kumori
SYSTEMS

Writing Services:

The service model

Goals

- Understand what the kumori service model entails
- Get proficiency on how to author kumori components and services
- Learn how to interact and deploy services through the axebow platform

https://docs.kumori.systems/

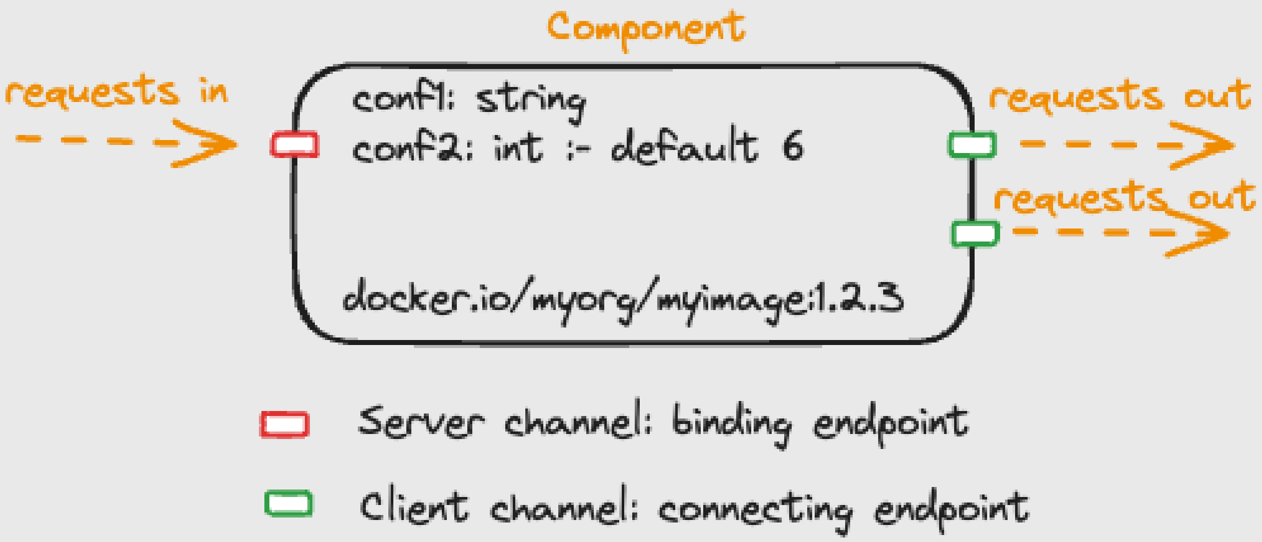
kumori
SYSTEMS

Short glossary

- Application
  - Program
  - Precise/unambiguous specification of how some system must behave
- Service
  - What one gets out of deploying, and running a program.
    - Holding state
    - Requiring life-cycle management
  - Typically through some endpoints
- Microservice
  - Service focused on delivering an atomic functionality
  - Typically part of a larger, more complex service.

Elements of the Kumori Service Model

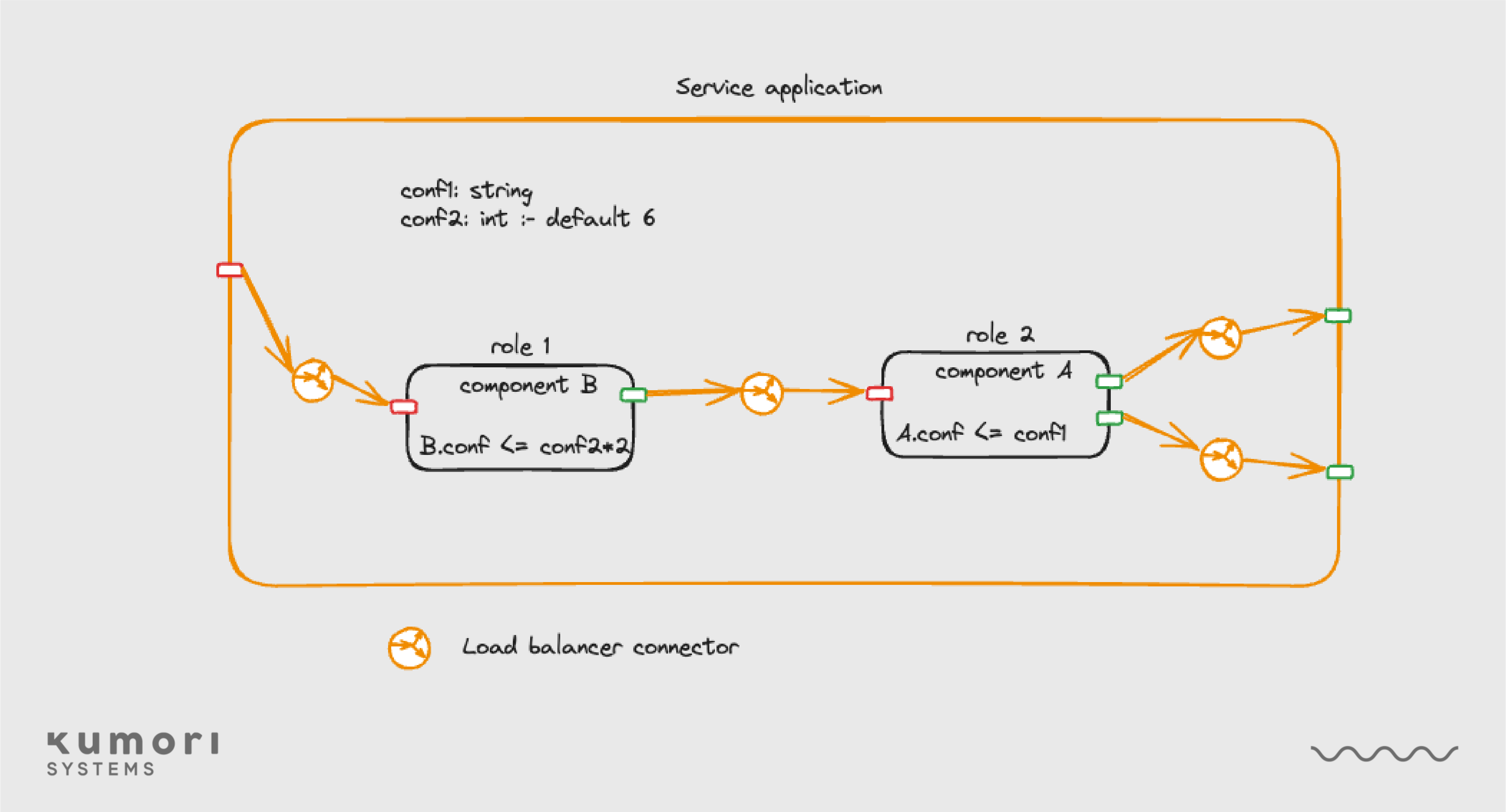
- *Component*
  - Specifies the behavior of a microservice
    - Contains the code/application
    - In Kumori, it must be packaged as a docker image.
  - Specifies, among other things, how the component can be configured when activated
  - Also specifies how in communicates
    - Which dependencies it needs to communicate with
    - What endpoints it opens to accept requests from others


