

Orchestration

Lifecycle Automation

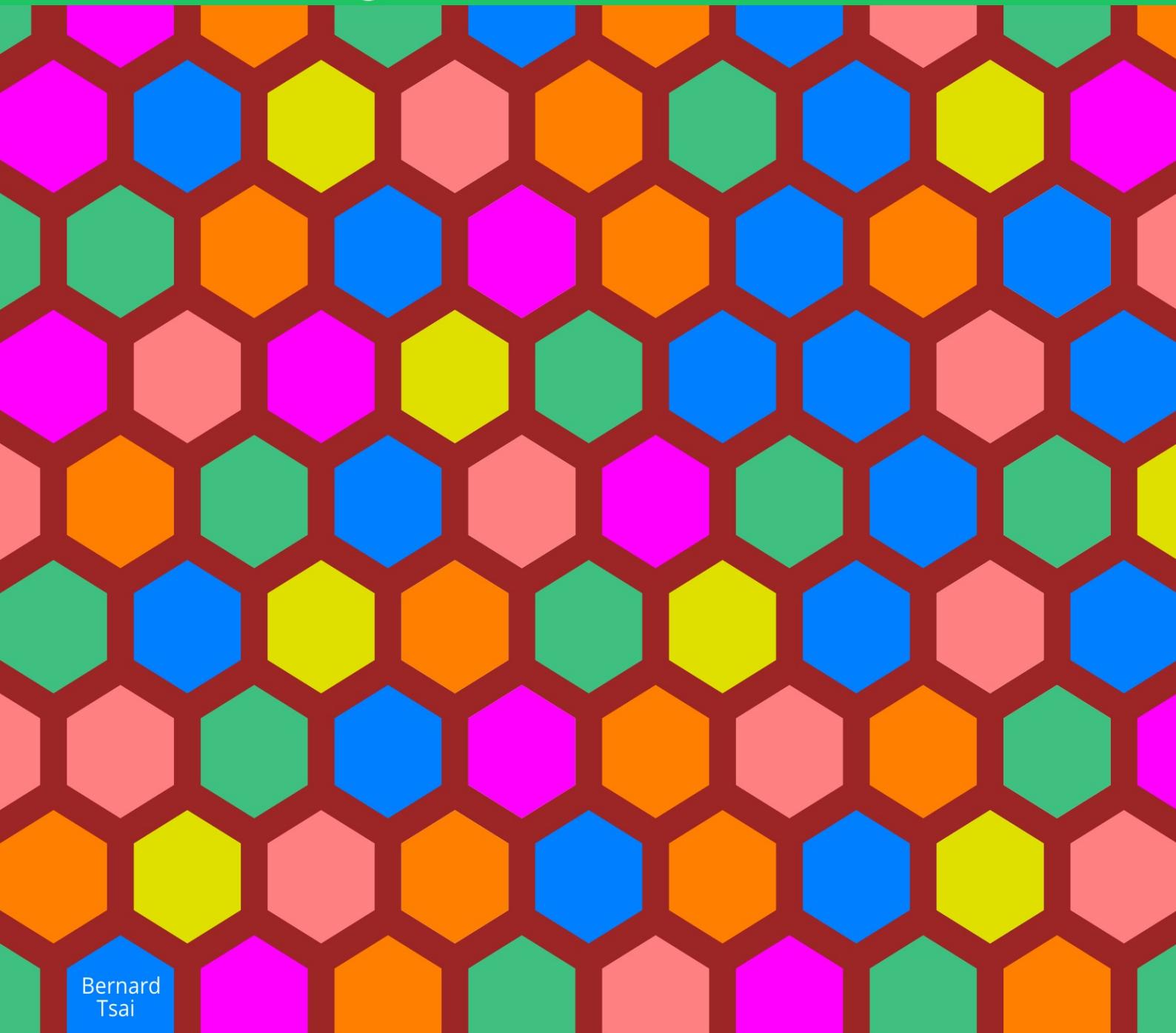


Table of Contents

Introduction

Motivation

Choreography

Concepts

Context

Decomposition

Components

Services

Versions

Endpoints

Cluster

Lifecycle Model

Dependencies

Closed-Loop Automation

Configuration

Model

Catalog

Architectures

Solutions

Automation

Orchestrator

Application Programming Interface

Command Line Interface

Message Bus

Repository

Engine

Monitor

Controller

Extensions

Example

Simple CMS

Architecture

Repository

Installation

Usage

Usage

Architecture

[Installation](#)

[Configuration](#)

[Usage](#)

[Glossary](#)

[Appendix](#)

[Orchestrator API](#)

[Orchestrator CLI](#)

[Orchestrator MSG](#)

[Controller API](#)

[create](#)

[destroy](#)

[start](#)

[stop](#)

[configure](#)

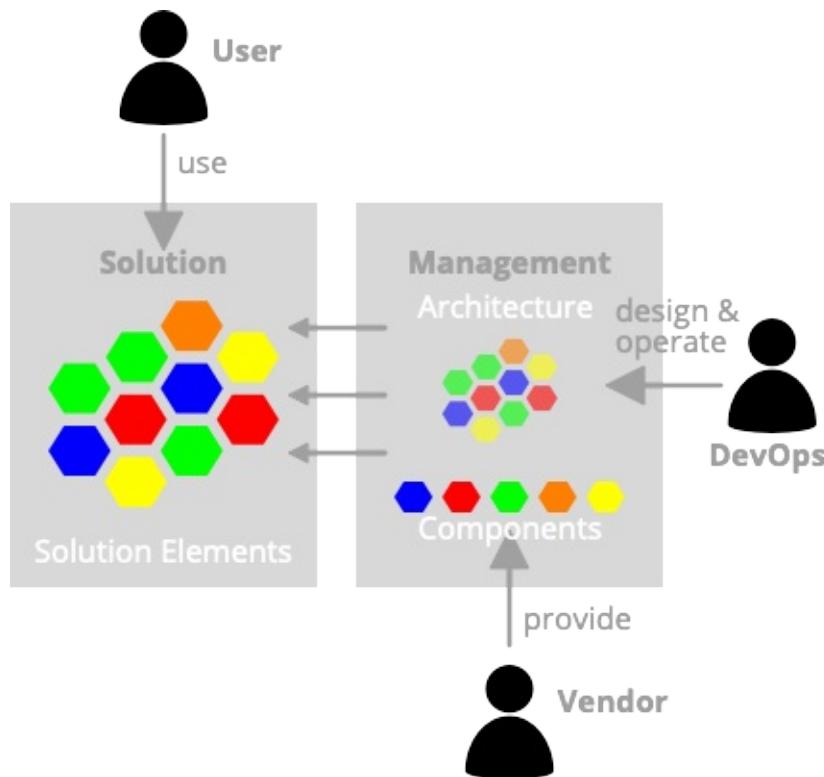
[reconfigure](#)

[reset](#)

[status](#)

Introduction

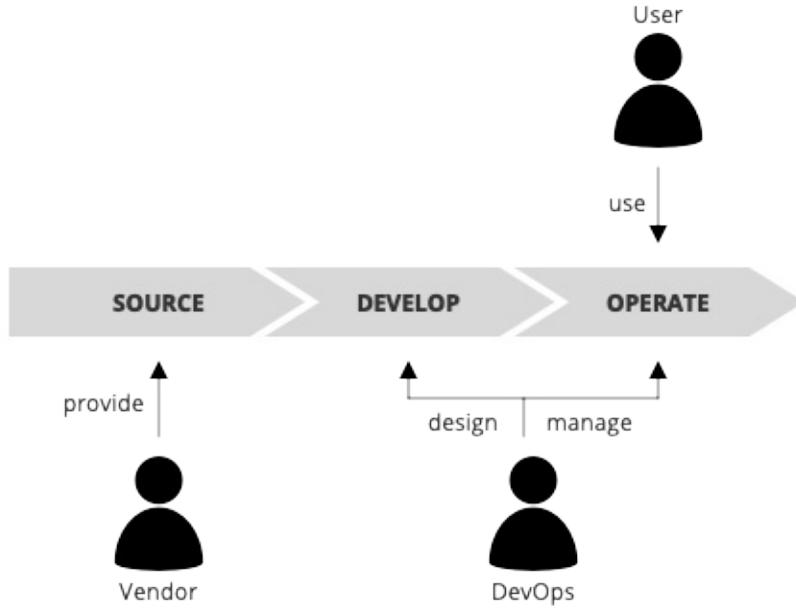
Typical solutions nowadays consist of a multitude of different **solution** elements which need to interact consistently in order to provide the desired services. These services are typically provided to the users as managed services in the sense that the user need not be concerned of how to assemble and maintain the various **solution** elements. This task can be handled by a DevOps teams which has partnered with a set of vendors providing the **solution** components from which the **solution architecture** with its elements can be derived.



It is not a trivial **task** for the DevOps team and this book will list a couple of the issues they will encounter when attempting to provide high quality services in a constantly evolving context.

The main expertise which a DevOps team needs to acquire the capability of managing the **solution** according to the varying demand of it's users and the required changes related to the lifecycle of its **solution** elements.

The value chain for a **solution** shown in the following diagram depicts how vendors of **solution** components and the DevOps team need to work together in order to provide a managed service to the users.



In the sourcing phase the vendors provide the components for the customer solutions. These serve as building blocks for the [architecture](#) which the DevOps teams design for their users in the development phase. The [architecture](#) is then instantiated in the operations phase and continuously updated in order to support the lifecycle of the managed service.

Motivation

How can we constantly change and optimise customer services which require the well-aligned interaction of various technologies and at the same time maintain overall service quality according to the customers needs.

Customer services nowadays make use of a vast amount of different technologies. Ensuring service quality for the customers therefore needs to align the various corresponding management concepts in such a way that a customer requirement is broken down into a sequence of changes on the appropriate technological components.

Technological evolution and the rapid increase of automation capabilities has lead in the previous years to a plethora of heterogenous management concepts which require rather immense integration efforts due to the complexity of the possible interaction patterns.

Taking into account that technological change is part of the game a core dilemma as stated above becomes obvious.

There is no golden bullet [solution](#) for solving this dilemma in the context of highly automated technology based services. Each approach has its advantages and disadvantages.

There are two traditional approaches of how to tackle this dilemma and an alternative concept which will be introduced in this book.

Approach A: Don't touch a running system

This approach is favoured by many Telco operators who need to guarantee a service quality of 99,999% service availability and often reduce the introduction of new technical features to an absolute minimum.

Approach B: Throw away

Why try to ensure compatibility with old hardware? Simply throw away the old technology and make use of the advantages of the new one. This has been the motto for the introduction of many storage technologies ranging from magnetic tapes, hard drives, solid state disks and cloud storage facilities. Each of the technological options addressed different qualities and found their own market leaving the consumer with the [task](#) to tackle the actual migration.

Approach C: Choreography of Managed Components and Services

This book presents a third approach of how to bridge the requirements related to change and in parallel maintaining a well defined service quality based on a choreography of managed components and services.

The disadvantage being that the various management and automation concepts need to adhere to a common way of interacting with one another. The benefits of this approach allow for the continuous integration of ever changing technological services in a seamless manner thereby focussing on the need of the customers in a consistent way.

The main abilities of such a [solution](#) approach are detailed in the chapter [Choreography](#)

Choreography of Managed Components and Services

These are the main abilities which need to be taken into consideration when addressing the dilemma:

Hyper Convergence

The ability to consistently manage a diverse variety of technologies and align possibly different functional and non-functional requirements.

Lifecycle Automation

The ability to manage the full lifecycle of all technological components and services in an automated manner.

Closed Loop Control

The ability to continuously evaluate the lifecycle state and configurations of components and services and triggering policies as needed in order to fulfill the customer requirements.

Model-Driven Automation

The ability to manage automation procedures with the help of models which define the desired outcome instead of having to provide custom code which describes for each type of requirement how to apply the needed changes.

Dependency Management

The ability to consistently handle the dependencies between different components and services when applying changes.

Fault Tolerance

The ability to recover from unforeseen errors and reestablish a comprehensive lifecycle management for all components and services.

Reconciliation

The ability to recover from manual interventions and reestablish a comprehensive lifecycle management for all components and services.

Concurrency

The ability to consistently handle multiple concurrent customer requests and changes in a well-defined manner.

Scalability

The ability to seamlessly scale with the number of managed [component](#), services as well with the number of changes regardless of the location of the managed elements.

Continuous Integration/Continuous Deployment

The ability to continuously roll-out or roll-back versions of components and services in a well-defined manner.

Near Real Time

The ability to manage changes in a close to real time timescale.

High Availability

The ability to provide highly available services even when applying changes.

Concepts

The choreography of managed components makes use of a couple of core concepts which serve as the foundation for the actual [solution](#) design:

- [Context](#): assumptions and prerequisites
- [Decomposition](#): divide and conquer
- [Components](#): everything is a [component](#)
- [Services](#): purpose of components
- [Versions](#): evolution of services
- [Endpoints](#): service channels
- [Cluster](#): no single points of failure
- [Versioning](#): evolution of [component](#) and services
- [Lifecycle Model](#): initial, inactive, active, failure
- [Dependencies](#): interaction patterns
- [Closed-Loop](#): converging towards target states
- [Configuration](#): customization of components

These concepts are detailed in the corresponding subchapters and together determine the design of the orchestration algorithm.

Context

Core assumptions and prerequisites

The [solution](#) approach is based on following assumptions and prerequisites:

- users consume services and require a certain quality of service,
- these services are provided by a set of technological components,
- each of these components can be fully controlled via automation APIs,
- there is an administrative realm allowing access to all available automation APIs,
- only some of the automation APIs will be exposed directly to the users for self-service purposes.

Decomposition

Divide and conquer

User services make use of the functionality provided by a **solution** consisting of a set of interacting technological elements. The overall **architecture** of such a **solution** can therefore be regarded as a composition of components.

Following aspects indicate how to decompose the overall **architecture** into its various elements:

- an **element** of the **solution** can be exchanged independently from the rest of the **solution**
- an **element** of the **solution** provides a specific functionality for other elements or users
- an **element** of the **solution** may have its individual lifecycle state

The result of the decomposition is map of individual technological elements identifiable by unique identifiers within the administrative realm.

PICTURE

Components

Everything is a **component**.

The concept assumes that the technological elements obtained by the **decomposition** of a **solution** share a set of common attributes, behaviours and relationships although they may each be of very different nature.

Therefore each technological **element** is regarded to be a **component** or to be more precise **component instance**. These **component** instances may be of specific **type** and may form **clusters** to ensure service availability.

Structure (C4-Model)

The C4-**Model** structures components into four fundamental aspects which may evolve independently from one another and are possibly managed by different roles.

PICTURE

1. Context: each **component** requires a dedicated environment to support the way it can evolve along its lifecycle and provide its services.
2. Core: each **component** consists of a core structure which is capable of providing the required functionality. of the **component**.
3. Configuration: each **component** may need to be configured, so that it fits into its context and provides its service according to the generic requirements of its service consumers.
4. Content: each **component** may have to make use of content information based on the usage of the service (in contrast to the **configuration** information).

The context is managed by the DevOps teams responsible for the corresponding components. The core structure is provided by the developers of the **component** itself. The **configuration** is defined by the DevOps team responsible for the **component** and the content is generated in the course of the interaction of the **component** with the service consumers.

Attributes

Instances of a **component** share following common attributes:

- Name: a unique identifier for the **instance** within the administrative realm
- Type: an indicator specifying which type of **component** the **instance** has
- Version: an indicator following the rules as specified by **Semantic Versioning**, describing which **version** of the **component** type the **instance** relates to.
- State: **lifecycle state** of the **instance**.
- Endpoint: **service interface** via with the **service** functionality can be accessed.
- Configuration: the **configuration** information controllers require to configure a **component** instance.

