Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

DVOP1 - Cloud Native Application Composition

Prof. Dr. Thomas M. Bohnert
Christof Marti

Content

- Architecture Recap
- Decomposition
- Composition
  - Centralised v.s. Decentralised
  - Declarative v.s. Imperative
  - Standards & Defacto
- Technologies
  - docker-compose
  - Kubernetes Helm

Recap: Architecture

SOA Principles:

- Standardized protocols (e.g., SOAP, REST)
- Abstraction (from service implementation)
- **Loose coupling**
- Reusability
- **Composability**
- Stateless services
- Discoverable services

Microservices architecture is a SOA architectural style to develop applications

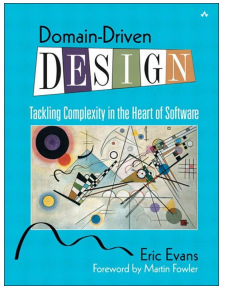
- as a **suite of "small" services**,
- each **running in its own process**
- and communicating with **lightweight** mechanisms (REST APIs or Messaging).
- They are **built around business capabilities** following the "do one thing well" principle.

Services are highly **decoupled** (yet ***composed***!) and focus on doing a small task.

Decomposition

- Key enabler of SOA and Microservice Architectures
- **Reduce** the application into a set of independent functional services
  - A service is a unit of software that is independently replaceable and upgradeable
  - A service encapsulates functionality and enforces an API
  - A service is independently deployable
  - A service lends itself to a continuous delivery software development process
  - Complex applications are **composed** of small, independent services (processes) and communicate with each other using APIs

Decomposition & Domain Driven Design (DDD)



Overview

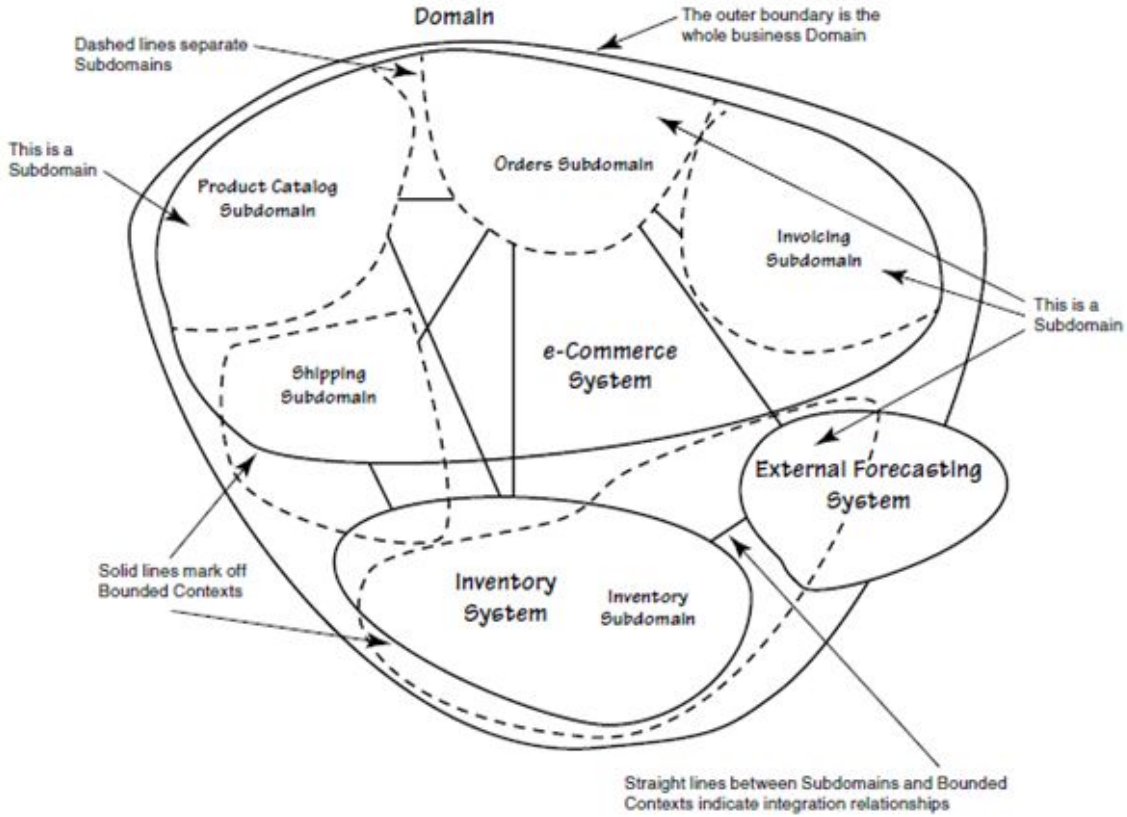
- A software development methodology used for complex systems
- Supports evolution of the design
- Primary focus of designs are domains
- Requires collaboration between domain (business) and technical expert