Component
requests in
conf1: string
conf2: int :- default 6
docker.io/myorg/myimage:1.2.3
requests out
requests out
Server channel: binding endpoint
Client channel: connecting endpoint

Elements of the Kumori Service Model

- *Service application*
  - Specifies how to structure multiple microservices within a complete service
    - Roles encapsulating components as microservices
    - Communication patterns between roles
      - Taking into account the component’s endpoints
  - Specifies how the whole service is configured
    - With rules distributing that configuration to each one of the service’s role
    - How each role distributes/transforms configuration for its component
  - Also specifies how in communicates
    - Which dependencies it needs to communicate with
    - What endpoints it opens to accept requests from others

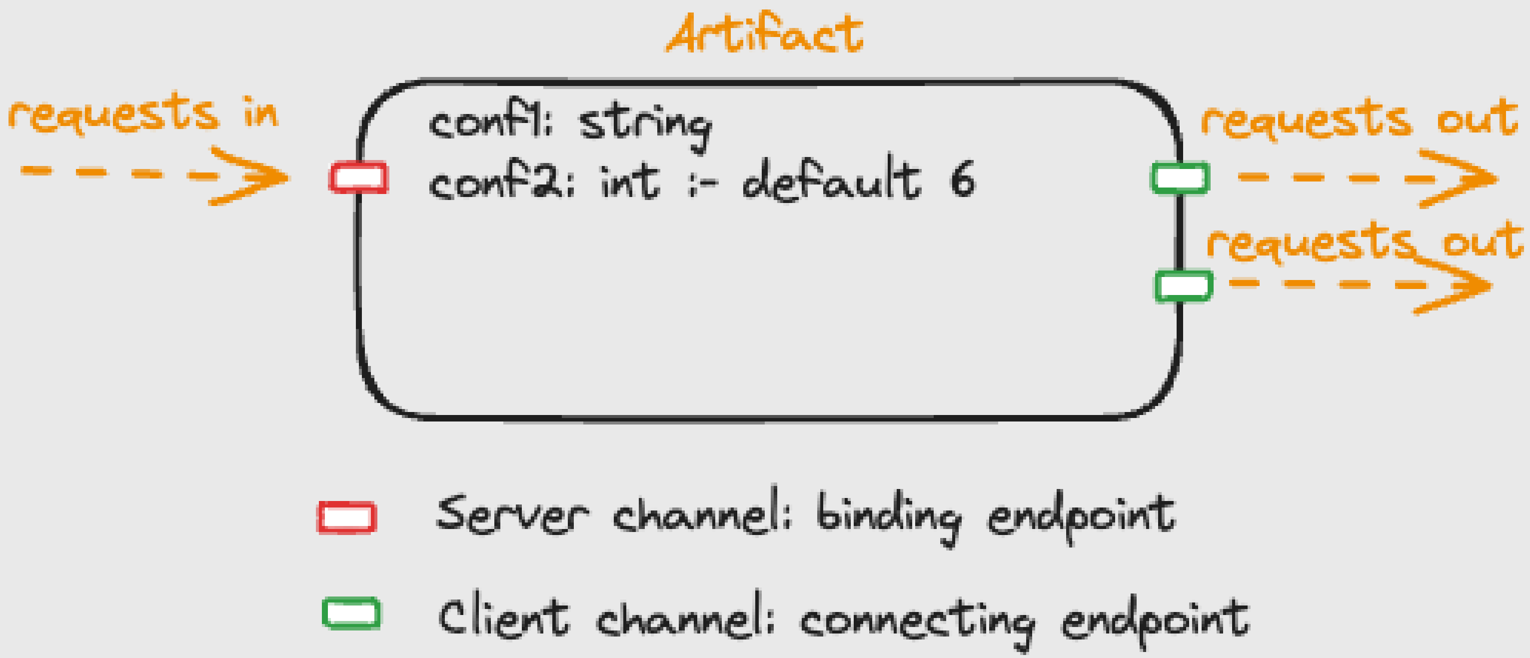
kumori
SYSTEMS


Service application
conf1: string
conf2: int :- default 6
role 1
component B
B.conf <= conf2*2
role 2
component A
A.conf <= conf1
Load balancer connector
kumori
SYSTEMS

