance of the provisioning database becomes available, Carthage doesn't even know what virtual systems it will ultimately build.
- Carthage needs to be able to interact with complex networks with potentially overlapping addressing plans. The test network topology directly mirrors the production topology; many of the key services are at the same address. Carthage needs to make sure that isolation is maintained. Carthage needs to function even when the test network is embedded entirely within the production network. However in limited cases, for example importing the data export, connectivity is required.
- Carthage needs to have facilities to reach into the virtual environment and explore test failures, both for graphical and non-graphical sessions.

1.2 Customer Build Use Case

Another motivating application was providing a way to incrementally build out equipment shipped to customer sites. During the sales process, a potential configuration is built.

Carthage simulates this configuration in a virtual environment. This allows field engineers to validate the configuration and potentially to plan for the site visit. In cases where the simulation is good enough, the customer may be able to get a better picture of how the product will help them.

Often it is necessary to build and configure systems before deploying them at the customer site. During this process, virtual components from the pre-sales simulation are replaced with real components that will eventually be shipped to the customer. As the ship date approaches then more of the system uses real hardware.

Several requirements emerged from this use case:

- Support for combining real and virtual equipment in the same layout and changing this over time
- Support for integrating with provisioning and asset management systems to define what components are present in a layout
- Ability to view and interact with virtual components of a layout in a manner similar to how they will work once deployed

1.3 Reproducing Problems Use Case

When a complex set of related equipment is shipped to a customer, it is not feasible to keep a duplicate set of equipment on which to reproduce problems. Even if spare inventory is available to recreate and configure the environment, doing so takes time and space.

Carthage can be used to reproduce customer environments in enough detail to reproduce problems. The approach is similar to the build out use case. Initially a fully virtual simulation of what is set up at the customer is used. Real components are substituted in until the problem can be reproduced. (For pure software problems, real components may not be needed.)

The same approach can be used for validating additions to a customer environment.

1.4 Cyber Training Use Case

In order to defend networks, defenders need a high quality training environment. This environment needs to be isolated from the defended network (and often entirely from the Internet):

1. Attacks in a training exercise must not affect production systems.
2. In some cases, defense strategies are confidential and there are concerns that attackers might be able to observe them if the environment is not isolated.

Carthage generates a cyber range similar to some defended system. In some cases, for example when defended networks have industrial automation, real components may be integrated into the range where purchasing a physical device for the training is more effective than creating a virtual model. Defenders and attackers access resources within the range using desktop virtualization tools.

Producing ranges using an IAC strategy has a number of benefits:

- Ranges can easily be reset to known conditions.
- Over time the fidelity of the range improves as more IAC components become available.

Cyber training requires relatively high-fidelity simulation of the defended system. Using micro services and containers in a simulation is desirable if that is what the actual defended system uses. But for many defended systems, a mix of virtual machines is required. For some attacks such as attacks on firmware, even normal virtual machines may not provide an accurate enough simulation. Carthage has not yet been used heavily in such environments.

CHAPTER

TWO

API DOCUMENTATION

2.1 Dependency Injection

Often in developing IAC systems, the part of the system that needs to know something about the environment is separated from the part of the system that can make that decision. For example:

1. Cloud resources are typically placed in some folder, region or tenancy. The resources are defined in a *layout* focused on describing how to create the resources. The information about where to put them is in a part of the code focused on instantiating those resources.
2. Depending on how it is being used, sometimes a layout may be instantiated on virtual machines (or containers) and sometimes on real hardware. As an example in the *Testing Use Case*, the entire layout may be virtualized. However, in the *Customer Build Use Case*, the same layout may be partially or completely built on real hardware. As above, the layout is focused on describing the resources and how to instantiate them. The application using the layout knows what hardware will be used and where virtual components will live.
3. A layout might contain a template for building a work group. This builds a network, router, and a series of workstations. These need to be connected to the broader layout. The template needs to know where to connect and needs to know details such as the names of constructed workstations. Other parts of the layout will instantiate the template multiple times.