Core Concepts

- **Context**: the setting in which a concept appears determines its meaning
- **Domain**: specific area of knowledge or activity. Composed of **sub-domains**.
- **Model**: system of abstractions that describes selected aspects of a domain
- **Ubiquitous Language**: language structured around the domain model
- **Bounded Context**: Explicit definition of the context where a model applies (organisation, usage boundaries).

https://www.pearson.com/us/higher-education/program/Evans-Domain-Driven-Design-Tackling-Complexity-in-the-Heart-of-Software/PGM168436.html

Decomposition & Domain Driven Design (DDD)



How to Decompose: by sub-domain

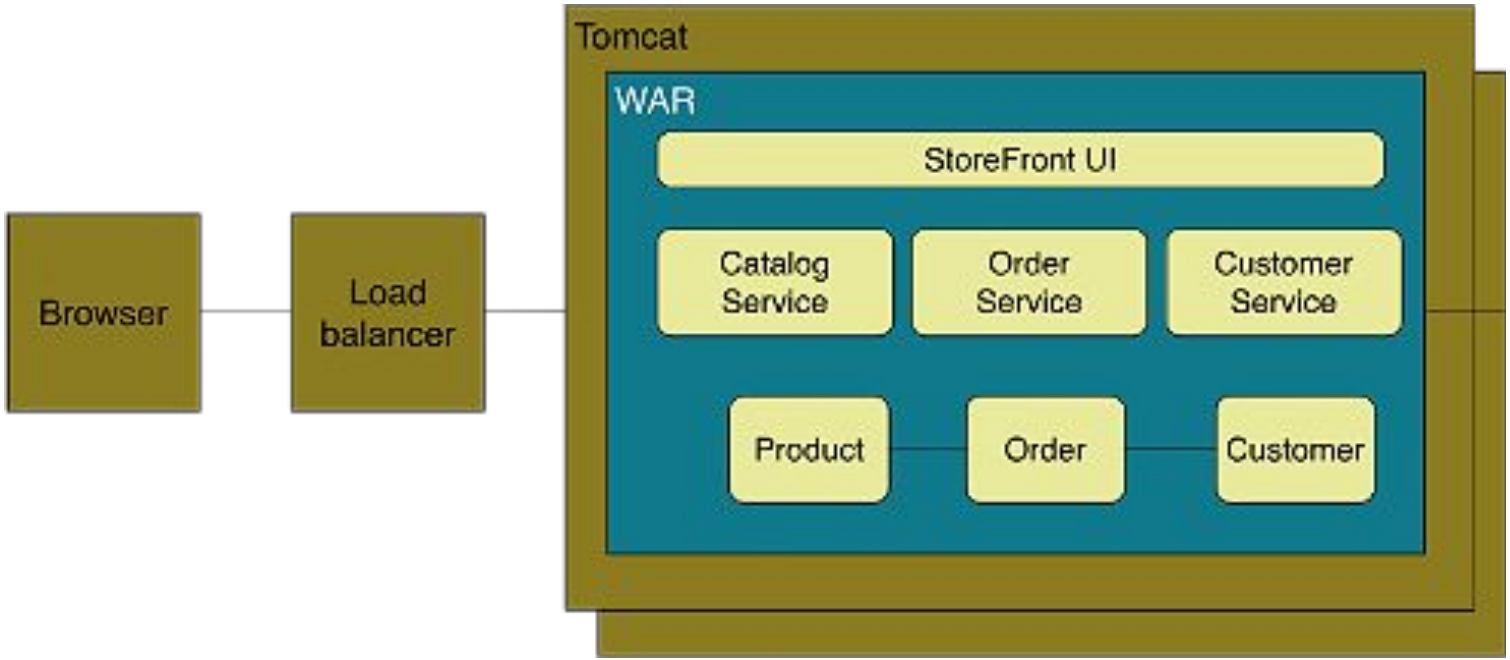
Define services corresponding to Domain-Driven Design (DDD) sub-domains

- Refers to the application's problem space - the business - as the domain
- A Domain consists of multiple sub-domains
  Each subdomain corresponds to a different part of the business. Subdomains can be classified as follows:
  - **Core** - key differentiator for the business and the most valuable part of the application
  - **Supporting** - related to what the business does but not a differentiator (implemented in-house or outsourced)
  - **Generic** - not specific to the business and are ideally implemented using off-the-shelf software

How to identify the sub-domains?