Elements of the Kumori Service Model

- *Artifact:*
  - *Component*
  - *Service Application*
- *Common elements of artifacts*
  - Configuration
  - Communication endpoints (aka “channels”)

kumori
SYSTEMS



kumori
SYSTEMS

Elements of the Kumori Service Model

- *Deployment*
  - Sets the <u>initial</u> goal state of a service
    - Identifies the service application to be deployed
    - Provides concrete values for the configuration of the service
      - Following the configuration spec of the service application being deployed

kumori SYSTEMS

Specifying Kumori artifacts

- Uses a Domain Specific Language

- In v. 1.0: Based on cuelang
  - Internal DSL on top of CUE
- In v 2.0
  - External DSL, with VSCode support
  - Better tooling
- We will look at the v1.0 language
  - V2.0 still in the oven ☹

kumori SYSTEMS

Manifests

- *An artifact is specified with a manifest for the artifact*
- *A manifest has the following general structure*

```
{
  spec: #Version
  ref:  #ManifestRef
  description: _
}
```

- *Specification language is CUE*
    - https://cuelang.org
- *Manifests form an internal DSL within CUE*

kumori SYSTEMS

Manifests

- *Artifact manifests are shipped within kumori modules*
- *Kumori module*
  - *Unit of versioning and distribution*
  - *A Kumori module has its own reference*
  - *A Kumori module is versioned*
- *A kumori module can contain many different manifests of artifacts*
  - *Each one within its own directory*
  - *Names of artifacts follow the path of the directory they are within*
  - *All artifacts within a module inherit the module’s version, domain and module name (oc)*

kumori
SYSTEMS

Manifests

```
{
  spec: #Version
  ref:  #ManifestRef
  description: _
}
```

- The *spec* field specifies the version of the manifest specification
  - Determines the format for the rest of the manifest
- *description* contains the specific info needed for each kind of artifact
- The *ref* field uniquely identifies the artifact being defined.

kumori SYSTEMS

Manifests: the Ref field

```
#ManifestRef: {
  version: *[1, 0, 0] | #Version
  domain: string
  module: string
  name:   string
  kind:   string
}
```

- Field *version* follows semantic versioning
- Field *kind* specifies the artifact kind (*component*, *service)*
- Field *domain* identifies the owner of the artifact (as a domain name)
- Field *module* identifies the module that distributes the artifact
- Field *name* completes the ref ID of the artifact.

Publishing/reproducibility

- Once a module has been published, it does not change
  - Module inmutability: all artifacts within a module's version stay constant
- The *ref* uniquely identifies an artifact
  - Its name derived from the full module's name
- A module can be ported
  - … and so all its artifacts
- A set of modules can be bundled
  - To move it or to prepare it for deployment

Publishing/reproducibility

- It is not necessary to have a local registry of modules
  - But there is a caching mechanism
  - Local modules can be used
- Modules are registered currently as npm packages
  - A CLI tool exists to facilitate this.
  - The npm scope must coincide with the domain of the module.
    - Thus, the publisher must have credentials to publish under the scope

kumori
SYSTEMS

Tooling (advance)

- Tool: *kam*
- Installation:

    npm i –g @kumori/kam

- Usage:
    - **kam** mod init domain/modname
    - …
- Can be used to run *cue*
    - **kam cue** …
- Needs nodejs

Preliminaries: the CUE language

- Configuration specification language
  - Superset of JSON
    - Any JSON is CUE
    - Any YAML can be converted to CUE
- Based on logic programming
- Lets you define data schemas
  - With more generality than JSON Schema or OpenAPI
  - Thus factoring out repetitive patterns
- Has sufficient expressiveness to derive values out of other values

CUE

- Implements a set type system
  - Thus, individual elements within a type are also types
  - CUE does not distinguish between them (exceptfor “exporting”)
- “type” specifications are simply restrictions imposed on a field

Schema:

```
ciudad: {
  nombre: string
  pop:   int
  capital: bool
}
```

Specialization:

```
cGrande: {
  nombre: string
  pop:   >5
  capital: true
}
```

Data:

```
moscu: {
  nombre: ”Moscú"
  pop:   11.92
  capital: true
}
```

kumori
SYSTEMS

CUE: basic types

Basic types
int
number
string
bool
null

Special	
_	any/top
_|_	bottom/ nothing

Structured	
[...]	Any array
{...}	Any struct

kumori SYSTEMS

CUE

- The same field can be specified multiple times
  - Each time adds a restriction to the contents the field can have
  - If the set of restrictions on a field end up being incompatible, the field equals _|_
    - ...which makes the structure it is part of also be _|_
- Order of field specifications is unimportant

```
a: int
a: >0
```

```
a: int
a: > 0
a: 5
```

```
a: < 100
```

```
a: 6
```