Behaviour

Each **component instance** will provide a type specific **service**. The common behaviour which all instances of components expose in addition to that is related to the transitions defined in the common **lifecycle model** and a status operation which allows to determine which lifecycle state a **component instance** has:

- Status: queries the current state of a [component instance](#)
- Install: provision a [component instance](#) as defined in a descriptor
- Start: activate a [component instance](#)
- Stop: deactivate a [component instance](#)
- Configure: configure an inactive [component instance](#)
- Reconfigure: configure an active [component instance](#)
- Deinstall: remove an inactive [instance](#)
- Cleanup: remove a [component instance](#) after it has failed

Relationships

There are three main types of core relationships defining how [component](#) instances need to interact amongst one another:

- Context-[Component](#): defining which [component](#) environment a [component](#) requires in order to be instantiated
- Client-Service: a service [relationship](#) defining which components need to provide services to a [component](#) so that it is able to provide its own service
- Cluster-Node: the [component](#) instances providing similar functionality need to coordinate their Lifecycle transitions in order to ensure overall service availability from a [cluster](#) perspective

PICTURE

Services

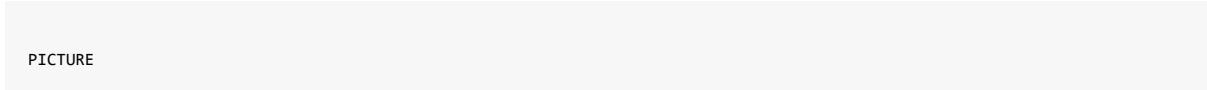
Services are the purpose of a [component](#)

Components as technological elements provide a type specific functionality to other components and/or users (service consumers). This functionality is the actual purpose of a [component](#) and is called its "service".

Which kind of functionality is provided is determined by the type and [version](#) of the [component](#) following the rules of [Semantic Versioning](#).

The interactions between components is governed by the services and the graph of service relationships between different components will be named "service-network". It is essential for the service quality from a user perspective that the service network is always kept in a consistent state (i.e. no failing service interactions).

PICTURE



Versions

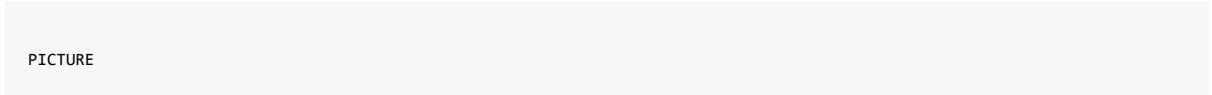
Versions mark how components change their purpose over time

Over time components will have to change the functionality they provide to their service consumers (other components or users). These changes need to be accounted for and are captured by tagging each specification with a version tag as specified by the rules for [Semantic Versioning](#).

The rules state that given a version number MAJOR.MINOR.PATCH, one needs to increment the:

1. MAJOR version when an incompatible change is made,
2. MINOR version when functionality is added in a backwards-compatible manner, and
3. PATCH version when backwards-compatible bug fixes are applied.

PICTURE



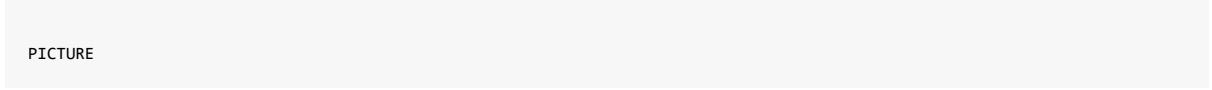
Endpoints

Channels for accessing services

Endpoints describe how the services of a [component](#) can be accessed. The service consumers need to able to interpret this information in order to access the [service](#) of the service providing [component](#).

Endpoints may change over time and the distribution of any changes to the endpoint information needs to be well organised in order to ensure a consistent service network.

PICTURE



Cluster

Clusters prevent single points of failure for services.

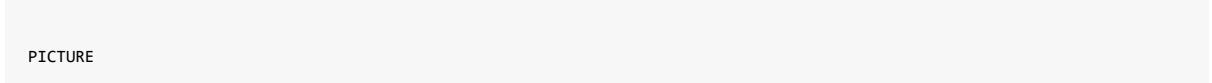
Several [component](#) instances of a specific type can group together to form a [cluster](#). These clusters follow the same [lifecycle model](#) as individual components instances.

The service [endpoints](#) of [component](#) instances are exposed by a [component cluster](#) in a stable manner ensuring that changes in the lifecycle state of individual [component](#) instances do NOT change the service endpoint of the whole [cluster](#).

This allows service consumers to access a service independent of the availability of a single [component instance](#). Service consumers should therefore only bind to the service endpoints of a [component cluster](#) and not to the endpoints of individual [component](#) instances.

The only times a service endpoint of a [component cluster](#) changes is when it transitions to or from the active state.

PICTURE

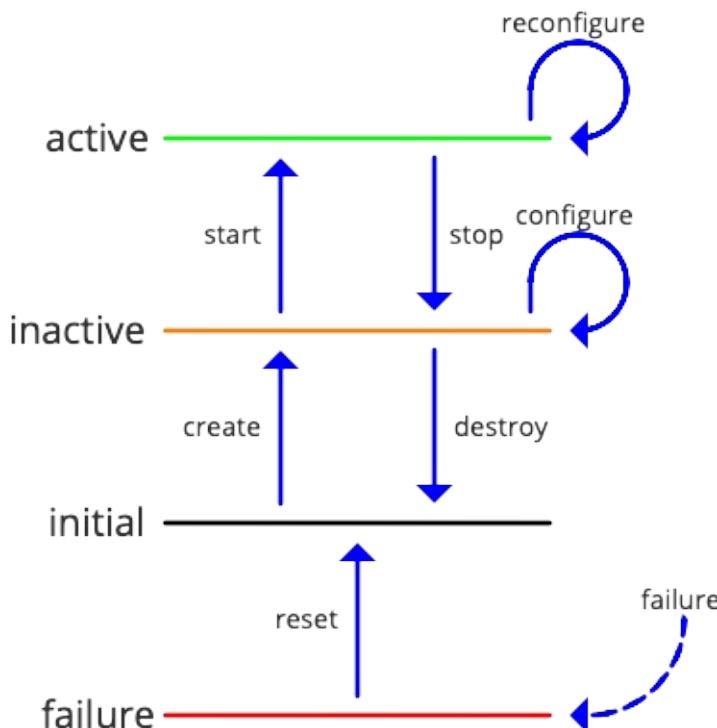


Lifecycle

All components and [component clusters](#) follow the exactly same [lifecycle model](#).

Each [component](#) regardless of type or version and every [component cluster](#) is governed by a generic [lifecycle model](#) which defines in which states these entities may be and which transitions are allowed between these aforementioned states.

These states and transitions are depicted in the following diagram:



The lifecycle of the components and clusters runs through following states:

- Initial State:

In the initial state the definition of the [component](#) has been stored in a [repository](#) or catalogue system which maintains the descriptor along with all corresponding software artefacts to bring this [component](#) to life with the help of an [orchestrator](#) and the type specific controllers.

Create transition

The [orchestrator](#) may be triggered to create the [component](#) and initiate the creation procedure for an [instance](#) of the [component](#).

It will have to ensure that the components runtime context is already available and active so that this transition can be conducted.

If the transition is successful within a certain time period the [component](#) is regarded to be inactive.

In the case the [component](#) cannot be installed due to time-outs or other errors even after a limited set of repetitions the [component](#) is regarded to be in the failure state.

- Inactive State:

In the inactive state the **component** already resides in its runtime context although it might not be configured and activated to provide services to its consumers

The **orchestrator** may also be triggered to either configure, start or destroy the **component instance**.

Configure Transition

The **orchestrator** may now be triggered to configure the **component** instance and initiate the **configuration** procedure. It will have to ensure that all of the components service dependencies at least have already been configured so that the corresponding endpoint information is available and that this transition can be conducted with the help of this information.

If the transition is successful within a certain time period the **component instance** is still regarded to be inactive.

In the case the **component instance** cannot be configured due to time-outs or other errors even after a limited set of repetitions the **component** is regarded to be in the failure state.

Start Transition

In the case of the start trigger the **orchestrator** initiates the starting procedure which will after successful termination change the state of the **component instance** to active.

In the case the **component instance** cannot be started due to time-outs or other errors even after a limited set of repetitions the **component** is regarded to be in the failure state.

Destroy Transition

If the trigger was destroy the **orchestrator** destroys the **component instance**. This will change the state if the **component instance** to initial again.

In the case the **component instance** cannot be destroyed due to time-outs or other errors even after a limited set of repetitions the **component** is regarded to be in the failure state.

- Active State:

In the active state the **component instance** has been deployed into its runtime context, has obtained all the information it requires to access the services of all the components it depends upon and now actively exposes its services to the service consumers.

The **orchestrator** may now be triggered to either stop or reconfigure the **component instance**.

Stop Transition

In the case of the stop trigger the **orchestrator** notifies the service consumers that the **component instance** will cease to provide its services and initiate the stopping procedure which will after successful termination change the state of the **component instance** to inactive.

In the case the **component instance** cannot be stopped due to time-outs or other errors even after a limited set of repetitions the **component** is regarded to be in the failure state.

Reconfigure Transition

If the trigger was update the **orchestrator** will reapply the new **configuration** to the **component** in the course of the updating procedure which will after successful termination again lead the **component** to the active state. During the updating of the **component** will still maintain the availability of its services.

Depending on the nature of the **component** this behaviour might or might not be supported. This behaviour is required for clusters which need to provide a 24x7 service, whereas components which serve as a part of a **cluster** should either be created anew with the correct **configuration** and then integrated into the **cluster** or first be

stopped and taken out of the [cluster](#), then reconfigured and after that started and taken into the [cluster](#) again.

In both cases, if the transition is not successful due to time-outs or other errors even after a limited set of repetitions the [component](#) is regarded to be in the failure state.

- Failure State:

There may be several reasons for a [component instance](#) to have reached the failure state in which the status of the [component instance](#) is undetermined and therefore regarded to be unusable.

Reset Transition

As soon as the [orchestrator](#) has identified that a [component](#) has reached this state it needs to inform all of its service consumers and then initiate a reset procedure which should free any bound resources and as a result transfer the [component](#) into the initial state again.

Dependencies

Components interact along well defined interaction patterns.

There are three main types of relationships between components or instances of components. These define the rules of how these elements need to interact with one another and the main reason for requiring an [orchestrator](#) to synchronise the activities of the controllers as type specific management components.

- [Runtime Context](#): runtime environment for components
- [Service Context](#): client-service relationships between components
- [Cluster Context](#): interactions within a [component cluster](#)

Runtime Context

Components can provide the runtime environment for other components.

Often components require a specific environment in which they will need to be provisioned. This is the runtime environment of a [component](#).

Relating this constraint to the [lifecycle model](#) of components immediately implies that components can not be created if the components which are providing their runtime environment are not in the active state.

Vice versa if a runtime environment leaves the active state it is necessary to inform the dependent components that they will have to be destroyed as well in the course of this process.

PICTURE

Service Context

Components can provide backend services for other components.

Often components require a services from other components in order to be capable of providing their own service. These backend components are the service environment of these components.

Relating this constraint to the [lifecycle model](#) of components immediately implies that components can not be activated if the components which are providing their service environment are not in the active state.

Vice versa if a [component](#) of a service environment leaves the active state it is necessary to inform the dependent components that they will have to be deactivated as well in the course of this process.

PICTURE

Cluster Context

[Component](#) instances interact in clusters to ensure service availability.

Several [component](#) instances of a specific type can group together to form a [cluster](#).

The service [endpoints](#) of [component cluster](#) are exposed by a [component cluster](#) in a stable manner ensuring that changes in the lifecycle state of individual [component](#) instances do NOT change the service endpoint of the whole [cluster](#).

Relating this constraint to the [lifecycle model](#) of [component](#) instances immediately implies that [component](#) instances can not be activated or deactivated without informing their siblings by triggering a corresponding reconfigure transition.

PICTURE

Closed Loop Automation

Control loop to converge the current state towards the desired target state.

A set of actors are involved in the closed loop automation procedure described in this subchapter.

Actors

DevOps

It is assumed that the administration of the [solution](#) providing customer services is handled by a DevOps team which understands the design of the [solution](#) and can trigger required administrative changes.

This team will also serve as escalation path if the automation procedures are not able to handle [configuration](#) requests even after reiterating the request a limited amount of times.

Controllers

In reality it will be impossible to implement a single automation [engine](#) capable of handling all lifecycle transitions for every [component](#) type. Therefore it is assumed that a set of type specific controllers will be available which can transition [component](#) instances along the transition paths defined in the [lifecycle model](#).

These controllers will receive intent based requests which specify the desired lifecycle state for a [component instance](#). These requests need to be handled in a fault-tolerant idempotent way so that issues which may have occurred during the execution could as a first option be addressed by restating the request.

Orchestrator

Controllers specialise on managing the lifecycle of [component](#) instances of a specific type. Since the components need to interact it is necessary to have an automation [element](#) which aligns the automation procedures between the various controllers, i.e. the [orchestrator](#).

Catalog and Repository System

The [solution](#) designers describe the structure of the [solution](#) with the help of a set of descriptors. These also specify which software artefacts are required by the controllers in the course of the closed loop automation.

The descriptors are stored in the [catalog](#) system and the artefacts need to be uploaded to the [repository](#) system.

Monitoring

The monitoring system continuously captures the status of the defined [component](#) instances by querying the corresponding controllers and forwarding this information to the [orchestrator](#).

Closed Loop

Picture

[Model](#) based closed loop automation makes use of following process steps in order to implement an intent based [solution](#) management approach.

1. Service Design: DevOps onboards a service descriptor along with all of its corresponding artefacts into a [catalog/repository](#) system. It is assumed for simplicity reasons that the validity of the service description has been ensured by test procedures before the onboarding takes place.

2. Service Provisioning: DevOps triggers the provisioning of a service by detailing parameters for the **component** instances and passing this way the intent, i.e. desired target state of the **solution** to the **orchestrator**.
3. Delta Calculation: The **orchestrator** compares the desired target state with the current state, i.e. the current information it has of the **solution**. The differences for each **component instance** are identified and passed as delta list containing information about which **component** instances need to be created, reconfigured or destroyed.
4. Sequencing: The **orchestrator** triggers all required lifecycle state changes in parallel for each **component instance** of the delta list. The **orchestrator** calculates the prerequisites for each transition based on the **relationship** information captured in the descriptor and triggers the corresponding required transitions. This may lead to additional ripple effects as further prerequisites may have to be met.
5. Transitions: The transition requests are forwarded from the **orchestrator** to the corresponding type specific controllers which then transition the **component instance** along the path of the **lifecycle model**. During this operation the **orchestrator** informs the monitoring system as well, that the specific **component instance** has a new desired target state and should be monitored accordingly.
6. Monitoring: The monitoring system continuously retrieves state information of each **component instance** by querying their controllers and forwards this information to the **orchestrator** so that it can update its status information of the **instance** specific current state and trigger a new delta calculation.

This automation approach constantly compares the current state with a desired target state and will eventually converge the **solution** in a fault tolerant way to the state specified by the intent of the DevOps team.

Configuration

Configuration customises a component to the environment and general requirements of its service consumers.

The configuration of a component is a type and possibly version specific task which can not be handled in a generic fashion but has to be executed by the corresponding controller.