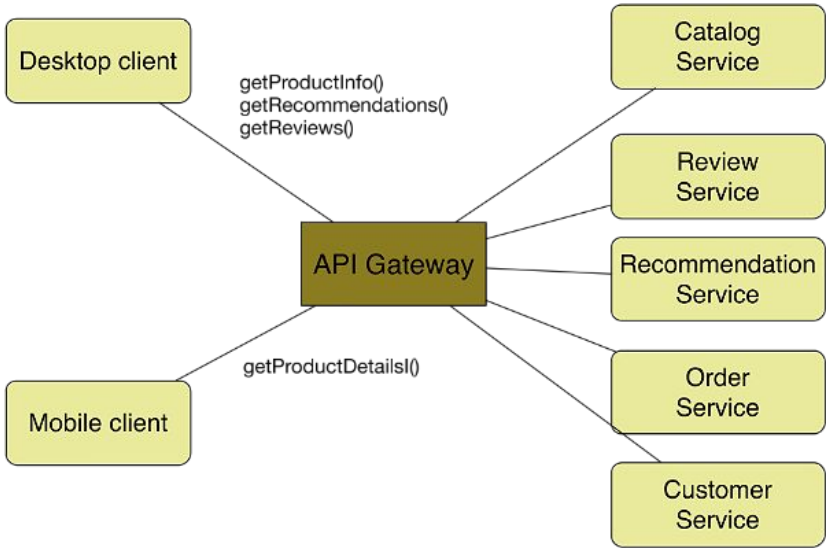
- Requires an understanding of the business.
- Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise using an iterative process.
- Good starting points for identifying subdomains are:
  - organization structure - different groups within an organization might correspond to subdomains
  - high-level domain model - subdomains often have a key domain object

Decomposing: An Example



Monolithic Application Design

Decomposing: An Example - decomposed



- How to assemble the application for deployment?

Application composition...

Application Composition

- Bringing these services together is known as composition.
- Composition's goal is to automatically bring together (deploy, provision) many components to deliver a functional system (e.g. replicated database system) or application (e.g. a 3-tier web application with API) that operates reliably.
- These services may be configured independently, or may depend on each other.
- Composition does not manage the lifecycle of an application

- **Overarching process' goal**:
  Present the complete application to the end-user

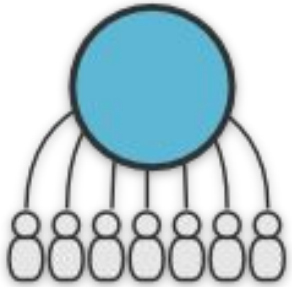
Centralised Application Composition

Centralised is a common (enterprise) approach

- centralised global model (representation of system)
- logically single controller

used by technologies:

- TOSCA
- Docker Compose



Centralised:

- Great for management (global control, oversight)
- Poor for scalability (the internet is not centralised!)

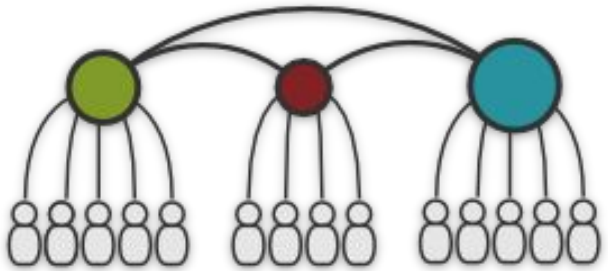
Decentralised Application Composition

Decentralised is **less common** in enterprise:

- decentralized local model (independent)
- logically multiple controllers with local state only, managed independently

Examples:

- Habitat.sh: closest approach today
- DNS, BGP, P2P

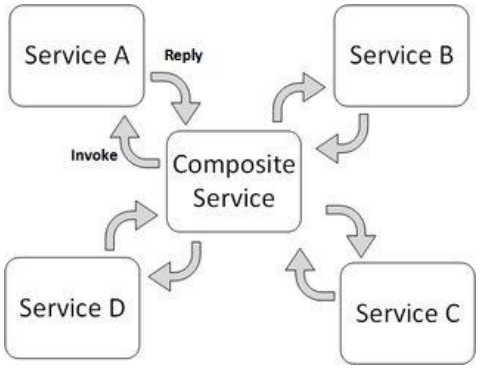


Decentralised:

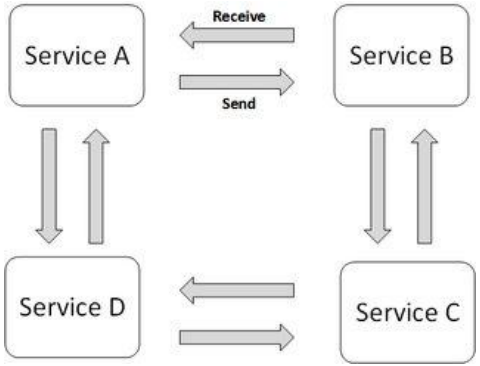
- Great for scalability
- Difficult to implement correctly and control

Composition Interactions

Orchestrated Compositions:
centralised process
one participant
global configuration



Choreographed Compositions:
decentralised process
multiple participants
local configurations



Composition Model Specification

Declarative

Example: `docker-compose`

```yaml
version: '2.1'

services:
  messaging:
    image: redis
    network:
        - mynet

networks:
  mynet:
    driver: "bridge"
```

- Runtime modifications difficult
- Debugging is difficult
- Independent of a language
- Requires design of document structure
  - States the system as it should be delivered
- Typical of today's approaches
  - Docker compose, K8S, CF manifest

Imperative

Example: kelda.io

```javascript
const app = new Container({
  name: 'appName',
  image: 'anyDockerImage',
});


const Redis = require('@kelda/redis');
const redis = new Redis(3, 'AUTH_PASSWORD');
allowTraffic(publicInternet, app, 80);
```

- Runtime modification easier (active, flexible)
- Could be debugged
- Uses existing languages, or specific DSL
  - less learning for dev
- Not very common today
  - More research-based
  - another example → Push2Cloud

Application Composition Standards

Industry standards are defined and driven out of the product and interest of the industry. Most of theses standards are developed in closed groups and require a consensus mechanism (slow process).