Conflict

kumori SYSTEMS

CUE: operations

- *Disjunction* (or union…)
  - Given two types, The resulting type is the union of both
  - Operator **|**
  - Conmutative and associative

```
ob1: "something" | "another"

ob2: string | uint

ob3: ob1 | ob2

ob3: 56
```

kumori SYSTEMS

CUE: unification

- Given two types, A, B
  - Find the “largest” type, C, such that C is part of A and C is part of B.
- “largest”?
  - Let C be included in A and B,
    - C is the largest iff
      - For any D included in A and B, D is included in C.
- Operator. &.

```
ob1: {
  a: int
}

ob2: {
  b: string
}

ob3: ob1 & ob2

ob3: {
  a: int
  b: string
}
```

kumori SYSTEMS

CUE: unification

- Two restrictions on the same field function as a unification operation
  - Conmutative and associative operation

```
ob1: {
  a: int
}
```

```
ob2: {
  b: string
}
```

```
ob3: ob1
ob3: ob2
```

```
ob3: {
  a: int
  b: string
}
```

kumori
SYSTEMS

CUE: unification

```
ob1: {
  a: string
  b: 3
}
```

```
ob2: {
  a: "something"
}
```

```
ob3: ob1
ob3: ob2
```

```
ob3: {
  a: “something”
  b: 3
}
```

kumori SYSTEMS

CUE: defaults

```
ob1: {
  a: string
  b: int | *3
}
```

```
ob2: {
  a: "something"
}
```

```
ob3: ob1
ob3: ob2
```

```
ob3: {
  a: “something”
  b: int | * 3
}
```

kumori SYSTEMS

CUE: defaults

```
ob1: {
  a: string
  b: int | *3
}
```

```
ob2: {
  a: "something"
  b: 67
}
```

```
ob3: {
  a: "something"
  b: 67
}
```

```
ob3: ob1
ob3: ob2
```

kumori SYSTEMS

CUE: definitions

- Another way to specify fields
- A field is a definition if its name starts with "#"
- A definition introduces a <u>*closed*</u> structure

<u>Closed Structure</u>

- Cannot be unified with anything that would introduce new fields
  - Si se unifica con otra estructura, la segunda no puede introducir un campo nuevo

CUE: estructuras cerradas

```
#A: {
  pop: int
  name: string
}

B: {
  capital: bool
}

madrid: #A & B
```

CONFLICT

kumori SYSTEMS

CUE: subsumptions/mezclas

```
#A: {
  pop: int
  name: string
}
```

```
B: {
  capital: bool
}
```

```
madrid: {
  #A
  B
}
```

```
madrid: {
  pop: int
  name: string
  capital: bool
}
```

kumori SYSTEMS

CUE: export

```
#A: {
  pop: int
  name: string
}
```

```
B: {
  capital: bool
}
```

```
madrid: {
  #A
  B
} & {
  pop: 3
  name: “Madrid”
  capital: true
}
```

cue export

ERROR

kumori SYSTEMS

CUE: export

```
#A: {
  pop: int
  name: string
}
```

```
#B: {
  capital: bool
}
```

```
madrid: {
  #A
  #B
} & {
  pop: 3
  name: "Madrid"
  capital: true
}
```

cue export

OK

```
madrid: {
  pop: 3
  name: "Madrid"
  capital: true
}
```

kumori SYSTEMS

CUE: referencias

```
#A: {
  spec: #B
  alt: spec.capital
}
```

```
#B: {
  capital: string
}
```

```
spain: #A & {
  spec: {
    capital: “Madrid”
  }
}
```

```
spain: {
  spec: {
    capital: “Madrid”
  }
  alt: “Madrid”
}
```

kumori SYSTEMS

Kumori Modules

- CUE code organized within modules
- Modules are the distribution units
- MODULE == special structure folder
  - Subfolder *cue.mod*
    - Managed by kumori tool, needed for CUE loader: DO NOT TOUCH
  - File *kmodule.cue* contains fields (see next slide),
    - *Relevant names (domain, module, version)*
    - Dependencies
      - With query/lock
      - With an alias that can be used in imports
    - Checksums
  - Common files for components and services
    - Auto-generated: DO NOT TOUCH

kumori SYSTEMS

- cue.mod
- inbound
- .gitignore
- .gitlab-ci.yml
- README.adoc
- componentref.cue
- kmodule.cue
- service_artifact.cue
- serviceref.cue
- utils.cue

```
package kmodule

{
	local:  true
	domain: "kumori.systems"
	module: "builtins/inbound"
	cue:    "v0.4.3"
	version: [
		1,
		3,
		1,
	]
	dependencies: "kumori.systems/kumori": {
		target: "kumori.systems/kumori/@1.1.6"
		query:  "1.1.6"
	}
	sums: "kumori.systems/kumori/@1.1.6": "jsXEYdYtlen2UgwDYbUCGWULqQIigC6HmkexXkyp/Mo="
	spec: [
		1,
		0,
	]
}
```

Kumori packages and manifests

- Kumori manifests reside within a kumori module (thus a CUE module)

- Given a KMODULE, several artifacts can be within
- The manifests for component artifacts reside within a “component” CUE package
- The manifests for service artifacts reside within a “service” CUE package
- Given the CUE loader
  - It is possible to share parts of manifests among multiple artifacts of the same kind

kumori
SYSTEMS

- cue.mod
- inbound
- .gitignore
- .gitlab-ci.yml
- README.adoc
- componentref.cue
- kmodule.cue
- service_artifact.cue
- serviceref.cue
- utils.cue

```
// Automatically generated file. Do not edit.
package component

import (
  k "kumori.systems/kumori:kumori"
  m "...:kmodule"
)

#Artifact: k.#Artifact & {
  spec: m.spec
  ref: {
    local: m.local
    version: m.version
    if m.prerelease != _|_ {
      prerelease: m.prerelease
    }
    if m.buildmetadata != _|_ {
      buildmetadata: m.buildmetadata
    }
    domain: m.domain
    module: m.module
    kind: "component"
  }
}
```