Nevertheless it is possible to distinguish between following aspects of the required configuration information:

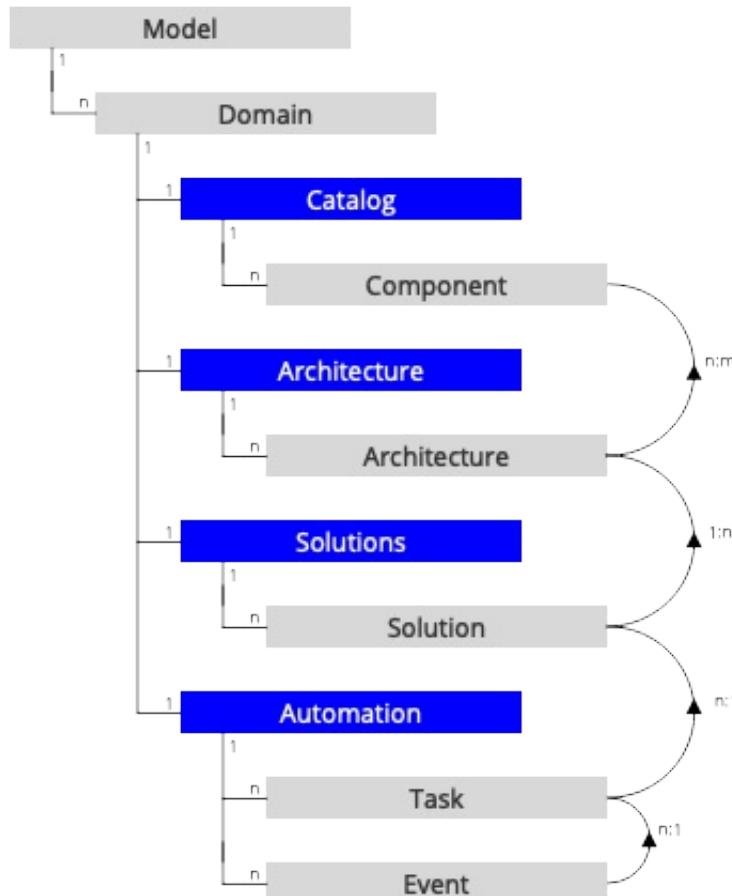
- the data a component requires to match its runtime-context
- the data a component requires to match its service-context
- the data a component requires to match its cluster-context
- the data a component requires to parametrise its generic service behaviour

PICTURE

Model

Intent based automation requires a well structured information [model](#).

The following diagram provides an overview of the information [model](#):



The information [model](#) or **Model** is divided into administration realms or domains. This allows for sharding the information [model](#) into substructures which can be managed independently from one another if needed.

Model

The [model](#) entity is the central entry point to the information required by the closed-loop automation procedures. It is subdivided into a set of subdomains which can be administrated independently from one another.

Attribute	Type	Description
Schema	string	schema of the model
Name	string	name of the model
Domains	map to Domains	map of names to domains

Domain

A [domain](#) is an administrative realm which can be managed independently from other domains and may serve as a context for sharding the [model](#).

Attribute	Type	Description
Name	string	name of the model
Catalog	map to Components	map of names to components
Architectures	map to Architectures	map of names to architectures
Solutions	map to Solutions	map of names to solutions
Tasks	map to Tasks	map of uuids to tasks
Events	map to Events	map of uuids to events

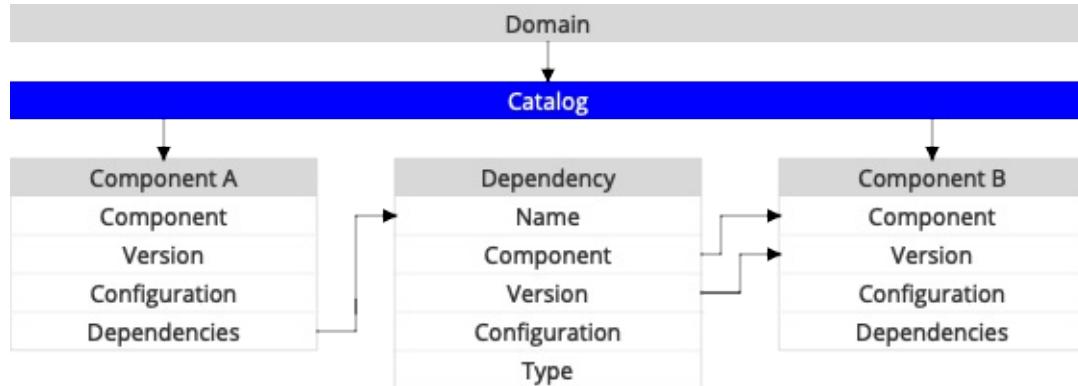
A [domain](#) holds four types of information:

- **Catalog:** the [catalog](#) holds the templates for all components including their version information, their dependencies amongst one another and a basic generic [configuration](#). The [catalog](#) can therefore be regarded as a [repository](#) of building blocks for the elements of a [solution](#).
- **Architectures:** The architectures are designed to depict what a [solution](#) should look like and in this way resemble the desired target state of a [solution](#). They describe the structure of the solutions by listing the required [solution](#) elements (which are of a specific [component](#) type), their [element](#) specific configurations including which [element](#) will need to fulfill the dependencies as defined in the [component](#) templates. Each [solution architecture](#) maintains version information since they will need to evolve over time.
- **Solutions:** the actual runtime information (current state) of a [solution](#) and its elements is registered here to allow for closed loop automation. It includes information related to the runtime [configuration](#), the status and the service endpoints of the elements.
- **Automation:** For a closed-loop automation it is also necessary to maintain information regarding which tasks need to be conducted to align the current state with the desired target state and react to events which are triggered by the execution of tasks.

The schema file in swagger: [swagger-model.yaml](#)

Domain-Catalog

The [catalog](#) is a subset of the [information model](#) of a [domain](#) which holds templates (components and their dependencies) for [solution](#) elements along with their basic [configuration](#).



Component

A [component](#) is a versioned template for a [solution element](#). It describes its basic [configuration](#) with the help of a structured object and a set of named dependencies to other components

Attribute	Type	Description
Component	string	name of the component
Version	string	version of the component
Configuration	object	basic configuration object
Dependencies	map to dependencies	map of names to dependencies

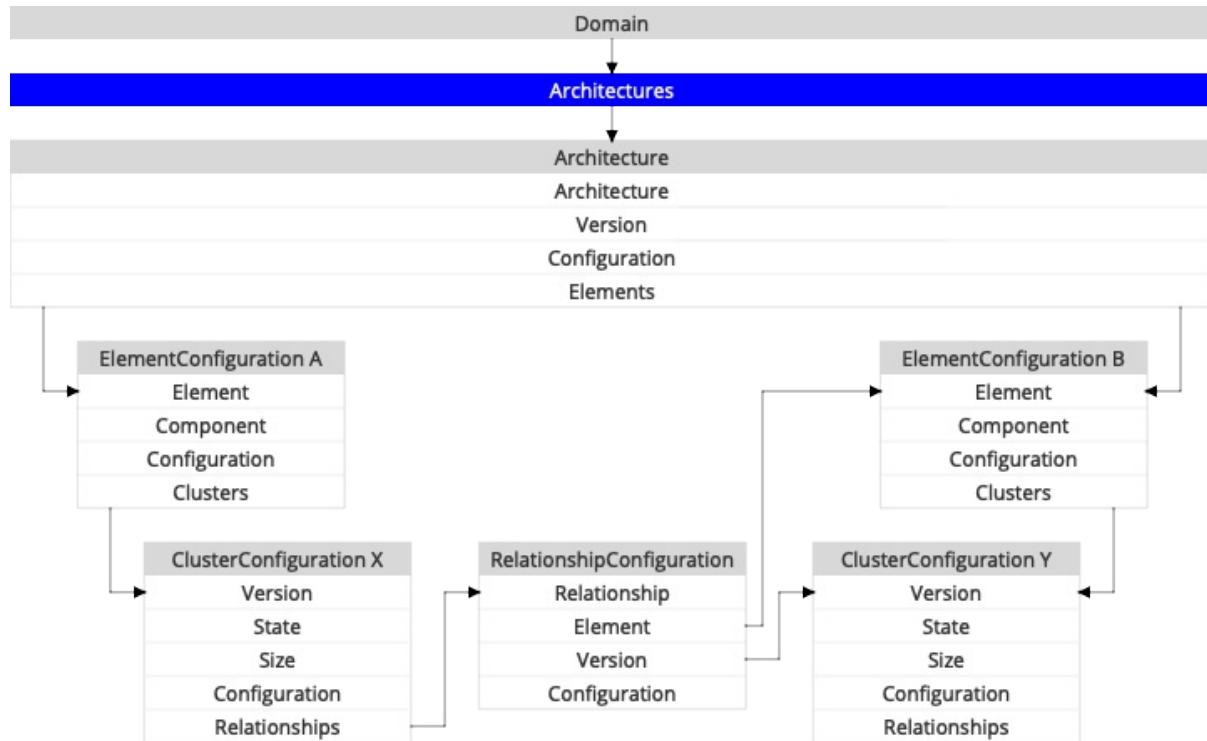
Dependency

A dependency describes how the containing [component](#) relates to other components. It can either be a "context" dependency which defines that the referenced [component](#) needs to be "active" before the [component](#) can be created, or it may be a "service" dependency which states that the referenced [component](#) needs to be "active" before the [component](#) can be started. The [configuration](#) attribute is a structured object with the basic [configuration](#) for the establishing the [component relationship](#) described in the dependency.

Attribute	Type	Description
Name	string	name of the dependency
Type	string	type of dependency ("context"/"service")
Component	name of the component	name of the referenced component
Version	version of the component	version of the referenced component
Configuration	object	basic configuration object of the dependency

Domain-Architectures

Architectures are a subset of the [information model](#) of a [domain](#) describing the desired structure and [configuration](#) of a [solution](#) with the help of a list configurations for [solution](#) elements and their relationships amongst one another.



The configurations are "design-time" configurations. Since they can only describe the [configuration](#) parameters which are known at the time the [solution](#) is being designed.

Architecture

An [architecture](#) is a versioned description of the desired [configuration](#) of a [solution](#). The name of the [architecture](#) matches the name of the [solution](#). It holds a general structured [configuration](#) object (with general parameters) and a set of configurations for [solution](#) elements which can be identified by their names.

Attribute	Type	Description
Architecture	string	name of the architecture
Version	string	version of the architecture
Configuration	object	architecture configuration object
Elements	map to ElementConfigurations	map of names to configurations

ElementConfiguration

An [ElementConfiguration](#) describes the [configuration](#) of a [solution element](#) based on a [component](#) type. A [solution element](#) may need to comprise several versions of a [component](#) (Clusters) which may require their own specific configurations.

Attribute	Type	Description
Element	string	name of the solution element
Component	string	name of the component
Configuration	object	element configuration object
Clusters	map to ClusterConfigurations	map of names to configurations

ClusterConfiguration

A ClusterConfiguration describes the [configuration](#) for a [cluster](#) within a [solution element](#) for a specific version of a [component](#) type. A [cluster](#) has a certain size and should be in a certain state of the lifecycle. Due to its dependencies to other [solution](#) elements it will also have to describe this information in its relationships attribute.

Attribute	Type	Description
Version	string	version of the component
State	string	desired lifecycle state
Size	positive integer	desired cluster size
Configuration	object	cluster configuration object
Relationships	map to RelationshipConfigurations	map of names to configurations

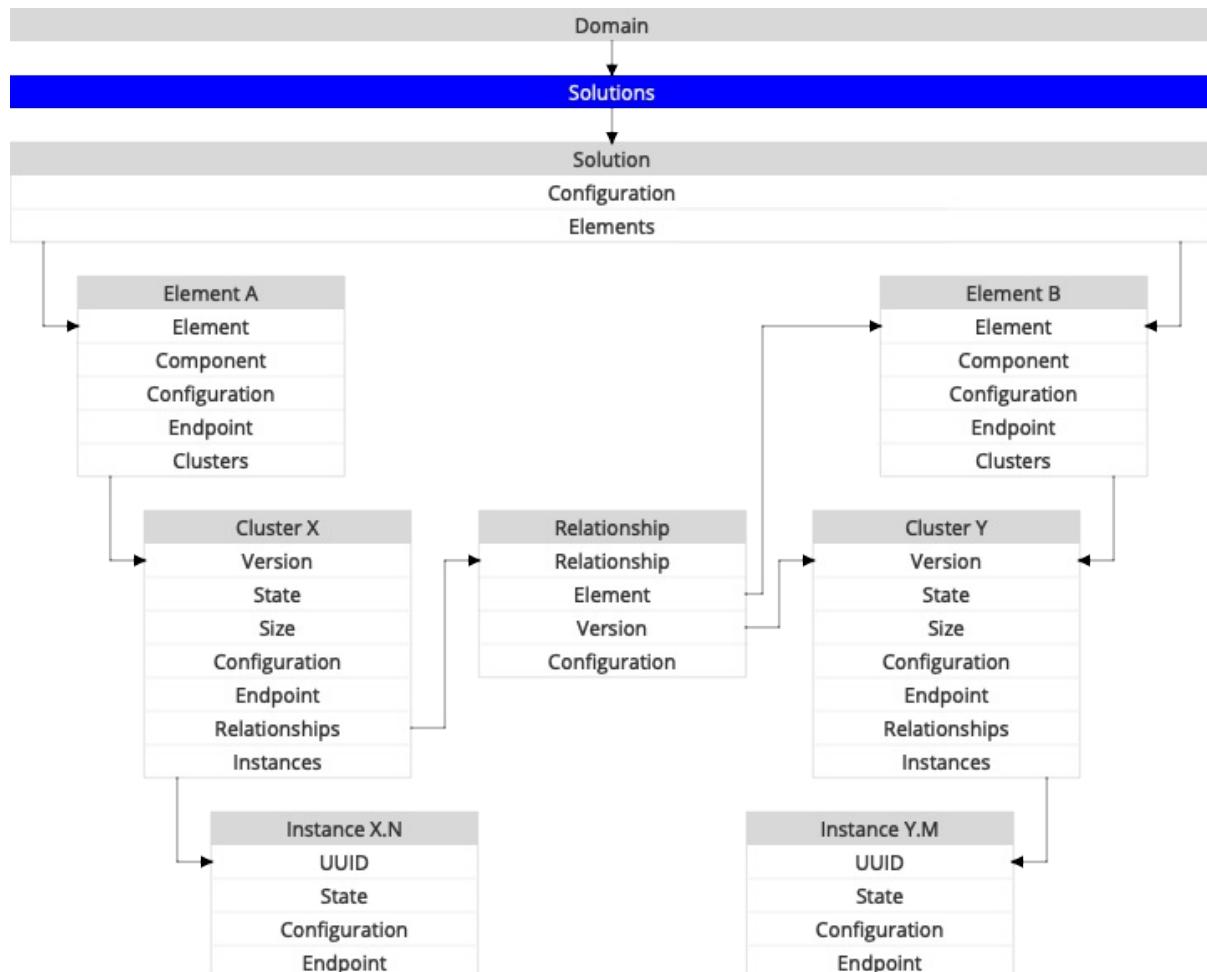
RelationshipConfiguration

A RelationshipConfiguration describes the [configuration](#) for a [relationship](#) between clusters of related solution elements.

Attribute	Type	Description
Relationship	string	name of the dependency
Element	string	name of the referenced element
Version	string	version of the referenced element
Configuration	object	relationship configuration object

Domain-Solutions

Solutions are a subset of the [information model](#) of a [domain](#) describing the current structure and [configuration](#) of a [solution](#) with the help of a list actual configurations for [solution](#) elements and their relationships amongst one another.



The configurations are "run-time" configurations. Since they also describe those [configuration](#) parameters which can only be known at the time the [solution](#) is being provisioned.

Solution

A [solution](#) entity describes the current [configuration](#) and state of a [solution](#). Its name matches the name of the [architecture](#) describing its design.

It holds a general structured [configuration](#) object (with general parameters) and a set of configurations for [solution](#) elements which can be identified by their names.

Attribute	Type	Description
Solution	string	name of the solution
Configuration	object	runtime configuration object
Elements	map to Elements	map of names to elements

Element

An [Element](#) describes the runtime [configuration](#) of a [solution element](#) based on a [component](#) type. A [solution element](#) may need to comprise several versions of a [component](#) (Clusters) which may require their own specific configurations.