- **TOSCA**, CAMP, WS-BPEL

***De facto* standards** are those defined by a (single) entity and are adapted by the community so quickly/widely that the gain is huge popularity.

- **Docker Compose, Helm (Kubernetes)**

De jure standards *are those driven from government related entities. Community driven, industry or de facto standards can become de jure standards if a government officially embraces one (e.g. TCP/IP protocol, ASCII encoding).*

Topology and Orchestration Specification for Cloud Applications (**TOSCA**)

OASIS STANDARD

- YAML template document.
- Definition of building blocks for a cloud application.
- Modeling of components and the relationships between them.
- Standard (declarative description model and format) that enables portable applications and services.

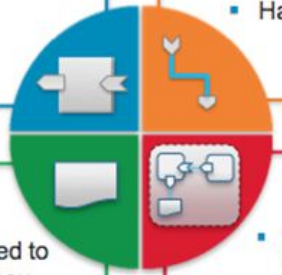
TOSCA Entities

Nodes

- Represent **Components** of an application or service and their **Properties**. Example nodes include:
  - **Infrastructure**: Compute, Network, Storage, etc.
  - **Platform**: OS, VM, DB, Web Server, etc.
  - **Granular**: functional Libraries, Modules, etc.
- Include **Operations** which are the management functions for the node
  - e.g. *deploy*(), *start*(), *stop*(), *connect*(), etc.
- Export their dependencies on other nodes as **Requirement** and **Capabilities**

Relationships

- Represent the logical **Relationships** between nodes
  - e.g. "*hostedOn*", "*connectsTo*", etc.
- Describes the valid **Source** and **Target nodes** they are designed to couple
  - e.g. source "*web application*" node is designed to "*connectTo*" a target "database" node
- Have their own **Properties** and **Constraints**



Artifacts

- Describe **Installables** and **Executables** required to instantiate and manage a service. Currently, they include:
- **Implementation Artifacts**:
  - Executables or ***Plans*** that implement a Node's or Relationship's Operations (e.g. a Bash script)
- **Deployment Artifacts**:
  - Installables of the components (e.g. a TAR file)

Service Templates

- **Group** the nodes and relationships that make up a service's topology
  - Allowing modeling of sub-topologies
- Service Templates *"look like nodes"* enabling:
  - **Composition** of applications from one or more service templates
  - **Substitution** of abstract Node types with available service templates of the same type

TOSCA Service Template Instance

General definition of a “service template instance” (STI)

- A solution stack is a set of software subsystems or components needed to create a complete platform
- Example:
  - LAMP
  - Wordpress

Basic Definition of a STI applied to a Resource Orchestrator

- Collection of resources that will be managed by Resource Orchestrator (e.g. K8s or docker swarm).
- Might include instances (VMs, containers), networks, subnets, routers, ports, router interfaces, security groups, security group rules, autoscaling rules, etc.

TOSCA Template

YAML format (once XML)

Consists of a header (meta data)
- Version
- Description

and 3 content sections
- Node templates
  - Definition of resources
- Inputs - parameters
- Outputs - results of creation