2.1.1 How Dependency Injection Works

An object such as a function or class declares dependencies using `inject()`

```
@inject(connect_to = Network)
def build_workstation(name, *, connect_to: Network):
  #Build a workstation called name and connect to connect_to
```

The *inject* decorator effectively says that the decorated object/function needs some parameter, but the direct caller is unlikely to be able to supply the value. An object decorated this way is said to have dependencies that need to be injected. Such objects can be called normally:

```
build_workstation(name = "ws1", connect_to = some_network)
```

Doing so requires the caller to provide all the dependencies. Instead, it is more common to use a `Injector` to call an object that requires dependencies:

```
injector(build_workstation, name = "ws1")
```

The `injector` injects (supplies values for) the dependencies. The *name* argument of *build_workstation* needed to be supplied by the caller, because it was not marked as an injected dependency. However, *connect_to* can be injected by

the injector if the injector or one of its parents provides a dependency for `Network`. An injector can be instantiated with such dependencies:

```
injector = Injector(parent)
injector.add_provider(some_network)
```

This sets up an injector which inherits dependencies from an existing injector and then adds an existing network to the injector. Most injectors eventually inherit from `carthage.base_injector`.

2.1.2 Injectors and Classes

`Injectable` is a base class for objects that need dependencies injected:

```
@inject_autokwargs(this_network = Network)
class NeedsNetwork(Injectable):

    def do_something(self):
        print(self.this_network)
```

The `inject_autokwargs()` decorator works like *inject* except that it raises `TypeError` if the parameter is not specified either by a caller or an injector. `Injectable.__init__()` examines dependencies associated with the class and sets an attribute on *self* capturing any provided dependency.

2.1.3 Injection Keys

Sometimes a class may require more than one of a given kind of object. Often an injector may have more than one of a given type of object available to provide dependencies. `injectionKey` combines a type with a set of named constraints to select which object is required:

```
@inject_autokwargs(
    outside_network = InjectionKey(Network, role="outside"),
    inside_network = InjectionKey(Network, role = "inside"))
class Firewall(Injectable):
    # outside_network and inside_network will both be set.
```

Then other code can set up an injector:

```
injector.add_provider(InjectionKey(Network, role="outside"), outside_network)
injector.add_provider(InjectionKey(Network, role="inside"), inside_network)
```

Although it might be more common for the outside and inside network to be set up in different injectors:

```
# outer_injector already provides InjectionKey(Network, role="outside")
# Provide a firewall for foo.com, bar.com and baz.com
for org in ("foo.com", "bar.com", "baz.com"):
    org_injector = outside_injector(Injector)
    org_network = org_injector(Network, name = f"{org} internal network")
    org_injector.add_provider(InjectionKey(Network, role="inside"), org_network)
    org_injector.add_provider(Firewall)
    org_firewall = org_injector.get_instance(Firewall)
```

2.1.4 API Reference

2.1.5 Events

The dependency injection system emits several `events`.

add_provider
: Emitted when `carthage.dependency_injection.Injector.add_provider()` is called. Dispatched to all the keys that the dependency will satisfy. The target of the event is the object providing the dependency, typically an uninstantiated class. Also dispatched to `InjectionKey(Injector)` as a wildcard. Contains the add_provider parameters as well as *other_keys*, indicating other keys by which this dependency will be provided.

dependency_progress
: Emitted whenever an instantiation makes progress (for example resolving a `AsyncInjectable` or calling a coroutine. The target is a `carthage.dependency_injection.InstantiationContext`. The value can be obtained with the *get_value* method. This event is dispatched to all the keys that the *add_provider* event would be dispatched to.

dependency_final
: Emitted whenever an instantiation finalizes (async object is ready for example). Same target and keys as *dependency_progress*.