Attribute	Type	Description
Element	string	name of the solution element
Component	string	name of the component
Configuration	object	runtime configuration object
Endpoint	object	service endpoint
Clusters	map to Cluster	map of names to clusters

Cluster

A [Cluster](#) describes the runtime [configuration](#) for a [cluster](#) within a [solution element](#) for a specific version of a [component](#) type. A [cluster](#) has a certain size and should be in a certain state of the lifecycle. Due to its dependencies to other [solution](#) elements it will also have to describe this information in its relationships attribute.

Attribute	Type	Description
Version	string	version of the component
State	string	desired lifecycle state
Size	positive integer	desired cluster size
Configuration	object	runtime configuration object
Endpoint	object	service endpoint
Relationships	map to Relationships	map of names to relationships
Instances	map to Instances	map of names to instances

Relationship

A [Relationship](#) describes the runtime [configuration](#) for a dependency between clusters of related [solution](#) elements.

Attribute	Type	Description
Relationship	string	name of the dependency
Element	string	name of the referenced element
Version	string	version of the referenced element
Configuration	object	runtime configuration object

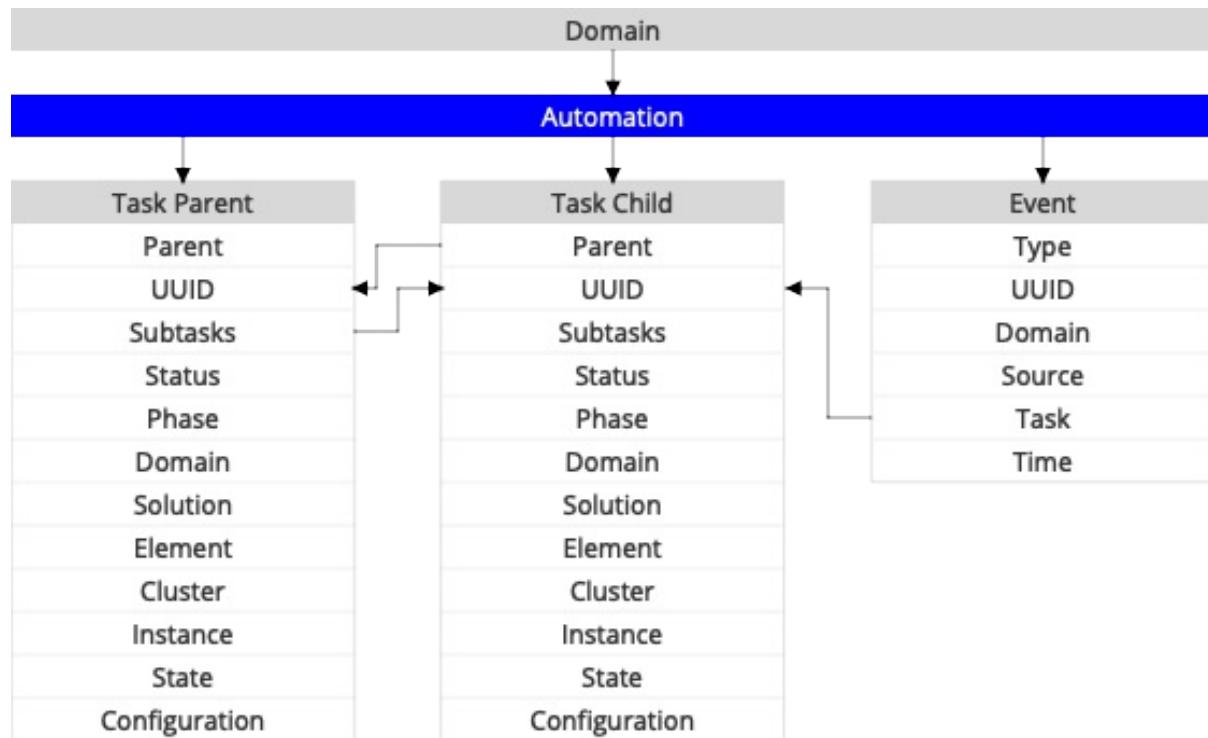
Instance

An [Instance](#) describes an [instance](#) of a [cluster](#) of a [solution element](#). This comprises its state, runtime configuration and service endpoint.

Attribute	Type	Description
UUID	string	unique identifier of the instance
State	string	current lifecycle state
Configuration	object	runtime configuration object
Endpoint	object	service endpoint

Domain-Automation

Tasks and events form the automation subset of the [information model](#) of a [domain](#) describing the current structure and [configuration](#) of a [solution](#).



Task

A [task](#) describes an activity to manipulate the [solution](#).

Attribute	Type	Description
Type	string	type of the task
UUID	string	unique identifier for the task
Parent	string	identifier of the parent task
Subtasks	array of UUIDs	array of UUIDs of subtasks
Status	string	status of the task
Phase	string	execution phase of the task
Domain	string	name of the domain
Solution	string	name of the solution
Element	string	name of the element
Cluster	string	name of the cluster
Instance	string	name of the instance
State	string	desired lifecycle state

Configuration

object

runtime configuration object

Event

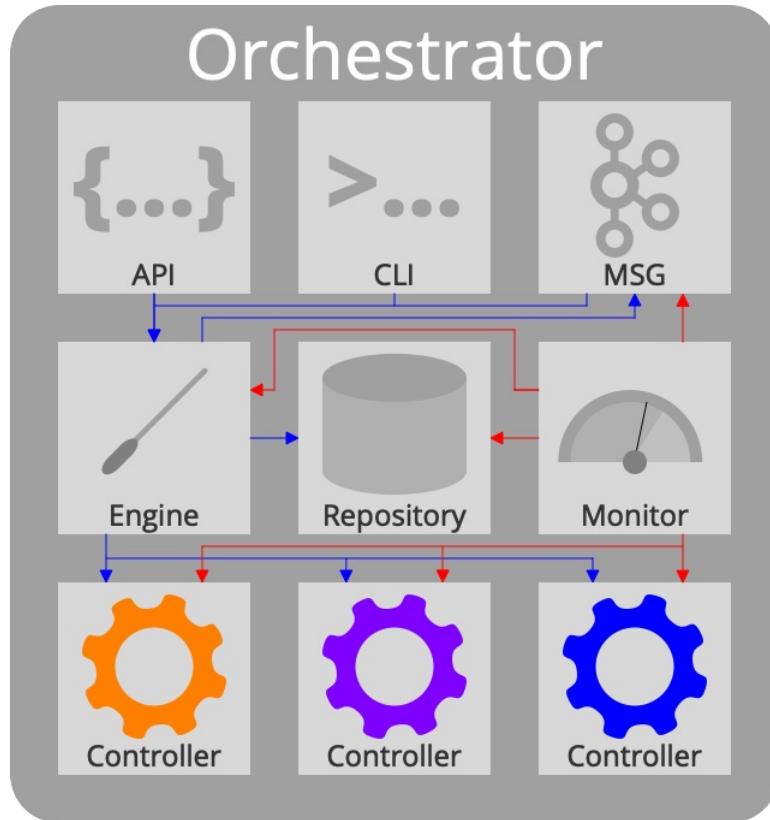
An [event](#) notifies about the occurrence of a [task](#) related [event](#).

Attribute	Type	Description
Type	string	type of event
UUID	string	unique identifier for the event
Domain	string	name of the domain
Source	string	identifier of the source task
Task	string	identifier of the task
Time	string	time since 1.1.1970 in nsecs

Orchestrator

A plug and play [architecture](#) to manage the lifecycle of various types of [solution](#) elements in a consistent manner in order to provide a managed customer service.

The diagram below shows the high-level [architecture](#) of the orchestration [engine](#) capable of flexibly coordinating the lifecycles of interdependent [solution](#) elements.



The [architecture](#) consists of three layers. The northbound interfaces allow for the interaction with the DevOps team and external systems and provide:

- [API](#): an application programming interface via REST
- [CLI](#): a command line interface for the operators
- [MSG](#): a message bus interface for sending and receiving events

The core modules of the [orchestrator](#) facilitate the closed loop automation of the lifecycle of [solution](#) elements and consist of:

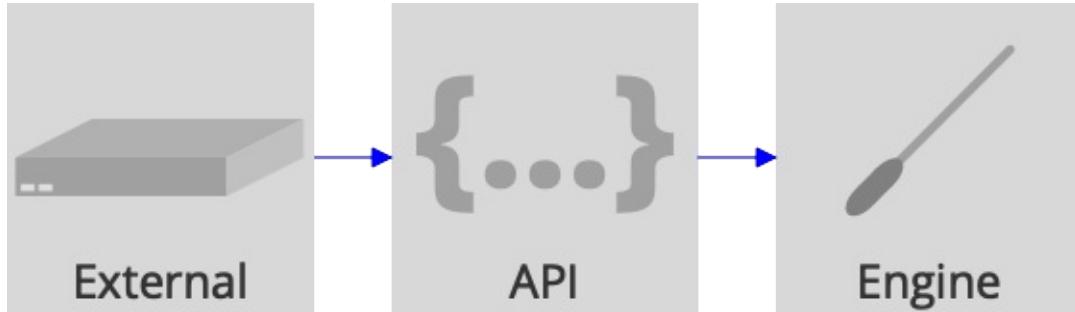
- [Engine](#): the orchestration [engine](#),
- [Repository](#): the data store and
- [Monitor](#): the monitoring system.

The [southbound controllers](#) represent specific code fragments capable of managing the lifecycle of [solution](#) elements of a specific type according to the common [lifecycle model](#).

Application Programming Interface

The [orchestrator](#) API exposes all required capabilities for the lifecycle automation of solutions covering the sourcing, design and operations phase to external systems via REST interface.

The application programming interface is exposed to external systems and serves as a gateway to expose the capabilities of the orchestration [engine](#).



The API provides interfaces to

- support the administration,
- manage templates in a [catalog](#),
- manage architectures and configurations and
- manage the lifecycle of solutions.

Administration

The API will need to support following administrative use cases to facilitate the management of independent domains by a DevOps team:

- ResetModel: reset [model](#) in the [repository](#) to initial state
- StoreModel: create [model](#) from yaml data and store it into the [repository](#)
- RetrieveModel: retrieve [model](#) from the [repository](#) as yaml data
- ListDomains: list all the available domains
- StoreDomain: create a new [domain](#) from yaml data and store it in the [repository](#)
- RetrieveDomain: retrieve [domain](#) from the [repository](#) as yaml data
- RemoveDomain: remove [domain](#) from the [repository](#)

Catalog Management

[Catalog](#) management is conducted in the context of a specific [domain](#). The API will need to support the following use cases for the sourcing phase to support the management of templates for components.

- ListComponents: list all the available components in a [domain](#)
- ListComponentVersions: list all the available versions of a [component](#) in a [domain](#)
- StoreComponent: create a new [component](#) from yaml data and store it in the [repository](#)
- RetrieveComponent: retrieve [component](#) from the [repository](#) as yaml data
- RemoveComponent: remove [component](#) from the [repository](#)

[Component](#) templates are regarded to be immutable. Therefore no update operations are provided for components.

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
 - a [component](#) references unknown components,
 - an attempt is made to redefine a [component](#) or
 - an unknown [component](#) is referenced when intending to show or delete a [component](#).
-

Architecture Management

[Architecture](#) Management is also conducted in the context of a specific [domain](#). The API will need to support the following use cases for the design phase to support the management of [solution](#) architectures.

- ListArchitectures: list all the available architectures in a [domain](#)
- ListArchitectureVersions: list all the available versions of an [architecture](#) in a [domain](#)
- StoreArchitecture: create a new [architecture](#) from yaml data and store it in the [repository](#)
- RetrieveArchitecture: retrieve [architecture](#) from the [repository](#) as yaml data
- RemoveArchitecture: remove [architecture](#) from the [repository](#)

Architectures are regarded to be immutable. Therefore no update operations are provided for architectures.

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
 - an [architecture](#) definition is inconsistent,
 - an attempt is made to redefine an [architecture](#) or
 - an unknown [architecture](#) is referenced when intending to show or delete an [architecture](#).
-

Solution Management

[Solution](#) Management is also conducted in the context of a specific [domain](#). The API will need to support the following use cases for the operations phase to support the management of solutions.

- ListSolutions: list all the available solutions in a [domain](#)
- StoreSolution: create a new [solution](#) from yaml data and store it in the [repository](#)
- RetrieveSolution: retrieve [solution](#) from the [repository](#) as yaml data
- RemoveSolution: remove [solution](#) from the [repository](#)
- ModifySolution: create a new [solution](#) or update an existing [solution](#) based on the information provided in an [architecture](#) blueprint
- ListElements: list all the available elements in a [solution](#)
- RetrieveElement: retrieve [element](#) from the [repository](#) as yaml data
- ModifyElement: update an existing [element](#) based on the information provided in a [configuration](#) object
- ListClusters: list all the available clusters of an [element](#)
- RetrieveCluster: retrieve [cluster](#) from the [repository](#) as yaml data
- ModifyCluster: update an existing [cluster](#) based on the information provided in a [configuration](#) object
- ListInstances: list all the available instances of a [cluster](#)
- RetrieveInstance: retrieve [instance](#) from the [repository](#) as yaml data
- ModifyInstance: update an existing [instance](#) based on the information provided in a [configuration](#) object

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
- an unknown [architecture](#) blueprint is referenced,
- an unknown [solution](#) is referenced when intending to retrieve or update a [solution](#),

- an unknown `element` is referenced when intending to retrieve or update an `element`,
 - an unknown `cluster` is referenced when intending to retrieve or update a `cluster` or
 - an unknown `instance` is referenced when intending to retrieve or update an `instance`.
-

Automation

Automation is also conducted in the context of a specific `domain`. The API will need to provide visibility and operational control over all the tasks being executed in order to manage the lifecycle of a `solution`.

- `ListTasks`: list all toplevel tasks in a `domain` (optionally matching a certain request ID)
- `RetrieveTask`: retrieve `task` from the `repository` as yaml data
- `TerminateTask`: stop a `task` in a `domain`
- `ListEvents`: list all events related to a `task` in a `domain`
- `RetrieveEvent`: retrieve `event` from the `repository` as yaml data

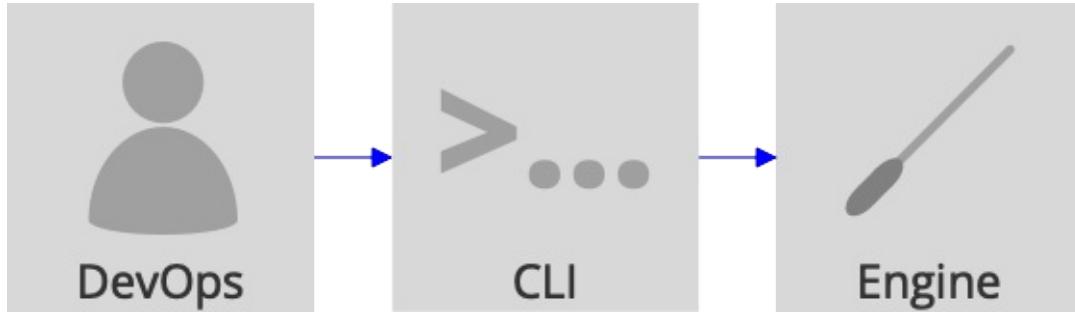
It is regarded to be an error if:

- an unknown `domain` is referenced,
- an unknown reference ID is used,
- an unknown `task` is referenced when intending to retrieve or terminate a `task` or list the corresponding events or
- an unknown `event` is referenced when intending to retrieve an `event`.

Command Line Interface

The [orchestrator](#) CLI exposes all required capabilities for the lifecycle automation of solutions covering the sourcing, design and operations phase to the DevOps team with console access.

The command line interface is exposed to the DevOps team and serves as a gateway to expose the capabilities of the orchestration [engine](#).