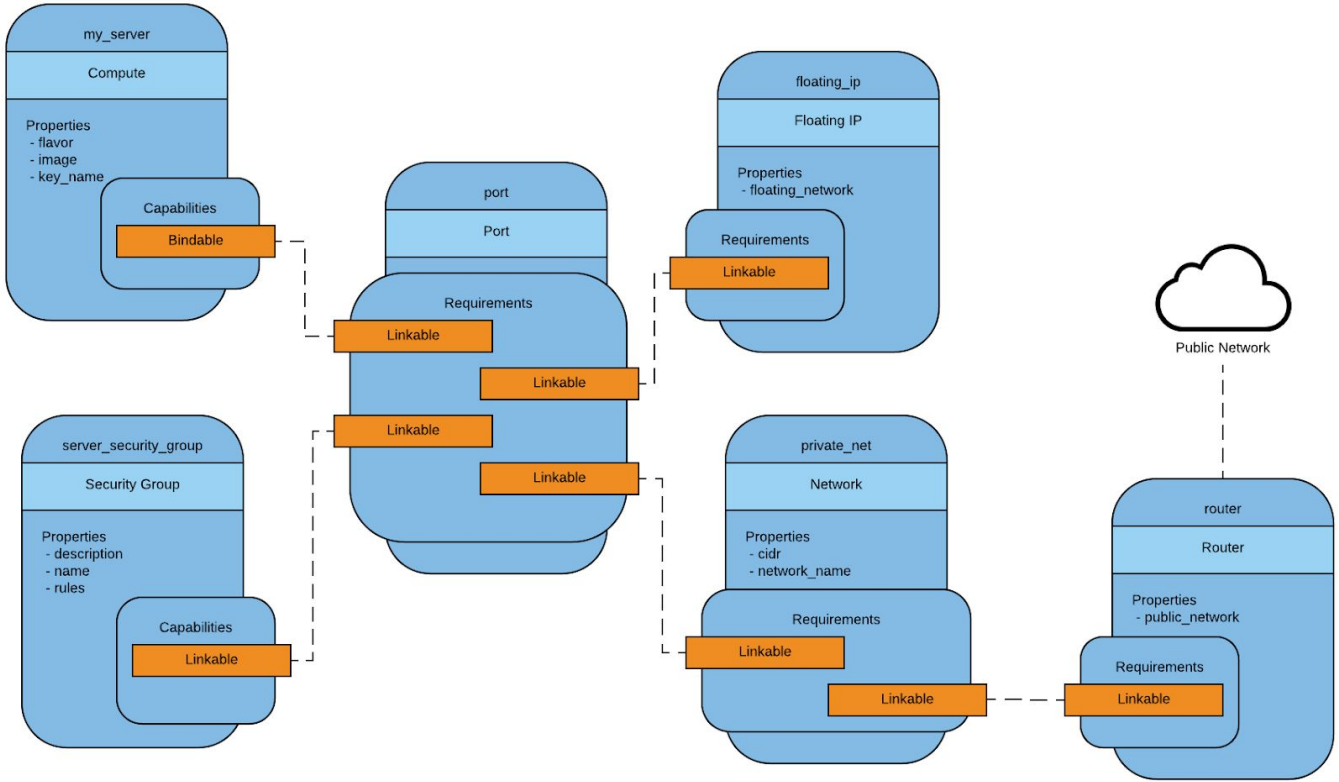
```yaml
tosca_definitions_version: tosca_simple_yaml_1_0
description: Monitoring Service Template

imports:
  - custom_types/tosca_compute.yaml
  - custom_types/tosca_floating_ip.yaml
  - custom_types/tosca_security_group.yaml
  - custom_types/tosca_router.yaml

topology_template:
  inputs:
    compute_image: ...
    compute_flavor: ...
    key_name: ...
    public_net: ...
    private_net_name: ...

  node_templates:
    my_server: ...
    private_net: ...
    router: ...
    floating_ip: ...
    port: ...
    server_security_group: ...

  outputs:
    it.hurtle.mon.dashboard: ...
```

Example:
- Monitoring as a Service
  - an IaaS application
  - one VM
  - attached to a network
  - has an external IP address
- Accessed under URL returned by `it.hurtle.mon.dashboard`

Complete TOSCA Example

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Monitoring Service Template

imports:
  - custom_types/tosca_compute.yaml
  - custom_types/tosca_floating_ip.yaml
  - custom_types/tosca_security_group.yaml
  - custom_types/tosca_router.yaml

topology_template:
  inputs:
    compute_image:
      type: string
      description: Compute instance image
      default: cactiSnapshot2
    compute_flavor:
      type: string
      description: Compute instance flavor
      default: m1.small
    key_name:
      type: string
      description: Key name
      default: mauikey
    public_net:
      type: string
      description: Name or ID of the public
   network
      default:
   77e659dd-f1b4-430c-ac6f-d92ec0137c85
    private_net_name:
      type: string
      description: Name of private network to be
   created
      default: fr-tosca-mon-net

  node_templates:
    my_server:
      type: hot.nodes.Compute
      properties:
        flavor: { get_input: compute_flavor }
        image: { get_input: compute_image }
        key_name: { get_input: key_name }
    private_net:
      type: tosca.nodes.network.Network
      properties:
        cidr: '192.168.1.0/24'
        network_name: { get_input: private_net_name }
    router:
      type: tosca.nodes.network.Router
      properties:
        public_network: { get_input: public_net }
      requirements:
        - link: private_net
    floating_ip:
      type: tosca.nodes.network.FloatingIP
      properties:
        floating_network: { get_input: public_net }
      requirements:
        - link: port
    port:
      type: tosca.nodes.network.Port
      properties:
        order: 0
      requirements:
        - link: private_net
        - link: server_security_group
        - binding: my_server
    server_security_group:
      type: tosca.nodes.network.SecurityGroup
      properties:
        description: Test group to demonstrate Neutron
  security group functionality with Heat.
        name: test-security-group
        rules:
          - protocol: tcp
            port_range_min: 22
            port_range_max: 22
          - protocol: tcp
            port_range_min: 80
            port_range_max: 80