2.2 Setup Tasks

2.3 Networking Documentation

2.3.1 Network Events

The following events may be generated by the networking system resolved

Emitted toward `InjectionKey(NetworkConfig)` when a `carthage.network.NetworkConfig` is resolved. The target is the network config that has resolved;

param other_futures
: When both sides of a link are configured at the same time, the other side of the link cannot be resolved until its `carthage.Machine` is ready.In this case a future is recorded and included in the event.

This event is typically used to collect the set of futures for other_links. When all these futures are *done*, then all effects from resolution of the `NetworkConfig` have been realized.

public_address
: Emitted toward `InjectionKey(NetworkLink)` and `InjectionKey(NetworkLink, host=model.name)` when some part of the system becomes aware of the public address of a link behind a NAT. A *public_address* property is set on the link's *merged_v4_config* prior to emitting the event.

2.4 Machines: Systems under Control or Simulation

2.4.1 Containers

2.4.2 VMs

2.4.3 Hardware Configuration

VMs and cloud instances will look for the following properties in a `AbstractMachineModel` to configure hardware:

cpus
: The number of CPUs on the virtual machine

memory_mb
: The amount of memory in megabytes

disk_sizes
: A sequence of disk sizes for primary and secondary disks. Provided in GiB.

disk_config
: A sequence of dicts configuring primary and secondary disks. The only key defined at this level is *size*, the size of the disk in GiB. If *disk_config* is provided, *disk_sizes* is ignored. The intent of *disk_config* is to permit MachineImplementation specific configuration of disks. Consult the specific machine implementations for details.

nested_virt
: Boolean indicating whether to allow nested virtualization

2.5 Open Container Interface Support

2.5.1 The OCI Layer

2.5.2 Podman

2.6 Images and Volumes

Images are handled for `containers` by `image.ContainerVolume` and VMs with `image.ImageVolume`.

2.7 SSH and Rsync Support

2.8 File Copying and Insertion

CHAPTER

THREE

MODELING LAYER

Classes like `carthage.Network` and `carthage.machine.AbstractMachineModel` provide an abstract interface to infrastructure resources.

The modeling layer provides a generally declarative interface for defining and configuring such models. The modeling layer provides a domain-specific language for describing models. Python metaclasses are used to modify Python's behavior in a number of ways to provide a more concise language for describing models.

3.1 A Simple Model

Listing 1: A simple modeling layout to define a machine

```
# Copyright (C) 2021, Hadron Industries, Inc.
# Carthage is free software; you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License version 3

from carthage.modeling import *

class layout(CarthageLayout):

    layout_name = "example_1"

    class foo(MachineModel):

        name = "foo.com"


```

With such a model, one might instantiate the layout by applying an injector:

```
layout_instance = injector(layout)
```

The *layout* class is an instance of `CarthageLayout` which is a kind of *`InjectableModelType`*. By default each assignment of a type in the class body of a *`InjectableModelType`* is turned into a runtime instantiation. This means that while `layout.foo` is a class (or actually a class property), `layout_instance.foo` is an `injector_access`. The first time `layout_instance.foo` is accessed, `layout_instance.injector` is used to instantiate it. Thereafter, `layout_instance.foo` is an instance of `layout.foo`.

3.2 The Modeling Language

Modeling classes are divided into several types (metaclasses). Names that include the word `modeling` are internal. Users may need to know about their attributes, but these classes should only be used in extending the modeling layer. Classes containing `model` in their name are directly usable in layouts. This section describes the behavior of the modeling types that make up the modeling language.

Model classes sometimes involve a new construct called a **modelmethod**. Unlike other types of methods, modelmethods are available in the class body. For example, *add_provider* can be used to indicate that on class instantiation, some object should be added to an *InjectableModel*'s injector:

```
class foo(InjectableModel):
    add_provider(InjectionKey("baz"), Network)
```

class carthage.modeling.**ModelingBase**

All modeling classes derive their type from *ModelingBase* and have the following behaviors:

- Unlike normal Python, an inner class can access the attributes of an outer class while the class body is being defined:

  ```
  class foo(metaclass  = ModelingBase):
      attr = 32
      b = attr+1
      class bar(metaclass = ModelingBase):
          a = b+1
          attr = 64
  ```

  In the above example, while the body of *bar* is being defined, *attr* and *b* are available.