Packages

- Modules are structured within packages
  - Three package names used within kumori modules
    - *kmodule*
    - *component*
    - *service*
  - Also, the main "*kumori*" package to distribute the model restrictions
    - Only used by kumori code
- Package loader works following folder structure.
  - We take advantage of it.

kumori SYSTEMS

cue.mod

inbound

.gitignore

.gitlab-ci.yml

README.adoc

componentref.cue

kmodule.cue

service_artifact.cue

serviceref.cue

utils.cue

```
// Automatically generated file. Do not edit.
package service

import (
  k "kumori.systems/kumori:kumori"
  m "...:kmodule"
)

#Artifact: k.#Artifact & {
  spec: m.spec
  ref: {
    local: m.local
    version: m.version
    if m.prerelease != _|_ {
      prerelease: m.prerelease
    }
    if m.buildmetadata != _|_ {
      buildmetadata: m.buildmetadata
    }
    domain: m.domain
    module: m.module
    kind: "service"
  }
}
```

KUMORI SYSTEMS

- cue.mod
- inbound
- .gitignore
- .gitlab-ci.yml
- README.adoc
- componentref.cue
- kmodule.cue
- service_artifact.cue
- serviceref.cue
- utils.cue

Name
..
placeholder.cue

```
package service

import (
  k "kumori.systems/kumori"
)

// Definition of a list of IPs (regular IPs or CIDRs, defined via regular expression)
let ip = "(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.(2
let cidr = "\(ip)/([0-9]|[1-2][0-9]|3[0-2])"
#ipcidr: =~ "^\(ip)|\(cidr)$"
#IPList: [...(#ipcidr)]

#Artifact: {

  ref: name: "inbound"

  description: {
     builtin: true
     config: {
      parameter: {
        type: *"https" | "tcp"
        if type == "https" {
          clientcert: bool | *false
          if clientcert == true {
            certrequired: bool | *false
          }
          websocket: bool | *false
          // Name of the header (for example, "X-Real-IP") where the remote address
          // will be setted. This may not be the physical remote address of the
          // peer if the address has been inferred from the x-forwarded-for
          // (depends on cluster configuration)
          remoteaddressheader: string | *""
          // Clean the x-forwarded-for header
          cleanxforwardedfor: bool | * false
          // An http inbound  can declare a list of allowed or denied IPs
          //
          // This syntax is preferable, because it prevents the simultaneous
          // existence of "allowediplist" and "deniediplist", but an error occurs
          // during the service spread.
          //   accesspolicies?: { allowediplist: #IPList } | { deniediplist: #IPList }
          //
          // (currently not supported for tpc inbounds)
          //
          accesspolicies?: {
            allowediplist?: #IPList
            deniediplist?: #IPList
          }
        }
      }
      resource: {
        if config.parameter.type == "https" {
          servercert   : k.#Certificate
          serverdomain : k.#Domain
          clientcertca?: k.#CA
        }
        if config.parameter.type == "tcp" {
          port: k.#Port
        }
      }
    }
    srv: client: inbound: _
  }
}
```

kumori SYSTEMS

Importing packages

- Within a CUE file it is posible to import other packages
  - Typically from other kumori modules holding artifacts
  - Potentially from the kumori module holding the model spec itself.
- Syntax similar to GOlang

```
import (
  k     "kumori.systems/kumori/kmv3_0_1"
        "kumori.systems/integrations/hazelcast/v3/server"
  mngmnt "kumori.systems/integrations/hazelcast/v3/management_center:component”
  mc     "mod.local/subdir:component"
)
```

- Can provide alias (imports 1, 3 & 4)
  - Implicitly, the basename of the import (import 2: server)
  - Must provide the kind of artifact (3)
  - Can refer to artifacts within the current module as “mod.local”
- A package fields are available qualifying them with the alias of the import.

kumori SYSTEMS

Components

- Description with several sections:
  - Microservice
  - Configuration
  - Size
  - Code

```
description: {
  srv:    {...}
  config: {...}
  size:   {...}
  code:   {...}
}
```

kumori SYSTEMS

Component: microservice

- Channel description:
  - **Client** channel (dependencies)
  - **Server** channel
    - Must specify the port
  - **Duplex**
    - Sort of shortcut representing a client and a server
    - Exposes the endpoints llinked to it (IP:port)
    - Complex protocols
      - E.g., DDBB, distributed runtimes (BEAM,…)
- All names are local to the component
  - Independent of the service where the component is used

kumori SYSTEMS

Component: microservice

```
srv: {
  client: [string]: #Channel
  server: [string]: #Server
  duplex: [string]: #Server
}

#Channel: {
  protocol: "udp" | "tcp" | *"http" | "grpc"
}

#Server: {
  #Channel
  port: uint16 | *80
}
```