The CLI provides commands to

- support the administration,
- manage templates in a [catalog](#),
- manage architectures and configurations and
- manage the lifecycle of solutions.

Administration

The API will need to support following administrative use cases to facilitate the management of independent domains by a DevOps team:

- ResetModel: reset [model](#) in the [repository](#) to initial state
- StoreModel: create [model](#) from yaml data and store it into the [repository](#)
- RetrieveModel: retrieve [model](#) from the [repository](#) as yaml data
- ListDomains: list all the available domains
- StoreDomain: create a new [domain](#) from yaml data and store it in the [repository](#)
- RetrieveDomain: retrieve [domain](#) from the [repository](#) as yaml data
- RemoveDomain: remove [domain](#) from the [repository](#)

Catalog Management

[Catalog](#) management is conducted in the context of a specific [domain](#). The API will need to support the following use cases for the sourcing phase to support the management of templates for components.

- ListComponents: list all the available components in a [domain](#)
- ListComponentVersions: list all the available versions of a [component](#) in a [domain](#)
- StoreComponent: create a new [component](#) from yaml data and store it in the [repository](#)
- RetrieveComponent: retrieve [component](#) from the [repository](#) as yaml data
- RemoveComponent: remove [component](#) from the [repository](#)

[Component](#) templates are regarded to be immutable. Therefore no update operations are provided for components.

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
 - a [component](#) references unknown components,
 - an attempt is made to redefine a [component](#) or
 - an unknown [component](#) is referenced when intending to show or delete a [component](#).
-

Architecture Management

[Architecture](#) Management is also conducted in the context of a specific [domain](#). The API will need to support the following use cases for the design phase to support the management of [solution](#) architectures.

- ListArchitectures: list all the available architectures in a [domain](#)
- ListArchitectureVersions: list all the available versions of an [architecture](#) in a [domain](#)
- StoreArchitecture: create a new [architecture](#) from yaml data and store it in the [repository](#)
- RetrieveArchitecture: retrieve [architecture](#) from the [repository](#) as yaml data
- RemoveArchitecture: remove [architecture](#) from the [repository](#)

Architectures are regarded to be immutable. Therefore no update operations are provided for architectures.

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
 - an [architecture](#) definition is inconsistent,
 - an attempt is made to redefine an [architecture](#) or
 - an unknown [architecture](#) is referenced when intending to show or delete an [architecture](#).
-

Solution Management

[Solution](#) Management is also conducted in the context of a specific [domain](#). The API will need to support the following use cases for the operations phase to support the management of solutions.

- ListSolutions: list all the available solutions in a [domain](#)
- StoreSolution: create a new [solution](#) from yaml data and store it in the [repository](#)
- RetrieveSolution: retrieve [solution](#) from the [repository](#) as yaml data
- RemoveSolution: remove [solution](#) from the [repository](#)
- ModifySolution: create a new [solution](#) or update an existing [solution](#) based on the information provided in an [architecture](#) blueprint
- ListElements: list all the available elements in a [solution](#)
- RetrieveElement: retrieve [element](#) from the [repository](#) as yaml data
- ModifyElement: update an existing [element](#) based on the information provided in a [configuration](#) object
- ListClusters: list all the available clusters of an [element](#)
- RetrieveCluster: retrieve [cluster](#) from the [repository](#) as yaml data
- ModifyCluster: update an existing [cluster](#) based on the information provided in a [configuration](#) object
- ListInstances: list all the available instances of a [cluster](#)
- RetrieveInstance: retrieve [instance](#) from the [repository](#) as yaml data
- ModifyInstance: update an existing [instance](#) based on the information provided in a [configuration](#) object

It is regarded to be an error if:

- an unknown [domain](#) is referenced,
- an unknown [architecture](#) blueprint is referenced,
- an unknown [solution](#) is referenced when intending to retrieve or update a [solution](#),

- an unknown [element](#) is referenced when intending to retrieve or update an [element](#),
 - an unknown [cluster](#) is referenced when intending to retrieve or update a [cluster](#) or
 - an unknown [instance](#) is referenced when intending to retrieve or update an [instance](#).
-

Automation

Automation is also conducted in the context of a specific [domain](#). The API will need to provide visibility and operational control over all the tasks being executed in order to manage the lifecycle of a [solution](#).

- [ListTasks](#): list all toplevel tasks in a [domain](#) (optionally matching a certain request ID)
- [RetrieveTask](#): retrieve [task](#) from the [repository](#) as yaml data
- [TerminateTask](#): stop a [task](#) in a [domain](#)
- [ListEvents](#): list all events related to a [task](#) in a [domain](#)
- [RetrieveEvent](#): retrieve [event](#) from the [repository](#) as yaml data

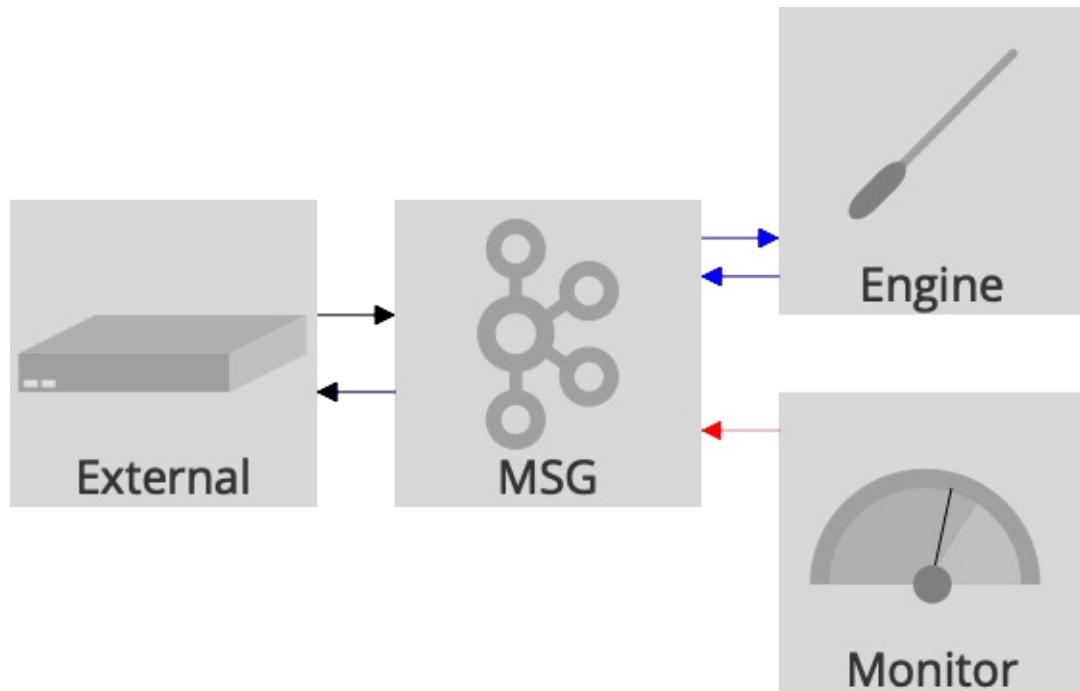
It is regarded to be an error if:

- an unknown [domain](#) is referenced,
- an unknown reference ID is used,
- an unknown [task](#) is referenced when intending to retrieve or terminate a [task](#) or list the corresponding events or
- an unknown [event](#) is referenced when intending to retrieve an [event](#).

Message Bus

The [orchestrator](#) produces lifecycle events and consumes commands to trigger lifecycle management operations.

The message bus interface is exposed to external systems and serves as a gateway to expose the capabilities of the orchestration [engine](#) and the monitoring system.



The message bus provides access to

- commands for the lifecycle management of solutions,
- [task](#) related notification events and
- status information of the the [solution](#).

Solution Management

The [orchestrator](#) can be handed a second unique topic identifier "TOPIC1" in the course of the startup process. This is the topic to which it will subscribe in order to receive commands from a [Kafka](#) broker.

The commands may relate to following types of tasks which need to be executed:

- [Architecture](#): tasks related to the instantiation or update of a [solution](#) based on an [architecture](#) blueprint
- [Element](#): tasks related to the instantiation, modification or deletion of an [element](#)
- [Cluster](#): tasks related to the instantiation, modification or deletion of a [cluster](#)
- [Instance](#): tasks related to the instantiation, modification or deletion of an [instance](#)

The attributes of the commands are:

Attribute	Type	Description
Request	string	request identifier
Type	string	type of the task

Domain	string	name of the domain
Solution	string	name of the solution
Element	string	name of the element
Cluster	string	name of the cluster
Instance	string	name of the instance
Size	integer	desired cluster size
State	string	desired lifecycle state
Configuration	object	runtime configuration object

Task Events

The [orchestrator](#) can be handed a unique topic identifier "TOPIC2" in the course of the startup process. This is the topic to which it will publish its events to a [Kafka](#) broker.

Each [event](#) is related to a [task](#) and will notify whether:

- Execution: the execution of a [task](#) has been triggered
- Timeout: a [task](#) has run into a time-out
- Failure: a [task](#) has failed
- Success: a [task](#) has completed successfully

The type of tasks to which an [event](#) may relate are:

- [Architecture](#): tasks related to the instantiation or update of a [solution](#) based on an [architecture](#) blueprint
- [Element](#): tasks related to the instantiation, modification or deletion of an [element](#)
- [Cluster](#): tasks related to the instantiation, modification or deletion of a [cluster](#)
- [Instance](#): tasks related to the instantiation, modification or deletion of an [instance](#)
- Parallel: a [task](#) representing a group of tasks which can be executed in parallel
- Sequential: a [task](#) representing a group of tasks which need to be executed sequentially

The [event](#) information captures following attributes of the [event](#) and the corresponding [task](#):

Attribute	Type	Description
Request	string	request identifier
Type	string	type of event
UUID	string	unique identifier for the event
Source	string	identifier of the source task
Time	string	time since 1.1.1970 in nsecs
Task	string	unique identifier for the task
TaskType	string	type of the task
Parent	string	identifier of the parent task
Subtasks	array of UUIDs	array of UUIDs of subtasks
Status	string	status of the task
Phase	string	execution phase of the task

Domain	string	name of the domain
Solution	string	name of the solution
Element	string	name of the element
Cluster	string	name of the cluster
Instance	string	name of the instance
State	string	desired lifecycle state
Configuration	object	runtime configuration object

Status

The [orchestrator](#) can be handed a unique topic identifier "TOPIC3" in the course of the startup process. This is the topic to which it will publish its status information to a [Kafka](#) broker.

TODO..

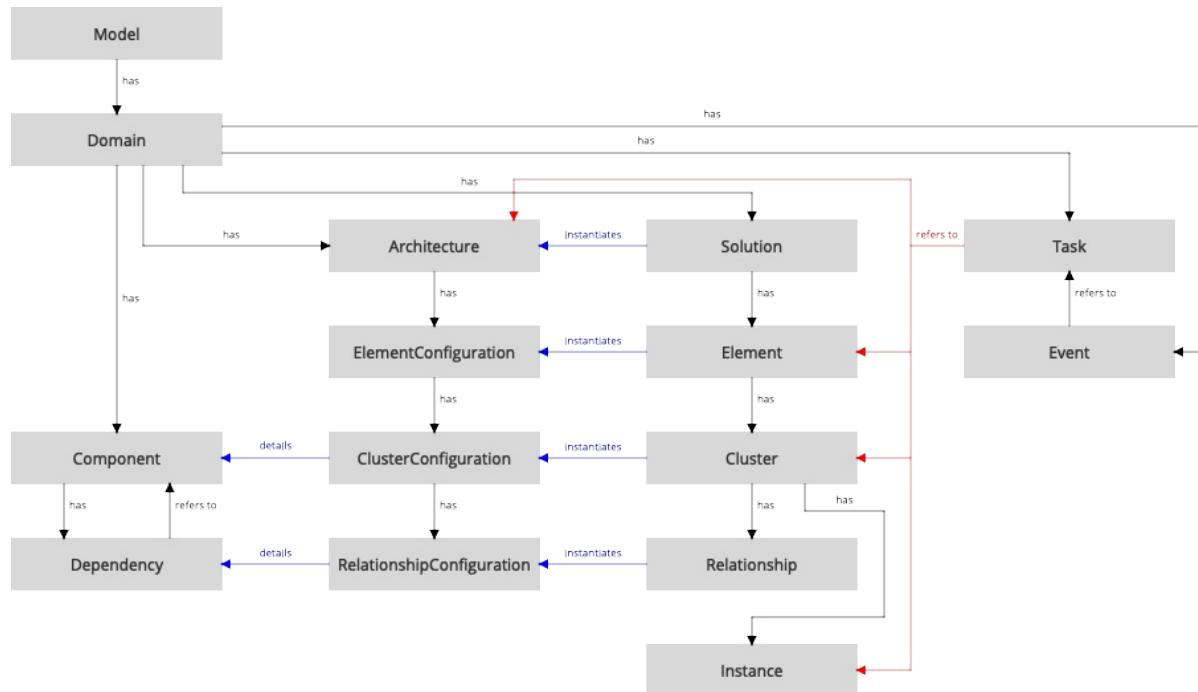
Repository

The [repository](#) maintains the entities defined in the information [model](#) supporting the sourcing, design and operations phase of a [solution](#).

The [repository](#) interacts with the [orchestrator engine](#) and [monitor](#) system to support the automation of the lifecycle management of solutions.



The following diagram provides an overview of the [model](#) maintained in the [repository](#).



CATALOG

ARCHITECTURE

SOLUTION

AUTOMATION

The [model](#) follows the structure defined in the [information Model](#) and is clearly subdivided into domains which allow for differentiating administrative realms.

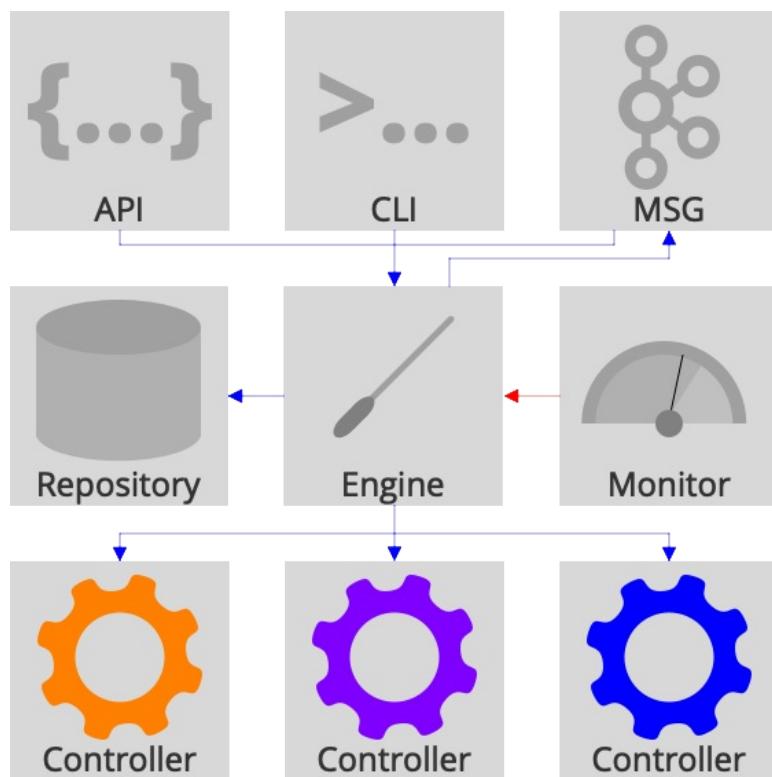
In addition it differentiates between sections for information obtained in the sourcing phase ([Catalog](#)), the design/development phase ([Architecture](#)) and the operations phase ([Solution](#)).

The information related to the automation of the lifecycle of the [solution](#) elements is persisted in the [Automation](#) section.