  However, only variables that are actually set in a class body survive into the actual class. So in the above example, `foo.bar.a` and `foo.bar.attr` are set in the resulting class. While it was used in the class body, `foo.bar.b` will raise `AttributeError`. If an attribute should be copied into an inner class, the following will work:

  ```
  class outer(metaclass = ModelingBase):
      outer_attr = []
      class inner(metaclass = ModelingBase):
          outer_attr = outer_attr
  ```

- ModelingBases support the *modeling decorators*.
- The `dynamic_name()` decorator can be used to change the name under which an assignment is stored. This permits programatic creation of several classes in a loop:

  ```
  class example(metaclass = ModelingBase):

      # create a machine for each user
      for u in users:
          @dynamic_name(f'{u}_workstation')
          class workstation(MachineModel): # ...
      del u #to avoid polluting class namespace Now we have
      #several workstation inner classes, named based on the
      #argument to dynamic_name rather than each being called
      #workstation.
  ```

The dynamic_name decorator is particularly useful with *injectors* where it can be used to build up a set of machines that can be selected using `Injector.filter_instantiate()`.

class carthage.modeling.**InjectableModel**

class carthage.modeling.**InjectableModelType**

InjectableModel represents an `Injectable`. InjectableModels have the following attributes:

- InjectableModels automatically have an `Injector` injected and made available as the *injector* attribute.
- By default, any attribute assigned a value in the body of the class is also added as a provider to the injector in the class using the attribute name as a key. That is:

```
class foo(InjectableModel):
    attr = "This String"

foo_instance = injector(foo)
assert foo_instance.injector.get_instance(InjectionKey("attr")) == foo_instance.
↪attr == foo.attr
```

  This makes it very convenient to refer to networks and to construct instances that need to be constructed in an asynchronous context. Ideally there would be a decorator to turn this behavior off for a particular assignment, but currently there is not.

- By default, any attribute in the class body assigned a value that is a type (or that has a `transclusion key`) will be transformed into an `injector_access()`. When accessed through the class, the *injector_access* will act as a class property returning the value originally assigned to the attribute. That is, class access generally works as if no transformation had taken place. However, when accessed as an instance property, the *get_instance* method on the Injector will be used to instantiate the class. See the *first example* for an example. If this transformation is not desired use the `no_instantiate()` decorator.
- Certain classes such as `carthage.network.NetworkConfig` will automatically be added to an injector if they are assigned to an attribute in the class body.
- The `provides()` and `globally_unique_key()` decorators can be used to add additional `InjectionKeys` by which a value can be known.
- The `allow_multiple()` and `no_close()` decorators can modify how a value is added to the injector.

Decorators are designed to be applied to classes or functions. If modeling decorators need to be applied to other values the following syntax can be used:

```
external_object = no_close()(object)
val_with_extra_keys = provides(InjectionKey("an_extra_key"))(val)
```

The `dynamic_name()` decorator is powerful when used with *InjectableModel*. As an example, a collection of machines can be created:

```
class machine_enclave(Enclave):

    domain = "example.com"
    for i in range(1,5):
        @dynamic_name(f'server_{i}')
        @globally_unique_key(InjectionKey(MachineModel, host = f'server-{i}.
↪{domain}'))
        class machine(MachineModel):
            name = f"server-{i}"
```

Note that the call to `globally_unique_key()` is included only for illustrative purposes. The `our_key()` method of `MachineModel` accomplishes the same goal.