Component: service discovery

- What is?
  - Process by means of which the code discovers at run time how to contact a dependency.
- KPaaS uses DNS mechanisms
  - Uses domain names → names of client channels
    - Local to the component.
      - Known at "compile time"
    - KPaaS injects dependencies at deployment time
    - KPaaS modifies info as it changes
- In KPaaS dependencies are mainly client channels
  - Also duplex channels, though they work differently

kumori SYSTEMS

Component: basic service discovery

- KPaaS solves dependencies as follows
  - Given a client channel, ***C,***
    - nslookup 0.C ➔ IP(s) of servers
    - nslookup –S 0.C ➔ IP:port of servers
    - Tag "0" refers to the first target of a dependency
      - A channel can point to several endpoints, each one has a different tag number
      - /kumori/config.json contains information about tags
- Bonus
  - Reserved channel: ***self***
  - nslookup self ➔ IP de la instancia
    - Useful for binding to a specific external interface for the container (***bind***)
- Bonus II
  - It is also possible to resolve channel "C"
    - The result is address 127.0.0.N (We will see this later)

kumori SYSTEMS

Component: Configuration

- Under field config
- Two subsections: parameter, resource
- parameter is a dictionary/tree
  - Arbitrary JSON
- resource is also a dictionary/tree
  - Leaves satisfy #Resource

```
config: parameter: [string]: _
```

kumori SYSTEMS

Componente: Configuración, recursos

```
config: resource: #ResourceTree

#Resource: #Volume     | #Secret
#Volume:   #Persistent | #Volatile

#Secret:     {secret: string}
#Volatile:   {size: uint, unit: string}
#Persistent: {volume: string}
```

kumori
SYSTEMS

Component: Size

- Simple declaration of the “vertical” resources that instances of a component need

```
size: #ComponentSize

#ComponentSize : {
  mincpu:          #ResourceAmount & {kind: "cpu"}
  minbandwidth?:    (number & >= 0  & <= bandwidth.size)
  bandwidth:       #ResourceAmount & {kind: "bandwidth"}
}

#ResourceAmount {
  kind:          #UnitKinds
  size:          number & >= 0
  unit:          #Units[kind]
}

#Units: {
  storage: "k" | "M" | "G" | "T" | "P" | "E" | "Ki" | "Mi" | "Gi" | "Ti" | "Pi" | "Ei"
  cpu:  "m" | ""
  ram:  "G" | "M" | "Gi" | "Mi"
  bandwidth: "G" | "M" | "Gi" | "Mi"
}
```

Component: Code

- Description of how the containers should be activated
- As a dictionary
  - The keys are the container names
  - Values describe how each container should be configured
    - Its image
    - How to map configuration to environment and filesystem entities

```
code: [nm=string]: #Container & {name: nm}
```

Component: Code

```
#Container: {
  name: string
  image: #Image
  entrypoint?: [...string]
  cmd?: [...string]
  user?: {
    userid: uint16
    groupid: uint16
  }
  mapping: #Mapping
}
```

```
#Image: {
  hub: #Hub | * {
    name:  "registry.hub.docker.com”
    secret: "”
  }
  tag: string
  digest?: string
}

#Hub: {
  name: string
  secret: string
}
```

kumori
SYSTEMS

Component: Mappings

- Two types: Environment variables, or file system
  - Use cue references from the config section.
- Mechanism to convert config to constructs usable by the code within the container

```
#Mapping: {
  filesystem: [...#FileMap | #FolderMap]
  env: #EnvMap
}

#EnvMap: {
  [string]: {value: string} | #Secret
}
```

kumori
SYSTEMS

Component: Mappings

- Two filesystem mapping types: Folders and Files
- Lets developers define arbitrary folder trees
  - A folder must map to a tree or to a Volume resource (by resource name)

```
#FileMap: {
  path: string
  mode: uint16 | *0o644
  data: _ // jsonable
  format: *"text" | "json" | "yaml"
} | {
  path: string
  mode: uint16 | *0o644
  #Secret
}
#FolderMap: {
  path: string
  { {tree: [...#FileMap | #FolderMap]} | #Volume }
}
```

kumori SYSTEMS

Component: Spread & Example

Frontend de calculator cache

https://gitlab.com/kumori-systems/community/examples/calc-cache

Worker de calculator-cache

kumori SYSTEMS

Service

- A service's description has 5 sections
  - Microservice interface
  - Configuration
  - Roles
  - Connectors
  - Links
- NOTE: Components and Artifacts share the two first sections

```
description: {
  srv:       {...}
  config:    {...}
  role:      {...}
  vset:      {...}
  connector: {...}
  link:      {...}
}
```

kumori SYSTEMS

Service: Roles

- A *role* is a way of specifying how to deploy a component within a service application.
- The roles section within a service app is structured as a dictionary whose names are the role names.

```
role: [rn=string]: #Role & {name: rn}
```

```
#Role: {
  artifact: #Artifact
  name:     string
  meta:    {...}
  config:  #Configurable
}
```

```
#Configurable: {
  parameter: [string]: _
  resource:  [string]: #Resource
  resilience: uint | *0
  scale: #ScaleSpec
}
```

kumori SYSTEMS

Service: Topology

- Contains connectors and links
- A connector allow linking from client to server channels within a service's role
  - Can be visualized as two endpoints:
    - Client endpoint: role client channels attach to it
    - Server endpoint: attaches to role server channels
- Connector types
  - Two types
    - LB: Load Balancer (rr)
    - FULL: complete connector/Full
    - Determine how to deal with the multiplicity of instances in the roles they attach to
      - Condition how service discovery ends up working