  outputs:
    it.hurtle.mon.dashboard:
      description: Monitoring dashboard
      value:
        str_replace:
          template: http://host/cacti/
          params:
            host: { get_attribute: [my_server,
  public_address ] }
```

TOSCA Template Logical Diagram



TOSCA Execution Environment

Service Template Instance (STI) is created by a Resource Orchestrator (RO)

- ROs manages instances (VMs, containers), networks, subnets, routers, ports, router interfaces, security groups, security group rules, autoscaling rules, etc.
- ROs supports multiple cloud environments (e.g. AWS, OpenStack, K8S, Docker Swarm)

Native RO (executes TOSCA)

- Cloudify
- OpenBaton
- Alien4Cloud

Adapted RO (Translator-based)

- OpenStack TOSCA to Heat Translator
- OpenStack Tacker

Application Composition: (2) Docker Compose [de facto]

- Docker allows a user to deploy many containers, but requires many commands to do so!
- Link a new web container to a previously created db container (see Container Linking)
  - `docker run -d -P --name web --link db:db training/webapp python app.py`
- Each new container to be linked needs to run a similar command (e.g. to have a new instance of the web frontend connected to the same database)
  - No automation! Manual, error prone process.
- **Docker Compose** aims to solve the process of **deploying applications comprised of multiple containers (services)**.
- Docker Compose is a local software which communicated with a locally accessible docker engine (daemon).
  - Can be a single docker engine or a clustered docker swarm
- **Define a multi-container application in a single file**, then deploy an application in a single command: **`docker-compose up`**.

Docker Compose Application Model

Compose is based on YAML template files describing an application.
It has a simple basic structure of:

- **`version`**
  - Defines featureset of Docker Compose
  - Higher versions features are not backward compatible
  - 3 major versions: 1, 2 and 3 (latest).
- **`services`**
  - Definition of the containers to be used in the application composition
  - Each service can be connected to multiple networks and volumes
- **`volumes`**
  - Definition of 1 or more storage resources to be used
- **`networks`**
  - Definition of 1 or more network resources to be used


```
version: '2.1'
services:
 myapp:
   image: myapp:1.0
   environment:
      ADMIN_PASSWORD: shush
```

Docker Compose – Service Definition


```
api:                          ⇒ service name
  build:                      ⇒ builds a container if it doesn't exist
    - context: ./             ⇒ path to Dockerfile
  image: me/my_image          ⇒ names the container to use
  depends_on: db              ⇒ dependency on other service(s)
  environment:                ⇒ environment variables to parameterise the service
    MY_VAR: value
  ports:                      ⇒ expose port 5000 of the container to port 5000 on the host
    - "5000:5000"
  links:                      ⇒ links this service to another service (db)
    - db
  volumes:                    ⇒ data volume is mounted in /info directory in the container
    - data:/info
  networks                    ⇒ the api service is connected to the my-net docker network
    - my-net
```

Docker Compose – Extending Services and Service Health Checks

Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

```
api:
  extends:
    file: ./base.yml
    service: simple-api
  ...
```

⇒ takes a service "`simple-api`" from an existing definition
In the file `base.yml`
⇒ now can override or add new elements to the existing definition

```
api:
  healthcheck:
    test: "curl -f http://localhost:5000"
    interval: 10s
    timeout: 10s
    retries: 5
```

⇒ defines a check to see if the service is operational
⇒ this is the command for the check - binary in the container
⇒ how often to execute the test
⇒ how long to wait before trying again
⇒ how many times to try before it's a failed check

execute `docker-compose ps` to see the current status of health checks