With a layout like the above, machine models are available as `machine_enclave.server_1`. But once the layout is instantiated, the injector can also be used:

```
machines = injector(machine_enclave)
machines.injector.get_instance("server_1")
#also available with the global key
machines.injector.get_instance(InjectionKey(MachineModel, host = "server-1.
↪example.com"))
#Or available all at once:
all_machines = machines.injector.filter_instantiate(MachineModel, ['host'],␣
↪stop_at = machines.injector)
```

add_provider(*key: InjectionKey*, *value*, ***options*)

Adds *key* to the set of keys that will be registered with an instance's injector when the model is instantiated. Eventually, in class initialization, code similar to the following will be called:

```
self.injector.add_provider(key, value, **options)
```

class carthage.modeling.**ModelContainer**

class carthage.modeling.**ModelingContainer**

InjectableModel provides downward propagation. That is, names defined in outer classes are available at class definition time in inner classes. Since `injector_access()` is used to instantiate inner classes, this means that the parent injector for the inner class is the outer class. Thus, attributes and provided dependencies made available in the outer class are available in the inner class at runtime through the injector hierarchy.

Sometimes upward propagation is desired. Consider the following example:

```
# Copyright (C) 2021, Hadron Industries, Inc.
# Carthage is free software; you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License version 3

from carthage import *
from carthage.modeling import *

class layout(CarthageLayout):

    class it_com(Enclave):

        domain = "it.com"

        class server(MachineModel): pass

    class bank_com(Enclave):

        domain = "bank.com"

        class server(MachineModel): pass
```

In this example machines can be accessed as `layout.bank_com.server` and `layout.it_com.server`. Once instantiated, the following injector access also works:

```
l = injector(layout)
l.bank_com.injector.get_instance(InjectionKey(MachineModel, host = "server.bank.com
↪"))
l.it_com.injector.get_instance(InjectionKey(MachineModel, host = "server.it.com"))
```

But you might want to look at machines without knowing where they are defined in the hierarchy:

```
l.injector.get_instance(InjectionKey(MachineModel, host = "server.it.com"))
# Or all the machines in the entire layout
l.injector.filter(MachineModel, ['host'], stop_at = l.injector)
```

Modeling containers provide upward propagation so these calls work:

entries registered in `l.it_com.injector` are propagated so they are available in `l.injector`. That's the opposite direction of how injectors normally work. Upward propagation is only at model definition time; the set of items to be propagated are collected statically as the class is defined. Items added to injectors at runtime are not automatically propagated up.

For upward propagation to work, containers must provide dependencies for some `InjectionKey`, and that key must have some constraints associated with it. For example, `Enclave`'s *our_key* method provides `InjectionKey(Enclave, domain = self.domain)`. If keys with constraints are marked with `propagate_key()`, then those are used. If not, then all keys with constraints are used.

When one container is added to another, all the container propagations in the inner container are propagated to the outer container as follows:

- If the propagation has a `globally_unique_key()`, then that key is registered unmodified in the outer container.
- If there is no globally unique key, then the constraints of the propagation's key are merged with the constraints of the key under which the inner container is registered with the outer container. Consider an inner container `InjectionKey(Enclave, domain="it.com")` and a propagation of `InjectionKey(Network, role = "site")`. Within the inner container, the network can be accessed using `InjectionKey(Network, role = "site")`. After the constraints are merged, the network can be accessed in the outer container as `InjectionKey(Network, role = "site", domain = "it.com")`.

The `injector_xref()` facility is used so that instantiating the key in the outer container both instantiates the inner container and the object within it.

Only the following objects are considered for propagation:

- Any *ModelContainer* including `MachineModel`, `NetworkModel`, `ModelGroup`, *ModelContainer*, and `Enclave` is propagated.
- The `propagate_key()` decorator can be used to request propagation for other objects.