kumori SYSTEMS

Services: Links

- They are the edges of the topology
- Only the following attachments are allowed
  - From a role client channel to a connector client endpoint
  - From a connector server endpoint to a role server channel
    - When the connector is full and absend links to the client enpoint
      - also allow link to a duplex server channel
- *self* is special:
  - From one of the service’s server channels to a conector’s client endpoint
  - From a connector’s server endpoint to one of the services’ client endpoints
  - Only attaches to LB connectors

kumori SYSTEMS

Service: Topology

```
connect: {
  as: “lb” | “full”
  from: [rn=#roles | “self”]: role[rn].srv.#clients
  to: [rn=#roles]: role[rn].#servers: {
     meta: {...}
  }
}
```

kumori
SYSTEMS

Service: Connectors and discovery

- LB Connector
  - Represented by a balanced IP
  - Channels connected to the client endpoint of an LB resolve the balanced IP
    - Its port is always 80
- FULL Connector
  - Channels connected to the client endpoint of the connector resolve all IP's of the server channels to which the server endpoint connects.
  - When resolving the SRV records, all IP:port are returned
  - When only the server endpoint is connected, it must connect to a duplex channel
    - Resolving the A record of the duplex channel returns all Ips of the instances
    - Resolving the SRV record of the duplex channel returns all IP:port of the instances

Configuration Spread

- Same as for components: using CUE references and *unification*
  - Within the role record
    - At the config field: left hand side is the config fo the role's artifact, right hand side is a reference to the service app fields
- Referencing, mainly, the configuration section of the service.

```
role: myrole: rsize: {
  resilience: config.parameter.maxfailures
}

role: myrole: artifact: myComponentManifest
role: myrole: cfg: {
  parameter: color: config.parameter.background
}
```

kumori SYSTEMS

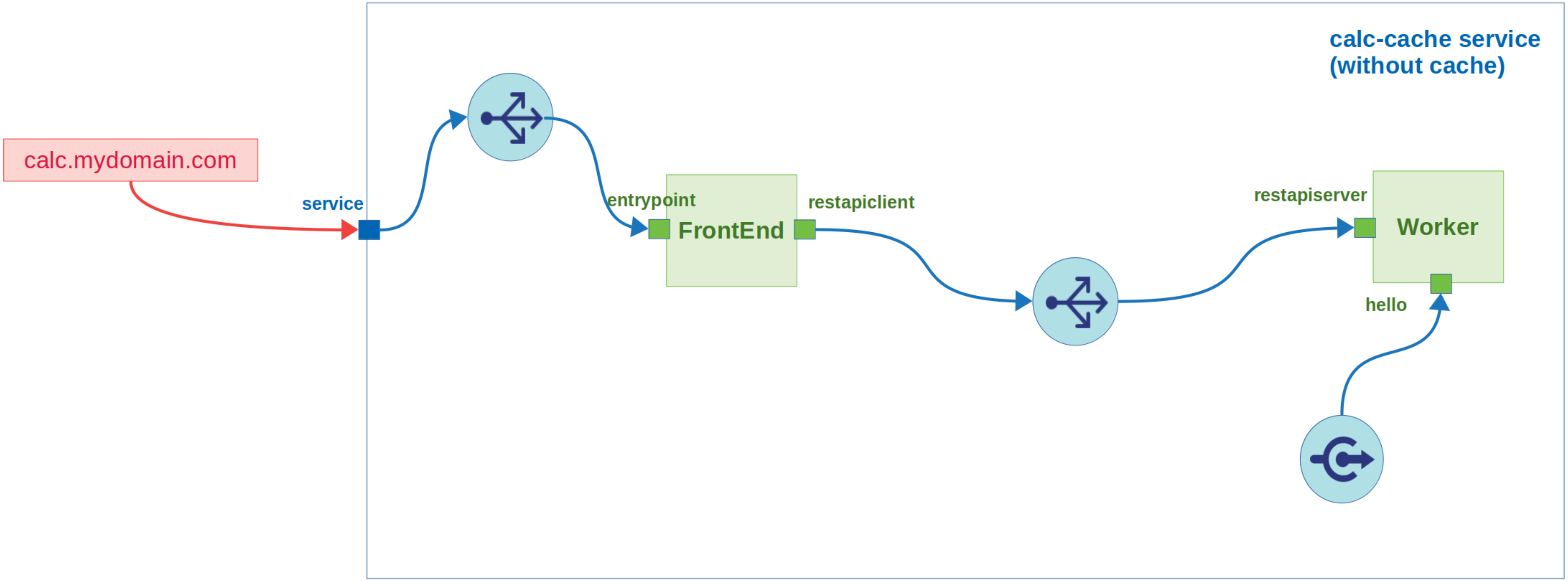
Deployment

- A deployment manifest provides
  - A reference to an artifact to be deployed
  - A set of concrete values for the configuration parameters of the artifact being deployed
    - Including data informing about the number of instances to be deployed for each role
- With a deployment manifest
  - A service initial revision can be deployed
  - A new revision can update a previously running revision