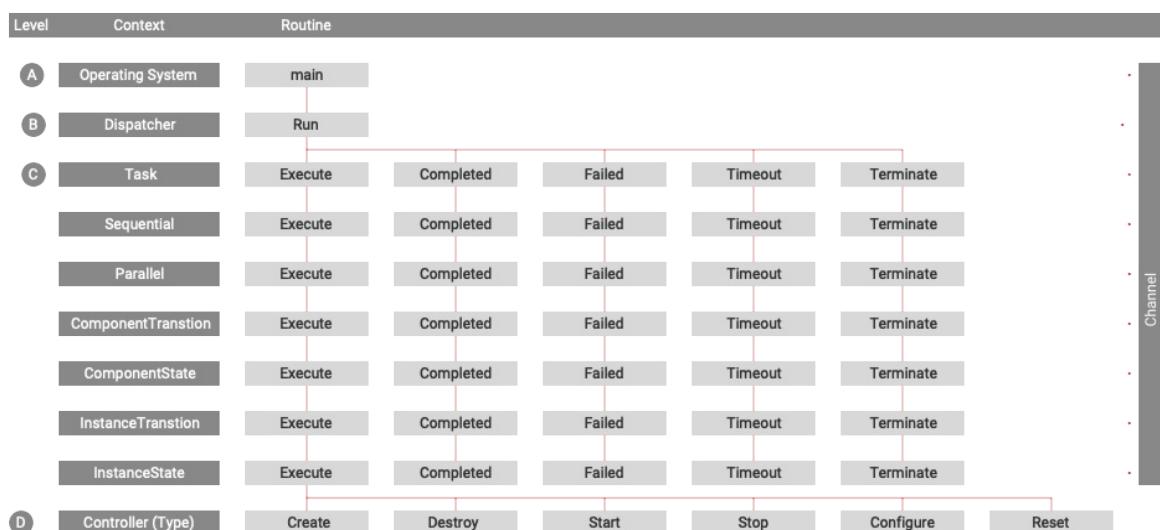
Engine

The [orchestrator engine](#) is responsible for coordinating the lifecycle management activities of the various [component](#) specific controllers.

The [engine](#) is a core module of the [orchestrator](#). It exchanges commands and information via the northbound interfaces, receives reconciliation requests from the monitoring system, parametrises its procedures with the help of descriptors maintained in the [repository](#) and orchestrates the lifecycle management procedures of the southbound controllers.



The following diagram provides an overview of the algorithm of the orchestration engine:



The algorithm splits up into several distinct layers:

Layer A: Main Routine

The main routine:

- registers all controllers,
- starts the internal communication channel,
- starts the timeout monitoring system,
- starts the monitoring system,
- starts the [task](#) dispatcher and finally
- initiates all external interfaces (application programming interface, command line interface and message bus interface).

The interfaces collect the requests coming in from DevOps or external systems and publish these requests to the internal channel.

Layer B: [Task](#) Dispatcher

The [task](#) dispatcher listens to requests published to the internal channel these requests are evaluated in order to identify what kind of [task](#) needs to be executed. The [task](#) dispatcher then spawns an appropriate subprocess and waits for the next request.

Layer C: [Task](#) Execution

The [task](#) to be executed depends on type of the [task](#) context and the state of the [task](#). The context of a [task](#) can be:

- [Architecture](#): the instantiation/update of a [solution](#) based on architectural blueprint,
- [Cluster](#): the resizing, status update or reconfiguration of a [cluster](#),
- [Instance](#): the status update or reconfiguration of an [instance](#).

The [task](#) states are related to the influencing events:

- Execution: indicates that a [task](#) should be executed/reexecuted
- Completed: indicates that a [task](#) has completed successfully and that all finalising activities need to be conducted
- Failed: indicates that a [task](#) has failed and that all required cleanup activities need to be conducted
- Timeout: indicates that a [task](#) has run into a timeout and that all required recovery or cleanup activities need to be conducted
- Terminate: indicates that a [task](#) needs to be terminated

The [component](#) type specific controllers are invoked synchronously when executing tasks in the context of instances.

The [task](#) may trigger subtasks depending on the information provided in the [catalog](#) and [architecture](#) section of the [model](#) by issuing corresponding requests.

Layer D: Controller

Controllers are capable of transitioning instances of a specific [component](#) type along the paths defined in the common [lifecycle model](#).

This includes the capabilities to:

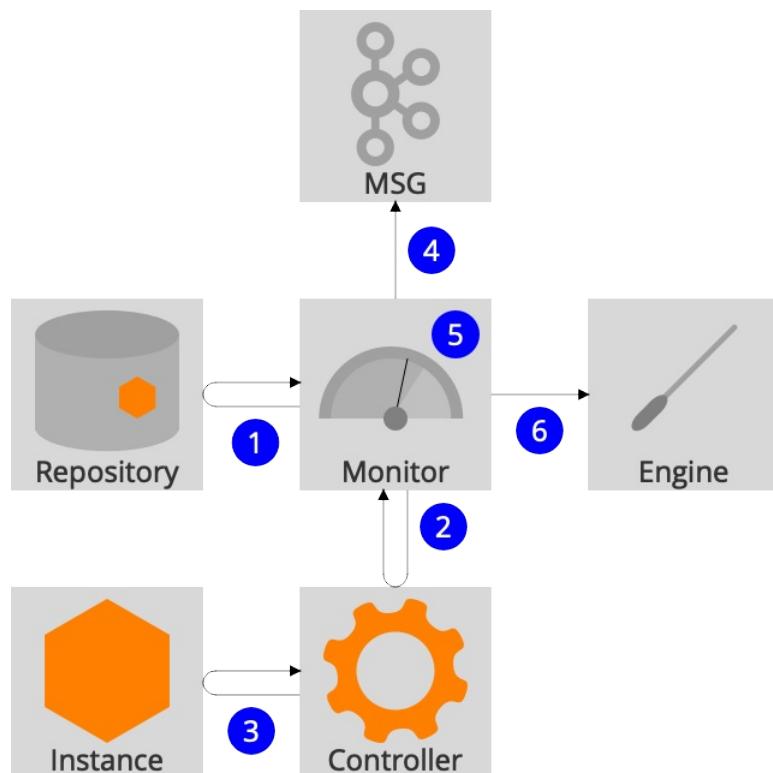
- Create: create an [instance](#),
- Destroy: destroy an [instance](#),
- Start: start an [instance](#),
- Stop: stop an [instance](#),
- Configure: configure or reconfigure an [instance](#) and

- Reset: cleanup an [instance](#).

Monitor

The monitoring system continuously compares the current state of instances with the desired target state and if needed triggers compensating measures.

The following diagram depicts the basic functionality provided by the monitoring system.



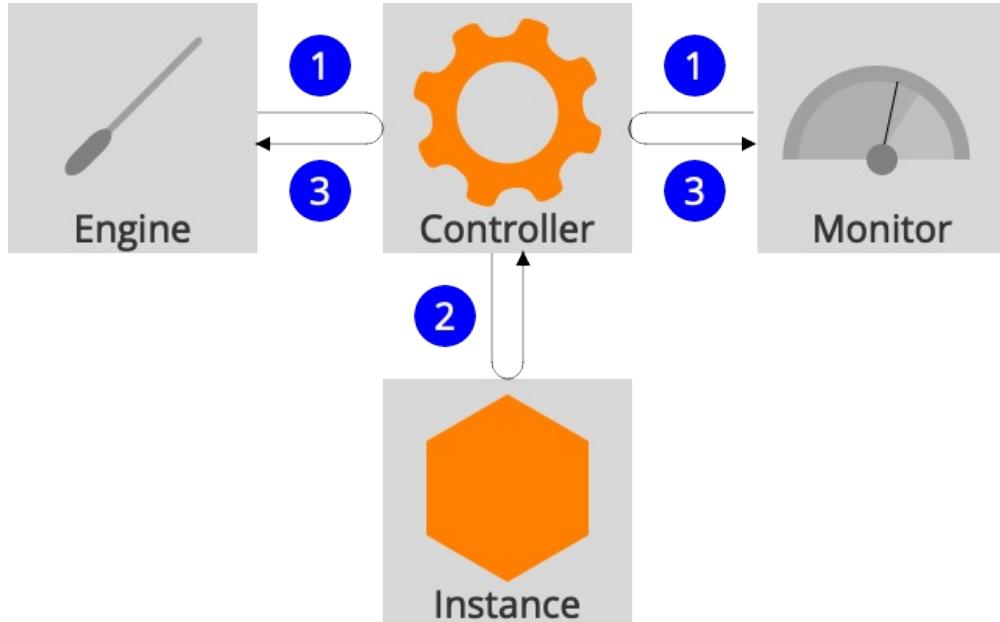
The monitoring system will conduct in regular intervals the following steps:

1. acquire from the **model** database all the instances of **solution** elements which need to be monitored
2. trigger the corresponding controllers to obtain the status of an **instance** of the **solution** elements
3. the controllers evaluate the status of the instances of **solution** elements
4. the monitoring system notifies external systems via message bus about the status of an **instance** of the **solution** elements
5. the monitoring system checks if the desired state matches the current state
6. the monitoring triggers if required a compensating measure to align the current state of an **instance** of a **solution element** with the desired state

Controller

A **controller** is a **component** type specific management **component** capable of applying the changes to **component** instances according to the standardised **component lifecycle model** and providing information about the status of these instances.

The diagram shows how the **orchestrator engine** and the **monitor** system interact with the **controller**:



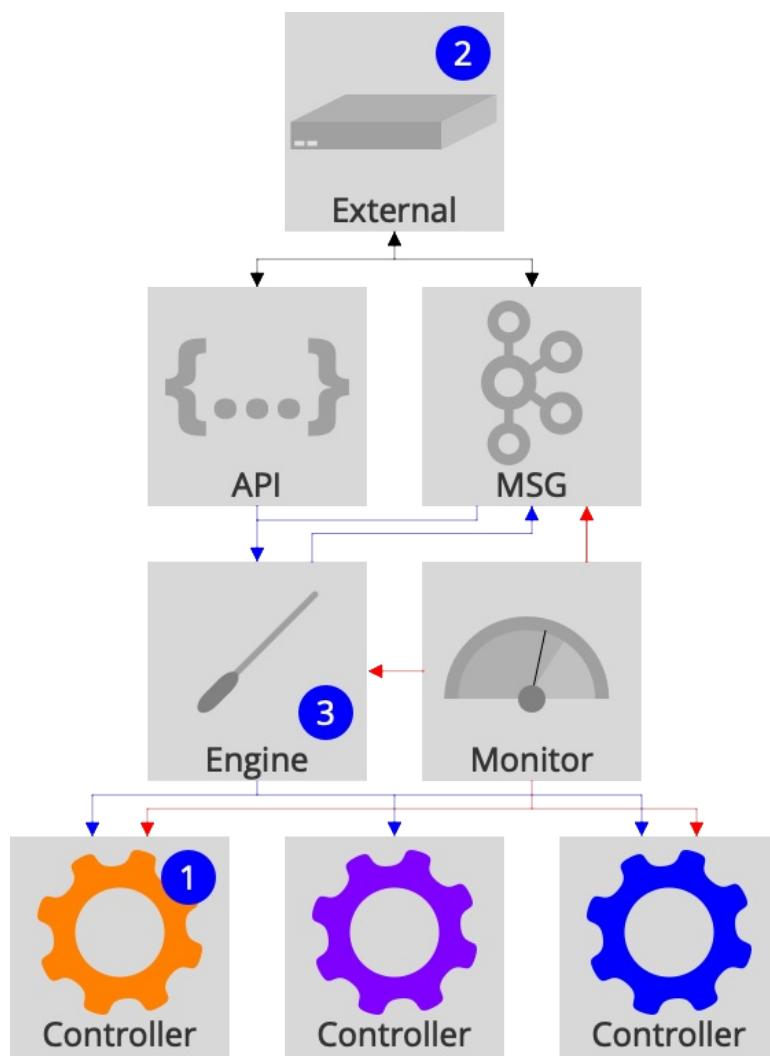
1. The **controller** can either receive lifecycle management instructions from the orchestration **engine** or be triggered by the monitoring system to determine the status of instances of **solution** elements.
2. The **controller** then executes these requests and
3. delivers the results to the requesting module of the **orchestrator**.

The interface specification of the **controller** is listed in the appendix [Controller API](#).

Extensions

Enhancing closed loop automation in a well defined way with custom code.

Managing the lifecycle of a complex [solution](#) may have to include sophisticated procedures which can not be covered by a closed-loop automation framework in a generic manner.



Therefore an extension mechanism is required to include these procedures in a well defined way. Following options could be made available:

1. **Controller**: writing a custom [controller](#) for a specific [component](#) type is the most obvious way to include custom code into the orchestration framework
2. **Hook**: custom code could subscribe to monitoring information published by the monitoring system via the message bus interface and trigger actions which could include interactions with the [orchestrator](#) via its API. The same applies to the [task](#) related events issued by the [orchestrator engine](#) via the message bus interface.
3. **Component Specific Task Handlers**: the [orchestrator](#) code could invoke [component](#) specific [task](#) handlers which would overwrite the generic handlers defined in the core orchestration [engine](#).

Glossary

Architecture

An [architecture](#) is a versioned description of the desired [configuration](#) of a [solution](#).

Catalog

The [catalog](#) is a subset of the information [model](#) of a [domain](#) which holds templates (components and their dependencies) for [solution](#) elements along with their basic [configuration](#).

Cluster

A [Cluster](#) describes the runtime [configuration](#) for a [cluster](#) within a [solution element](#) for a specific version of a [component](#) type.

Component

A [component](#) is a versioned template for a [solution element](#). It describes its basic [configuration](#) with the help of a structured object and a set of named dependencies to other components

Configuration

A [configuration](#) describes how to setup a [Solution](#), [Element](#), [Cluster](#), [Relationship](#) or [Instance](#). A basic [configuration](#) is provided by the defintion of a [Component](#) in the [catalog](#). Architectures capture design time configurations in the entities: [ElementConfiguration](#), [ClusterConfiguration](#) and [RelationshipConfiguration](#). The actual run-time [configuration](#) is held in the entities themselves.

Controller

A [component](#) type specific management [component](#) capable of applying the changes to [component](#) instances according to the standardised [component](#) life cycle [model](#).

Orchestrator

An automation tool which coordinates the versions, configurations and lifecycle states of a set of interdependent components so that a desired target version, [configuration](#) and lifecycle state of these components is achieved.

Depedency

A dependency describes how the containing [component](#) relates to other components. It can either be a "context" dependency which defines that the referenced [component](#) needs to be "active" before the [component](#) can be created, or it may be a "service" dependency which states that the referenced [component](#) needs to be "active" before the

[component](#) can be started.

Domain

A [domain](#) is an administrative realm which can be managed independently from other domains

Element

An [Element](#) describes the runtime [configuration](#) of a [solution element](#) based on a [component](#) type. A [solution element](#) may need to comprise several clusters which resemble versions of a [component](#).

Engine

The [orchestrator engine](#) is responsible for coordinating the lifecycle management activities of the various [component](#) specific controllers.

Event

An [event](#) notifies about the occurrence of a [task](#) related [event](#).

Extensions

[Extensions](#) enhance the functionality of the [orchestrator](#) by augmenting it with custom code.

Hook

A [hook](#) subscribes to lifecycle events and status information coming from the message bus interface and triggers custom code.

Instance

An [Instance](#) describes an [instance](#) of a [cluster](#) of a [solution element](#).

Lifecycle Model

A [lifecycle model](#) can be represented as a directed graph of states and transitions describing how an [instance](#) of a [component](#) may evolve in the course of its existence.

Model

The information required for the closed-loop automation procedures for managing the lifecycle of solutions.

Monitor

The monitoring system is a module of the orchestrator and continuously compares the current state of instances with the desired target state and if needed triggers compensating measures.

Relationship

A [Relationship](#) describes the runtime [configuration](#) for a dependency between clusters of related [solution](#) elements.