```
        Name                      Command               State
-------------------------------------------------------------------
ccp2dvop1lab_dbext_1      docker-entrypoint.sh mysqld      Up
ccp2dvop1lab_horizon_1    /bin/sh -c /usr/sbin/apach ...   Up (unhealthy)
ccp2dvop1lab_keystone_1   /etc/bootstrap.sh                Up (healthy)
```

failed
successful

Docker Compose – Volumes and Network

`volumes:` ⇒ Declares `volumes` for use in the application

`data:` ⇒ One volume named `data`

`driver: "local"` ⇒ uses the default `local` storage driver

- Others include: `overlay(2), aufs, devicemapper, btrfs, zfs`

Execute `docker volume list` to see existing volumes

`docker volume --help` shows all management options

`networks:`

`my-net:` ⇒ Declares `networks` for use in the application

`driver: "bridge"` ⇒ One network named `my-net`

⇒ uses the default `bridge` storage driver

- Others include: `overlay` and `macvlan`

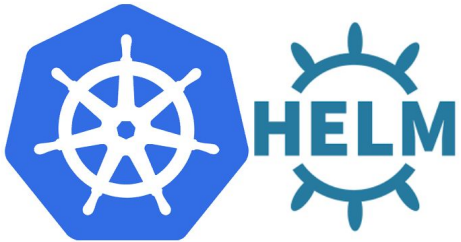
Execute `docker network list` to see existing networks

`docker network --help` shows all management options

Docker Compose – Creating the Application

- **`docker-compose -f <MANIFEST_FILE> up`**
  Deploys an application from the manifest (default `docker-compose.yml` in the current directory).
  Following steps happen with the manifest on the previous pages:
  `Building api...` ⇒ Build web image from Dockerfile
  `Starting api_1...` ⇒ Start api container first
  `api_1 | * Running on http://0.0.0.0:5000/`
- API starts first, but as it does not exist in the local registry or docker hub, it is built and then executed.
  - **Note:** a container can only be linked to **after** its creation.
- **`docker-compose up --scale api=2 --no-recreate`**
  Specify how many instances of a container type should be active.
  - e.g. `docker-compose up --scale api=2 --no-recreate` ensures that two containers (no more, no less) of the type api are active, it will create one more if there is only one, or stop any additional one. The flag `--no-recreate` ensures that the service is not restarted.
- To delete the application: **`docker-compose -f <MANIFEST_FILE> down`**
- More commands: **`docker-compose --help`**

Application Composition: (1) Helm [de facto]



- A “package manager” for Kubernetes (K8s)
- Bundles related k8s **manifests** together in a chart
  - Manifest is a k8s application description
  - Set of manifests used to deliver an application is known as a **chart**
- Enables reuse and composition through dependencies
- Provides templating for k8s manifests
  - allows for template variable overriding (from sub-charts, value files, command line options)
- Installing a chart creates a release
  - Maintains versions, allows for rolling updates & roll-back

Helm – Key Concepts

- **Chart**
  - a package; bundle of Kubernetes resources (pods, services, ...)
- **Release**
  - a chart instance is loaded into Kubernetes
  - same chart can be installed several times into the same cluster (separate namespace); each will have its own Release
- **Repository**
  - a repository of published Charts (e.g. like docker hub but for k8s)
  - Public repository here: https://artifacthub.io/ (Private can be hosted too)
- **Template**
  - a K8s configuration file mixed with Go/Sprig* template

* http://masterminds.github.io/sprig/

Helm-Chart package structure

Package structure of a Helm-Chart:

see: https://helm.sh/docs/topics/charts/

```
mypackage/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Chart.yaml structure


```yaml
apiVersion: The chart API version (required)
name: The name of the chart (required)
version: A SemVer 2 version (required)
kubeVersion: A SemVer range of compatible Kubernetes versions (optional)
description: A single-sentence description of this project (optional)
type: The type of the chart (optional)
keywords:
 - A list of keywords about this project (optional)
home: The URL of this projects home page (optional)
sources:
 - A list of URLs to source code for this project (optional)
dependencies: # A list of the chart requirements (optional)
 - name: The name of the chart (nginx)
   version: The version of the chart ("1.2.3")
   repository: (optional) The repository URL ("https://example.com/charts") or alias ("@repo-name")
   tags: # (optional)
     - Tags can be used to group charts for enabling/disabling together
   import-values: # (optional)
     - ImportValues holds the mapping of source values to parent key to be imported. Each item can be a string or pair of child/parent sublist items.
   alias: (optional) Alias to be used for the chart. Useful when you have to add the same chart multiple times
maintainers: # (optional)
 - name: The maintainers name (required for each maintainer)
   email: The maintainers email (optional for each maintainer)
   url: A URL for the maintainer (optional for each maintainer)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
appVersion: The version of the app that this contains (optional). Needn't be SemVer. Quotes recommended.
deprecated: Whether this chart is deprecated (optional, boolean)
annotations:
   example: A list of annotations keyed by name (optional).
```

see: https://helm.sh/docs/topics/charts/

Example Chart.yaml


```
apiVersion: v2
name: helm-osi
description: A Helm chart for the osi application
type: application
version: 0.1.0
appVersion: "1.0"
dependencies:
  - name: apache
    version: 1.2.3
    repository: https://example.com/charts
  - name: postgresql
    version: 0.1.0
  - name: microservice
    version: 0.1.0
    alias: microservice-order
  - name: microservice
    version: 0.1.0
    alias: microservice-shipping
  - name: microservice
    version: 0.1.0
    alias: microservice-invoicing
```

Version of the chart (semantic version number)

Version of the app (Text)

Subcharts this charts depends on

Download from repo to **`chart`** folder

If no repository is given, it has to be put to the **`chart`** folder manually

name of the chart in the chart folder

alias can be used to install multiple similar components using the same chart

Example: Deployment Template


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: {{ quote .Values.name }}
    version: {{ quote .Values.deploymentVersion }}
  name: {{ quote .Values.name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ quote .Values.name }}
  strategy: {}
  template:
    metadata:
      labels:
        app: {{ quote .Values.name }}
    spec:
      containers:
      - name: {{ quote .Values.name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        ports:
        - containerPort: {{ .Values.port }}
        resources: {}
```

see: https://helm.sh/docs/chart_template_guide/

{{ Placeholder }}
values will be taken from the `--set` arguments, the provided `values` file or the default `values.yaml` of the chart.

Template Functions allow more complex operations (**quote, print, b64enc, b64dec, include, ...**) .
see https://helm.sh/docs/chart_template_guide/function_list/

If no value is found an alternative default value can be specified.

Also chart (`.Chart`) or release (`.Release`) specific values can be used

Example: values.yaml