3.3 Base Models

3.4 Decorators

CHAPTER

FOUR

DESIGNING A GOOD MODEL

4.1 Arguments

Arguments for a model can come from multiple places:

- From the injector hierarchy:

```
@inject_autokwargs(verbose_logging=InjectionKey("verbose_logging"))
class Model(InjectableModel): pass

class enclave(Enclave):

    verbose_logging = True

    class submodel(Model): pass #gets verbose_logging from its environment
```

- At instantiation time:

```
model = await self.ainjector(Model, verbose_logging='tuesdays_only')
```

- From within a subclass in a layout:

```
class enclave(Enclave):

    class model_instance(Model):

        verbose_logging = 'wednesdays'
```

In most situations, if an argument can be specified in the injector hierarchy, the other two methods of argument passing should also work. Supporting an argument as either kwargs or specified in a subclass is desired in most cases. Kwarg (and injector) support may not be needed if subclass support is provided and an argument can be adjusted on an instance after instantiation.

When considering which forms of argument passing are appropriate for a given model, consider two usage scenarios. A model should be usable in a Python program that instantiates it in a procedural function. Such usage can easily supply kwargs and can easily set properties on an instance after instantiation. Models should also be usable in the declarative modeling language. That usage makes it easy to specify dependencies provided by injectors and to set default values in subclasses. Specifying kwargs not supplied by dependencies from injectors and adjusting properties after instantiation are more difficult in the declarative language.

Note there are some interactions between these argument methods.

1. When arguments are specified in the injector hierarchy, the `Injector` will populate kwargs from the injected dependencies, so when instantiated the model will receive the arguments as kwargs.

2. In contrast, when arguments are specified within an instance of a model in the modeling language, kwargs are never used. Instead, properties are set directly on the class that the model inherits from.
3. Arguments specified in a modeling language instance will typically cascade into sub models via the injector hierarchy—the third method of specifying arguments also can become the first:

```
class outer_model(Model):

    verbose_logging = 'odd_thursdays' # sets for this instance

    class inner_model(Model):

        # But also sets a value in the injector, so that
        # the inner instance also gets a verbose_logging of 'odd_thursdays'
```

4.1.1 Implementing the Three Argument Strategies

The following class can take *verbose_logging* either from the injector environment, as a kwarg, or set in subclasses:

```
@inject_autokwargs(
verbose_logging=InjectionKey('verbose_logging', _optional=NotPresent),
)
class SomeModel(InjectableModel):

    # do things here
    verbose_logging = False
```

If the injector environment does not contain *verbose_logging* then no kwarg is specified (because of setting *_optional* to *NotPresent*). That will permit a subclass to override the default for *verbose_logging* specified in the class. An explicit kwarg will override the injector environment, as is always the case for `Injector.__call__()`.

Note that because this model is a modeling class, any contained subclass will have *verbose_logging* in its injector environment:

```
  @inject_autokwargs(
verbose_logging=InjectionKey('verbose_logging', _optional=NotPresent),
)
class SomeModel(InjectableModel):

    # do things here
    verbose_logging = False
    class interior(InjectableModel):

        # verbose_logging is set.
```

CHAPTER

FIVE

CONFIGURING VAULT

The `Vault.apply_configuration()` method will apply a set of configurations to a Hashicorp Vault. The intent is to allow initial configuration of policies and authentication.

The method takes a dictionary, but typically this dictionary comes from a YAML file. The following special keys are recognized:

policy:
: A Dictionary mapping policies to HCL documents.

auth:
: A dictionary for auth method configuration. The keys in this dictionary are paths on which authentication methods are mounted; The values are dictionaries containing the following keys:

  type
  : The type of authentication method; `cert`, or `github` for example.

  default_lease_ttl
  : The default lease TTL

  maximum_lease_ttl
  : The longest lived tokens issued by this auth backend.

All other keys in the configuration dictionary will be taken as paths that will be written. The value will be JSON encoded. An example YAML file might look like:

```
policy:
  sec_admin: |
    path "/sys/policy/*" {
      capabilities = ["read", "update", "create", "delete", "list"]
      }
auth:
  cert:
    type: cert
    default_lease_ttl: 60m
auth/cert/cert/hadron:
  certificate: |
    A PEM encoded certificate goes here
  allowed_common_names: ["*@hadronindustries.com"]
```

PYTHON MODULE INDEX

C

INDEX