Repository

The [repository](#) is a module of the [orchestrator](#) and holds the information required for the closed-loop automation procedures.

Solution

A [solution](#) entity describes the current [configuration](#) and state of a [solution](#) which may consist of a number of elements.

Task

A [task](#) describes an activity to manipulate a [solution](#) or parts of a [solution](#).

Appendix

- [Orchestrator Application Programming Interface](#)
- [Orchestrator Command Line Interface](#)
- [Orchestrator Message Bus Interface](#)
- [Controller Application Programming Interface](#)

Orchestrator API

Orchestrator

Version: 0.0.1

[Orchestrator API \(mailto:bernard@tsai.eu\)](#)

Schemes: http

Summary

Tag: orchestration

Orchestration

Operation	Description
-----------	-------------

Tag: default

Operation	Description
POST /	Store model
GET /	Retrieve model
DELETE /	Reset model
GET /domain	List domains
POST /domain/{d}	Store domain
GET /domain/{d}	Retrieve domain
DELETE /domain/{d}	Remove domain
GET /domain/{d}/catalog	List components
GET /domain/{d}/catalog/{c}	List versions of a component template
POST /domain/{d}/catalog/{c}/version/{v}	Create component
GET /domain/{d}/catalog/{c}/version/{v}	Retrieve component
DELETE /domain/{d}/catalog/{c}/version/{v}	Remove component
GET /domain/{d}/architecture	List architectures
GET /domain/{d}/architecture/{a}	List architecture versions

<code>POST /domain/{d}/architecture/{a}/version/{v}</code>	Store architecture
<code>GET /domain/{d}/architecture/{a}/version/{v}</code>	Retrieve architecture
<code>DELETE /domain/{d}/architecture/{a}/version/{v}</code>	Remove architecture
<code>GET /domain/{d}/solution</code>	List solutions
<code>POST /domain/{d}/solution/{s}</code>	Store solution
<code>GET /domain/{d}/solution/{s}</code>	Retrieve solution
<code>DELETE /domain/{d}/solution/{s}</code>	Remove solution
<code>PUT /domain/{d}/solution/{s}/version/{v}</code>	Modify solution
<code>GET /domain/{d}/solution/{s}/element</code>	List elements
<code>GET /domain/{d}/solution/{s}/element/{e}</code>	Retrieve element
<code>PUT /domain/{d}/solution/{s}/element/{e}</code>	Modify element
<code>GET /domain/{d}/solution/{s}/element/{e}/cluster</code>	List clusters
<code>GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}</code>	Retrieve cluster
<code>PUT /domain/{d}/solution/{s}/element/{e}/cluster/{c}</code>	Modify cluster
<code>GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instance</code>	List instances
<code>GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instance/{e}</code>	Retrieve instance
<code>PUT /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instance/{e}</code>	Modify instance
<code>GET /domain/{d}/task</code>	List tasks
<code>GET /domain/{d}/task/{t}</code>	Retrieve task
<code>PUT /domain/{d}/task/{t}</code>	Terminate task
<code>GET /domain/{d}/event</code>	List events
<code>GET /domain/{d}/event/{e}</code>	Retrieve event

Paths

[Reset model](#)

`DELETE /`

reset `model` in the `repository` to initial state

200 OK

success

Retrieve `model`

GET /

retrieve `model` from the `repository` as yaml data

application/x-yaml

200 OK

`model` data as yaml

`Model`

Store `model`

POST /

create `model` from yaml data and store it into the `repository`

application/x-yaml

`model` data as yaml

`Model`

200 OK

Success

List domains

GET /domain

Retrieves list of `domain` names

application/x-yaml

200 OK

`domain` names as yaml

`Names`

Remove `domain`

DELETE /domain/{d}

remove `domain` from the `repository`

Name	Description	Type	Data Type	Annotation
d	<code>domain</code> name	path	string	

application/x-yaml

200 OK

Success

404 Not Found

domain not found

Retrieve domain

GET /domain/{d}

retrieve domain from the repository as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	

application/x-yaml

200 OK

domain data as yaml

Domain

404 Not Found

domain not found

Store domain

POST /domain/{d}

create a new domain from yaml data and store it in the repository

application/x-yaml

domain data as yaml

Domain

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	

200 OK

Success

List architectures

GET /domain/{d}/architecture

list all the available architectures in a domain

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	

application/x-yaml

200 OK

architecture names as yaml

Names

List [architecture](#) versions

GET /domain/{d}/architecture/{a}

list all the available versions of an [architecture](#) in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
a	architecture name	path	string	

application/x-yaml

200 OK

version names as yaml

[Names](#)

Remove [architecture](#)

DELETE /domain/{d}/architecture/{a}/version/{v}

remove [architecture](#) from the [repository](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
a	architecture name	path	string	
v	version of architecture	path	string	

200 OK

success

Retrieve [architecture](#)

GET /domain/{d}/architecture/{a}/version/{v}

retrieve [architecture](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
a	architecture name	path	string	
v	version of architecture	path	string	

application/x-yaml

200 OK

[architecture](#) data as yaml

[Architecture](#)

Store [architecture](#)

POST /domain/{d}/architecture/{a}/version/{v}

create a new [architecture](#) from yaml data and store it in the [repository](#)

application/x-yaml

[architecture](#) data as yaml

Architecture

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
a	architecture name	path	string	
v	version of architecture	path	string	

200 OK

Success

List components

GET /domain/{d}/catalog

list all the available components in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	

application/x-yaml

200 OK

[component](#) names as yaml

Names

List versions of a [component](#) template

GET /domain/{d}/catalog/{c}

list all the available versions of a [component](#) in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
c	component name	path	string	

application/x-yaml

200 OK

version names as yaml

Names

Remove [component](#)

DELETE /domain/{d}/catalog/{c}/version/{v}

remove [component](#) from the [repository](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
c	component name	path	string	
v	version of component	path	string	

200 OK

success

Retrieve [component](#)

GET /domain/{d}/catalog/{c}/version/{v}

retrieve [component](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
c	component name	path	string	
v	version of component	path	string	

application/x-yaml

200 OK

[component](#) data as yaml

Component

Create [component](#)

POST /domain/{d}/catalog/{c}/version/{v}

create a new [component](#) from yaml data and store it in the [repository](#)

application/x-yaml

[component](#) data as yaml

Component

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
c	component name	path	string	
v	version of component	path	string	

200 OK

Success

List events

GET /domain/{d}/event

list all events related to a [task](#) in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
t	task name	query	string	

application/x-yaml

200 OK

[event](#) names as yaml

Names

Retrieve [event](#)

GET /domain/{d}/event/{e}

retrieve [event](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
e	event name	path	string	

application/x-yaml

200 OK

[event](#) data as yaml

Event

List solutions

GET /domain/{d}/solution

list all the available solutions in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	

application/x-yaml

200 OK

[solution](#) names as yaml

Names

Remove [solution](#)

DELETE /domain/{d}/solution/{s}

remove [solution](#) from the [repository](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	

200 OK

success

Retrieve [solution](#)

GET /domain/{d}/solution/{s}

retrieve [solution](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	

application/x-yaml

200 OK

[solution](#) data as yaml

[Solution](#)

Store [solution](#)

POST /domain/{d}/solution/{s}

create a new [solution](#) from yaml data and store it in the [repository](#)

application/x-yaml

[solution](#) data as yaml

[Solution](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	

200 OK

Success

List elements

GET /domain/{d}/solution/{s}/element

list all the available elements in a [solution](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	

application/x-yaml

200 OK

[element](#) names as yaml

[Names](#)

Retrieve [element](#)

GET /domain/{d}/solution/{s}/element/{e}

retrieve [element](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	

application/x-yaml

200 OK

[element](#) data as yaml

[Element](#)

Modify [element](#)

PUT /domain/{d}/solution/{s}/element/{e}

update an existing [element](#) based on the information provided in a [configuration](#) object

application/x-yaml

[configuration](#) data as yaml

[Configuration](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	

200 OK

Success

List clusters

GET /domain/{d}/solution/{s}/element/{e}/clusterlist all the available clusters in an [element](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	

application/x-yaml

200 OK

[cluster](#) names as yaml[Names](#)Retrieve [cluster](#)**GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}**retrieve [cluster](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	
c	cluster name	path	string	

application/x-yaml

200 OK

[cluster](#) data as yaml[Cluster](#)Modify [cluster](#)**PUT /domain/{d}/solution/{s}/element/{e}/cluster/{c}**update an existing [cluster](#) based on the information provided in a [configuration](#) object

application/x-yaml

[configuration](#) data as yaml

Configuration

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	
c	cluster name	path	string	

200 OK

Success

List instances

GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instancelist all the available instances in a [cluster](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	
c	cluster name	path	string	

application/x-yaml

200 OK

[instance](#) names as yaml[Names](#)Retrieve [instance](#)**GET /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instance/{e}**retrieve [instance](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	
c	cluster name	path	string	
i	instance name	path	string	

application/x-yaml

200 OK

[instance](#) data as yaml

[Instance](#)

Modify [instance](#)

PUT /domain/{d}/solution/{s}/element/{e}/cluster/{c}/instance/{e}

update an existing [instance](#) based on the information provided in a [configuration](#) object

application/x-yaml

[configuration](#) data as yaml

[Configuration](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
e	element name	path	string	
c	cluster name	path	string	
i	instance name	path	string	

200 OK

Success

Modify [solution](#)

PUT /domain/{d}/solution/{s}/version/{v}

create a new [solution](#) or update an existing [solution](#) based on the information provided in an [architecture](#) blueprint

application/x-yaml

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
s	solution name	path	string	
v	version of the solution	path	string	

200 OK

Success

List tasks

GET /domain/{d}/task

list all toplevel tasks in a [domain](#) (optionally matching a certain request ID)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
r	request name	query	string	

application/x-yaml

200 OK

[task](#) names as yaml

[Names](#)

Retrieve [task](#)

GET /domain/{d}/task/{t}

retrieve [task](#) from the [repository](#) as yaml data

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
t	task name	path	string	

application/x-yaml

200 OK

[task](#) data as yaml

[Task](#)

Terminate [task](#)

PUT /domain/{d}/task/{t}

stop a [task](#) in a [domain](#)

Name	Description	Type	Data Type	Annotation
d	domain name	path	string	
t	task name	path	string	

200 OK

Success

Schema definitions

Architecture: object

An [configuration](#) for a version of a [solution](#)

Name: string

Name of the solution architecture

Version: string

Version of the solution architecture

Configuration: Configuration

Elements: object

Map of component names to configurations for solution elements

ElementConfiguration

Cluster: object

A cluster of instances of an element

Version: string

Version of the instance

Configuration: Configuration

Endpoint: Endpoint

Relationships: object

Map of dependency names to dependencies

Relationship

Instances: object

Map of uuids to cluster element instances

Instance

ClusterConfiguration: object

A configuration for a cluster of solution elements

Version: string

Version of a component template

State: string

Desired state of the version of a solution element

Size: integer , { $x \in \mathbb{Z} | x \geq 0$ }

Desired size of the version of a solution element

Configuration: Configuration

Dependencies: object

Map of dependency names to dependency configurations

RelationshipConfiguration

Component: object

A template for a version of a component

Component: string

Name of the component

Version: string

Version of the [component](#)

Configuration: Configuration

Dependencies: object

Map of names to dependencies of a [component](#) version

[Dependency](#)

Configuration: object

[Configuration](#) information as structured object

object

Dependency: object

A template for a [component](#) dependency

Name: string

Name of the dependency

Type: string , x ∈ { context , service }

Type of dependency (context/service)

Component: string

Name of the referenced [component](#)

Version: string

Version of the referenced [component](#)

Configuration: Configuration

Domain: object

An administrative realm within a [model](#)

Name: string

Name of the [domain](#)

Catalog: object

Map of names to templates for elements

[Component](#)

Architectures: object

Map of names to [architecture](#) blueprints

[Architecture](#)

Solutions: object

Map of names to components

[Solution](#)

Task: object

Map of uuids to tasks

[Task](#)

Event: object

Map of uuids to events

Event**Element: object**

An [element](#) of a [solution](#)

Name: string

Name of the [element](#)

Component: string

Name of the [component](#)

Configuration: Configuration***Endpoint: Endpoint******Clusters: object***

Map of version names to clusters

Cluster**ElementConfiguration: object**

A [configuration](#) for a [solution element](#)

Name: string

Name of the [solution element](#)

Component: string

Name of the [component](#) template

Configuration: Configuration***Versions: object***

Map of version names to configurations for versions of [solution](#) elements

ClusterConfiguration**Endpoint: object**

Endpoint information as structured object

object

Event: object

An [event](#) related to a [task](#)

Domain: string

Name of the [domain](#)

UUID: string

Univesal unique identifier of the [event](#)

Task: string

Univesal unique identifier of the [task](#)

Type: string , $x \in \{ \text{execution} , \text{completion} , \text{failure} , \text{timeout} \}$

Type of the event

Source: string

Univesal unique identifier of the issuing task or "

Time: string

time since 1.1.1970 in nsecs

Instance: object

An instance of an element

UUID: string

Unique universal identifier of the instance

State: string

State of the instance

Configuration: Configuration

Endpoint: Endpoint

Model: object

The information model for the orchestration of lifecycles

Schema: string

Schema of the model

Name: string

Name of the model

Domains: object

Map of names to domains

Domain

Names: string[]

List of names

string

Relationship: object

An instance of an element

Name: string

Name of the dependency

Configuration: Configuration

Endpoint: Endpoint

RelationshipConfiguration: object

A configuration for a dependency of a version of a solution element

Name: string

Name of the dependency

Configuration: Configuration

Solution: object

A solution consisting of elements

Name: string

Name of the solution

Configuration: Configuration

Elements: object

Map of element names to elements

Element

Task: object

An administrative realm within a model

Type: string

Type of the task

Domain: string

Name of the Domain

Solution: string

Name of the Solution

Element: string

Name of the element

Cluster: string

Cluster (version) of the element

Instance: string

Universal unique identifier of the instance

State: string

Desired state for the entity

UUID: string

Universal unique identifier of the task

Parent: string

Universal unique identifier of the parent task

Status: string , $x \in \{ \text{execution} , \text{completion} , \text{failure} , \text{timeout} \}$

Execution status of the task

Phase: string

Execution phase of the task

Subtasks: string[]

Unique universal identifiers of the subtasks

string

Command Line Interface Version: 0.0.1

Documentation of the command line interface of the [orchestrator](#).