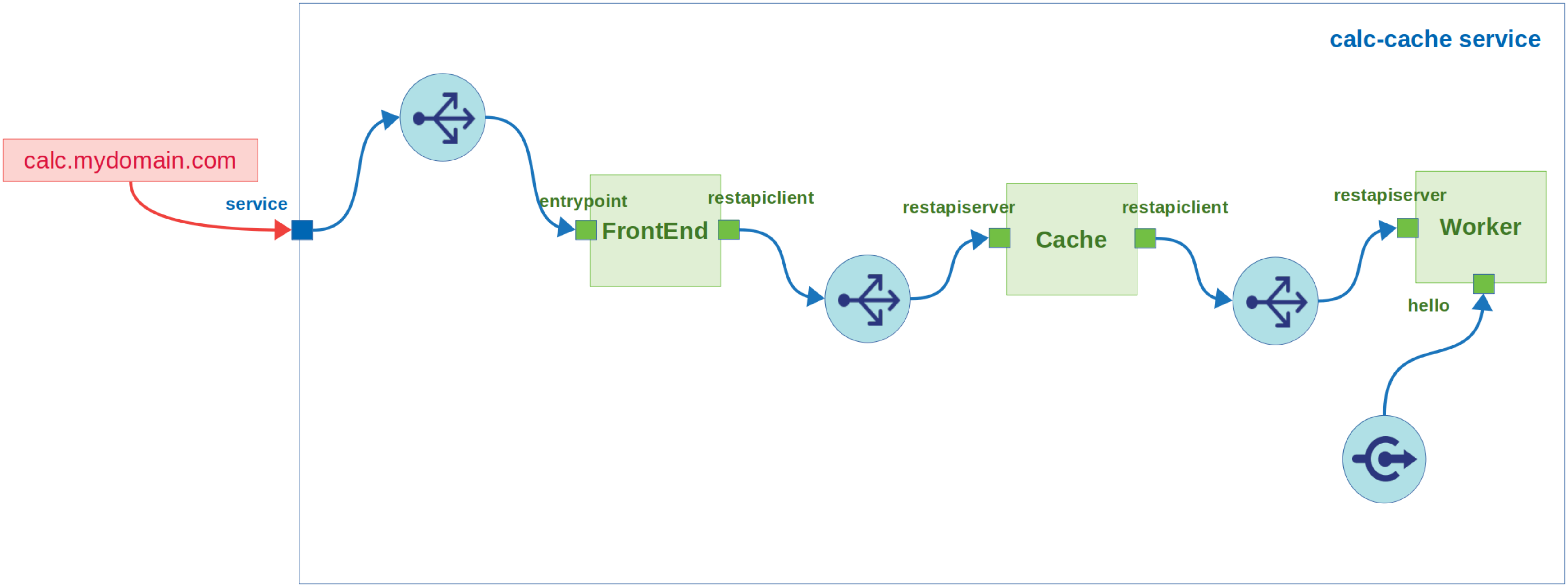
```
#Deployment: {
  artifact: #Artifact
  meta : {…}
  config: {…}
}
```

Example: calculator-cache

- Toy example
- A simple calculator but with a possible twist
  - Frontend accepts requests and pases them to a worker
    - Worker has a dúplex cannel but this toy example does not need it.
  - A variation has the frontend request a value from the cache instead
    - The cache will contact the worker if need be
- A total of 2 roles in the simple case
  - 3 in the “more complex” case.



kumori
SYSTEMS


calc-cache service
calc.mydomain.com
service
entrypoint
FrontEnd
restapiclient
restapiserver
Cache
restapiclient
restapiserver
Worker
hello

Turn to editor

kumori
SYSTEMS

kam

- Setting up the working environment
  - Create the working directory
    - mkdir tutorial
    - cd tutorial

kam mod init kumori.examples/calccache

- Creates a set of files
  - Sets it up with some defaults

kam mod dependency kumori.systems/kumori

```
componentref.cue
cue.mod
kmodule.cue
serviceref.cue
```

```
{
        domain: "kumori.examples"
        module: "calccache"
        cue:    "0.4.2"
        version: [
                0,
                0,
                1,
        ]
        dependencies: {}
        sums: {}
        spec: [
                1,
                0,
        ]
}
```

kumori SYSTEMS

kam

- Setting up the working environment
  - Create the working directory
    - mkdir tutorial
    - cd tutorial

kam mod init kumori.examples/calccache

- Creates a set of files
  - Sets it up with some defaults
  - Then:

kam mod dependency kumori.systems/kumori

```
componentref.cue
cue.mod
kmodule.cue
serviceref.cue
```

```
{
        domain: "kumori.examples"
        module: "calccache"
        cue:    "0.4.2"
        version: [
                0,
                0,
                1,
        ]
        dependencies: {}
        sums: {}
        spec: [
                1,
                0,
        ]
}
```

kam

- Setting up the working environment
  - Create the working directory
    - mkdir tutorial
    - cd tutorial

kam mod init kumori.examples/calccache

- Creates a set of files
  - Sets it up with some defaults
  - Then:

kam mod dependency kumori.systems/kumori

```
componentref.cue
cue.mod
kmodule.cue
serviceref.cue
```

```
{
	domain: "kumori.examples"
	module: "calccache"
	cue:    "0.4.2"
	version: [
		0,
		0,
		1,
	]
	dependencies: {}
	sums: {}
	spec: [
		1,
		0,
	]
}
```

```
{
	domain: "kumori.examples"
	module: "calccache"
	cue:    "0.4.2"
	version: [
		0,
		0,
		1,
	]
	dependencies: "kumori.systems/kumori": {
		target: "kumori.systems/kumori/@1.1.7"
		query:  "latest"
	}
	sums: "kumori.systems/kumori/@1.1.7": "kPdupjoBs/7ZLsDSsJCXEoY4Su+L3LCpbXMK4nBwbQY="
	spec: [
		1,
		0,
	]
}
```

kumori SYSTEMS

kumori
SYSTEMS

The next
cloud platform