Store model.

create [model](#) from yaml data and store it into the [repository](#)

Usage:

```
> StoreModel -model [model data as yaml]
```

Parameters:

-[model](#) [[model](#) data as yaml] (required)

Response:

Retrieve model.

retrieve [model](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveModel
```

Response:

[Model](#): [model](#) data as yaml

Reset model.

reset [model](#) in the [repository](#) to initial state

Usage:

```
> ResetModel
```

Response:

List domains.

Retrieves list of [domain](#) names

Usage:

```
> ListDomains
```

Response:

Names: [domain](#) names as yaml

Store domain.

create a new [domain](#) from yaml data and store it in the [repository](#)

Usage:

```
> StoreDomain -d [domain name] -domain [domain data as yaml]
```

Parameters:

-d [\[domain name\]](#) (required)
-domain [\[domain data as yaml\]](#) (required)

Response:

Retrieve domain.

retrieve [domain](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveDomain -d [domain name]
```

Parameters:

-d [\[domain name\]](#) (required)

Response:

Domain: [domain](#) data as yaml

Remove domain.

remove [domain](#) from the [repository](#)

Usage:

```
> RemoveDomain -d [domain name]
```

Parameters:

-d [\[domain name\]](#) (required)

Response:

List components.

list all the available components in a [domain](#)

Usage:

```
> ListComponents -d [domain name]
```

Parameters:

-d [\[domain name\]](#) (required)

Response:

Names: [component](#) names as yaml

List versions of a component template.

list all the available versions of a [component](#) in a [domain](#)

Usage:

```
> ListComponentVersions -d [domain name] -c [component name]
```

Parameters:

-d [\[domain name\]](#) (required)

-c [\[component name\]](#) (required)

Response:

Names: version names as yaml

Create component.

create a new [component](#) from yaml data and store it in the [repository](#)

Usage:

```
> CreateComponent -d [domain name] -c [component name] -v [version of component] -component [component data as yaml]
```

Parameters:

-d [\[domain name\]](#) (required)

-c [\[component name\]](#) (required)

-v [\[version of component\]](#) (required)

-component [\[component data as yaml\]](#) (required)

Response:

Retrieve component.

retrieve [component](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveComponent -d [domain name] -c [component name] -v [version of component]
```

Parameters:

- d [\[domain name\]](#) (required)
- c [\[component name\]](#) (required)
- v [\[version of component\]](#) (required)

Response:

[Component](#): component data as yaml

Remove component.

remove [component](#) from the [repository](#)

Usage:

```
> RemoveComponent -d [domain name] -c [component name] -v [version of component]
```

Parameters:

- d [\[domain name\]](#) (required)
- c [\[component name\]](#) (required)
- v [\[version of component\]](#) (required)

Response:

List architectures.

list all the available architectures in a [domain](#)

Usage:

```
> ListArchitectures -d [domain name]
```

Parameters:

- d [\[domain name\]](#) (required)

Response:

Names: [architecture](#) names as yaml

List architecture versions.

list all the available versions of an [architecture](#) in a [domain](#)

Usage:

```
> ListArchitectureVersions -d [domain name] -a [architecture name]
```

Parameters:

-d [[domain](#) name] (required)

-a [[architecture](#) name] (required)

Response:

Names: version names as yaml

Store architecture.

create a new [architecture](#) from yaml data and store it in the [repository](#)

Usage:

```
> StoreArchitecture -d [domain name] -a [architecture name] -v [version of architecture] -architecture [architecture data as yaml]
```

Parameters:

-d [[domain](#) name] (required)

-a [[architecture](#) name] (required)

-v [version of [architecture](#)] (required)

-architecture [[architecture](#) data as yaml] (required)

Response:

Retrieve architecture.

retrieve [architecture](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveArchitecture -d [domain name] -a [architecture name] -v [version of architecture]
```

Parameters:

- d [domain name] (required)
- a [architecture name] (required)
- v [version of architecture] (required)

Response:

Architecture: architecture data as yaml

Remove architecture.

remove architecture from the repository

Usage:

```
> RemoveArchitecture -d [domain name] -a [architecture name] -v [version of architecture]
```

Parameters:

- d [domain name] (required)
- a [architecture name] (required)
- v [version of architecture] (required)

Response:

List solutions.

list all the available solutions in a domain

Usage:

```
> ListSolutions -d [domain name]
```

Parameters:

- d [domain name] (required)

Response:

Names: solution names as yaml

Store solution.

create a new solution from yaml data and store it in the repository

Usage:

```
> StoreSolution -d [domain name] -s [solution name] -solution [solution data as yaml]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)
- solution [solution data as yaml] (required)

Response:

Retrieve solution.

retrieve [solution](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveSolution -d [domain name] -s [solution name]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)

Response:

[Solution](#): [solution](#) data as yaml

Remove solution.

remove [solution](#) from the [repository](#)

Usage:

```
> RemoveSolution -d [domain name] -s [solution name]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)

Response:

Modify solution.

create a new [solution](#) or update an existing [solution](#) based on the information provided in an [architecture](#) blueprint

Usage:

```
> ModifySolution -d [domain name] -s [solution name] -v [version of the solution]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)
- v [version of the solution] (required)

Response:

List elements.

list all the available elements in a [solution](#)

Usage:

```
> ListElements -d [domain name] -s [solution name]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)

Response:

Names: [element](#) names as yaml

Retrieve element.

retrieve [element](#) from the repository as yaml data

Usage:

```
> RetrieveElement -d [domain name] -s [solution name] -e [element name]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)
- e [element name] (required)

Response:

Element: [element](#) data as yaml

Modify element.

update an existing [element](#) based on the information provided in a [configuration](#) object

Usage:

```
> ModifyElement -d [domain name] -s [solution name] -e [element name] -configuration [configuration data as yaml]
```

Parameters:

- d [\[domain name\]](#) (required)
- s [\[solution name\]](#) (required)
- e [\[element name\]](#) (required)
- configuration [\[configuration data as yaml\]](#) (required)

Response:

List clusters.

list all the available clusters in an [element](#)

Usage:

```
> ListClusters -d [domain name] -s [solution name] -e [element name]
```

Parameters:

- d [\[domain name\]](#) (required)
- s [\[solution name\]](#) (required)
- e [\[element name\]](#) (required)

Response:

Names: [cluster](#) names as yaml

Retrieve cluster.

retrieve [cluster](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveCluster -d [domain name] -s [solution name] -e [element name] -c [cluster name]
```

Parameters:

- d [\[domain name\]](#) (required)
- s [\[solution name\]](#) (required)
- e [\[element name\]](#) (required)
- c [\[cluster name\]](#) (required)

Response:

Cluster: [cluster](#) data as yaml

Modify cluster.

update an existing [cluster](#) based on the information provided in a [configuration](#) object

Usage:

```
> ModifyCluster -d [domain name] -s [solution name] -e [element name] -c [cluster name] -configuration [configuration data as yaml]
```

Parameters:

- d [\[domain name\]](#) (required)
- s [\[solution name\]](#) (required)
- e [\[element name\]](#) (required)
- c [\[cluster name\]](#) (required)
- configuration [\[configuration data as yaml\]](#) (required)

Response:

List instances.

list all the available instances in a [cluster](#)

Usage:

```
> ListInstances -d [domain name] -s [solution name] -e [element name] -c [cluster name]
```

Parameters:

- d [\[domain name\]](#) (required)
- s [\[solution name\]](#) (required)
- e [\[element name\]](#) (required)
- c [\[cluster name\]](#) (required)

Response:

Names: [instance](#) names as yaml

Retrieve instance.

retrieve [instance](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveInstance -d [domain name] -s [solution name] -e [element name] -c [cluster name] -i [instance name]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)
- e [element name] (required)
- c [cluster name] (required)
- i [instance name] (required)

Response:

Instance: [instance](#) data as yaml

Modify instance.

update an existing [instance](#) based on the information provided in a [configuration](#) object

Usage:

```
> ModifyInstance -d [domain name] -s [solution name] -e [element name] -c [cluster name] -i [instance name] -configuration  
[configuration data as yaml]
```

Parameters:

- d [domain name] (required)
- s [solution name] (required)
- e [element name] (required)
- c [cluster name] (required)
- i [instance name] (required)
- configuration [[configuration](#) data as yaml] (required)

Response:

List tasks.

list all toplevel tasks in a [domain](#) (optionally matching a certain request ID)

Usage:

```
> ListTasks -d [domain name] -r [request name]
```

Parameters:

-d [domain name] (required)

-r [request name] (optional)

Response:

Names: [task](#) names as yaml

Retrieve task.

retrieve [task](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveTask -d [domain name] -t [task name]
```

Parameters:

-d [domain name] (required)

-t [task name] (required)

Response:

Task: [task](#) data as yaml

Terminate task.

stop a [task](#) in a [domain](#)

Usage:

```
> TerminateTask -d [domain name] -t [task name]
```

Parameters:

-d [domain name] (required)

-t [task name] (required)

Response:

List events.

list all events related to a [task](#) in a [domain](#)

Usage:

```
> ListEvents -d [domain name] -t [task name]
```

Parameters:

-d [domain name] (required)

-t [task name] (required)

Response:

Names: [event](#) names as yaml

Retrieve event.

retrieve [event](#) from the [repository](#) as yaml data

Usage:

```
> RetrieveEvent -d [domain name] -e [event name]
```

Parameters:

-d [domain name] (required)

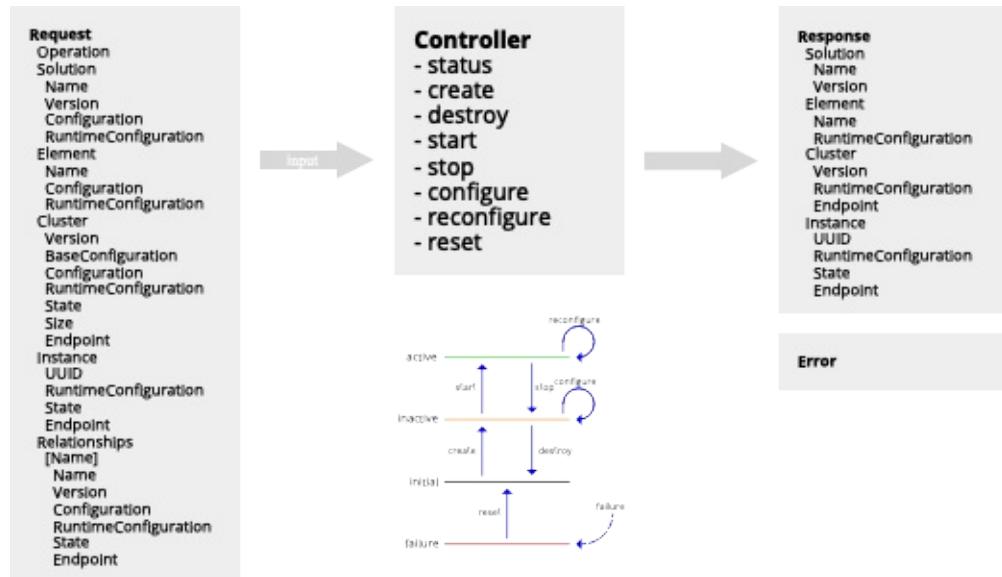
-e [event name] (required)

Response:

[Event](#): [event](#) data as yaml

Controller API

The methods which a [controller](#) needs to expose are described in the following sections. They aim to support the standard lifecycle management concept for instances and allow for monitoring their state as well.



Methods

Create

Creates a new [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = create(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Destroy

Destroys an existing [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = destroy(request)
```

The variables are:

Variable	Type	Description

request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Start

Activates an [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = start(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Stop

Deactivates an [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = stop(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Configure

Configures a deactivated [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = configure(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Reconfigure

Reconfigures an active [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = reconfigure(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Reset

Resets an [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = reset(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Status

Determines the status of an [instance](#) of a [cluster](#) of an [element](#) of a [solution](#).

```
response, error = status(request)
```

The variables are:

Variable	Type	Description
request	Request	Request as issued by the orchestrator
response	Response	Response returned by the controller
error	Error	Error object if an error has occurred

Objects

Request

A request object is passed as a serialised yaml object.

▼ Schema Definition:

```

Request:
  type: "object"
  description: |
    The request object as passed by the orchestration engine to the controller
    when requesting a lifecycle operation on a specific instance.
  properties:
    Operation:
      type: string
      description: "The requested lifecycle operation."
      enum: ["create", "destroy", "start", "stop", "configure", "reconfigure", "reset", "status"]
    Solution:
      type: object
      description: "Solution"
      properties:
        Name:
          type: string
          description: "Name of the solution"
        Version:
          type: string
          description: "Version of the solution"
        Configuration:
          type: object
          description: "Configuration of the solution"
          additionalProperties: true
        RuntimeConfiguration:
          type: object
          description: "Runtime configuration of the solution"
          additionalProperties: true
    Element:
      type: object
      description: "Element of the solution"
      properties:
        Name:
          type: string
          description: "Name of the element"
        Configuration:
          type: object
          description: "Configuration of the element"
          additionalProperties: true
        RuntimeConfiguration:
          type: object
          description: "Runtime configuration of the element"
          additionalProperties: true
    Cluster:
      type: object
      description: "Cluster of the element of the solution"
      properties:
        Version:
          type: string
          description: "Version of the element"
        BaseConfiguration:
          type: object
          description: "Base configuration of the cluster"
          additionalProperties: true
        Configuration:
          type: object
          description: "Configuration of the cluster"
          additionalProperties: true
        RuntimeConfiguration:
          type: object
          description: "Runtime configuration of the cluster"
          additionalProperties: true

```

```

Endpoint:
  type: object
  description: "Endpoint information of the cluster"
  additionalProperties: true
Instance:
  type: object
  description: "Instance of a cluster of the element of the solution"
  properties:
    UUID:
      type: string
      description: "Universal unique identifier of the instance"
RuntimeConfiguration:
  type: object
  description: "Runtime configuration of the instance"
  additionalProperties: true
Endpoint:
  type: object
  description: "Endpoint information of the instance"
  additionalProperties: true
Relationships:
  type: object
  description: "Instance of a cluster of the element of the solution"
  additionalProperties:
    $ref:
      type: object
      description: "Relationship"
      properties:
        Name:
          type: string
          description: "Name of the relationship"
Configuration:
  type: object
  description: "Configuration of the relationship"
  additionalProperties: true
RuntimeConfiguration:
  type: object
  description: "Runtime configuration of the relationship"
  additionalProperties: true
Endpoint:
  type: object
  description: "Endpoint information of the relationship"
  additionalProperties: true

```

Response

A response object is passed back as a serialised yaml object as a result from a lifecycle operation:

▼ Schema Definition:

```

Response:
  type: "object"
  description: |
    A response object is passed back as a serialised yaml object as a result from a lifecycle operation.
  properties:
    Solution:
      type: object
      description: "Solution"
      properties:
        Name:
          type: string
          description: "Name of the solution"
    Version:
      type: string
      description: "Version of the solution"
    Configuration:
      type: object
      description: "Configuration of the solution"

```

```

        additionalProperties: true
    RuntimeConfiguration:
        type: object
        description: "Runtime configuration of the solution"
        additionalProperties: true
Element:
    type: object
    description: "Element of the solution"
    properties:
        Name:
            type: string
            description: "Name of the element"
    Configuration:
        type: object
        description: "Configuration of the element"
        additionalProperties: true
    RuntimeConfiguration:
        type: object
        description: "Runtime configuration of the element"
        additionalProperties: true
Cluster:
    type: object
    description: "Cluster of the element of the solution"
    properties:
        Version:
            type: string
            description: "Version of the element"
        BaseConfiguration:
            type: object
            description: "Base configuration of the cluster"
            additionalProperties: true
        Configuration:
            type: object
            description: "Configuration of the cluster"
            additionalProperties: true
        RuntimeConfiguration:
            type: object
            description: "Runtime configuration of the cluster"
            additionalProperties: true
    Endpoint:
        type: object
        description: "Endpoint information of the cluster"
        additionalProperties: true
Instance:
    type: object
    description: "Instance of a cluster of the element of the solution"
    properties:
        UUID:
            type: string
            description: "Universal unique identifier of the instance"
    RuntimeConfiguration:
        type: object
        description: "Runtime configuration of the instance"
        additionalProperties: true
    Endpoint:
        type: object
        description: "Endpoint information of the instance"
        additionalProperties: true
Relationships:
    type: object
    description: "Instance of a cluster of the element of the solution"
    additionalProperties:
        $ref:
            type: object
            description: "Relationship"
            properties:
                Name:
                    type: string
                    description: "Name of the relationship"

```

```
Configuration:  
  type: object  
  description: "Configuration of the relationship"  
  additionalProperties: true  
RuntimeConfiguration:  
  type: object  
  description: "Runtime configuration of the relationship"  
  additionalProperties: true  
Endpoint:  
  type: object  
  description: "Endpoint information of the relationship"  
  additionalProperties: true
```

Error

An error object indicating possible issues is passed as a serialised yaml object according to following schema definition:

▼ Schema Definition:

```
Error:  
  type: "object"  
  description: |  
    An error indicating the issues which occurred in the context of a lifecycle operation.  
  properties:  
    additionalProperties: true
```