```
name: apache
deploymentVersion: 1.0
replicaCount: 1
port: 80
host: web.160.85.253.<X>.nip.io

image:
  repository: registry.localhost:5000/ccp2-apache
  tag: latest
  pullPolicy: Always

service:
  type: ClusterIP
  port: 80
```

see: https://helm.sh/docs/chart_template_guide/

Helm commands

Looking up charts

see: https://helm.sh/docs/helm/helm/

- `helm search hub postgresql` in artifacthub.io
- `helm search repo postgresql` in the repo list

Managing repos

- `helm repo list`
- `helm repo add bitnami https://charts.bitnami.com/bitnami`
- `helm repo delete bitnami`

Manage applications

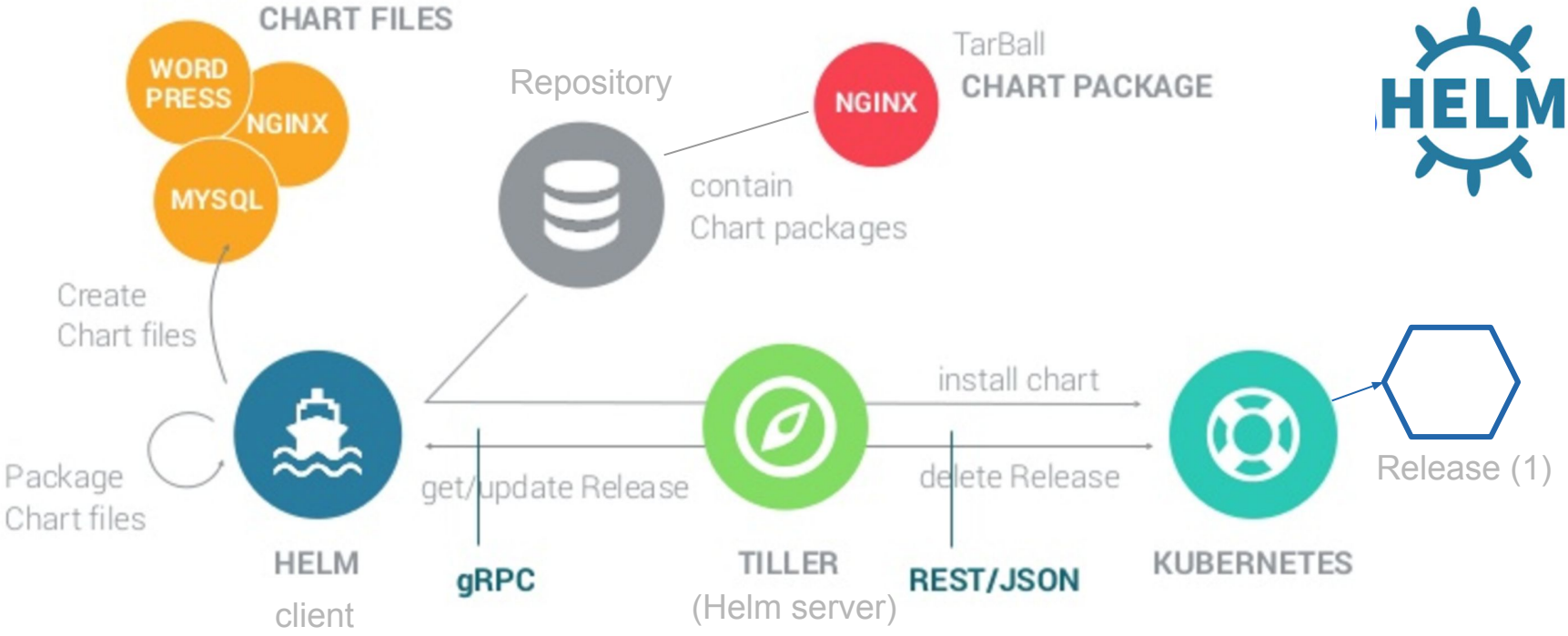
- `helm install -f myvalues.yaml myDB bitnami/postgresql`
- `helm install --set auth.username=dbuser myDB bitnami/postgresql`
- `helm list`
- `helm uninstall myDB`

Manage packages

- `helm pull bitnami/postgresql` Downloads the chart package (tgz)
- `helm package ./ois-package/` Create a chart package from a path

Appendix

Helm – Architecture v1 (deprecated)



Helm – Getting Started

Prerequisites

- A Kubernetes Cluster
  - Many ways to install from small to large
  - See `minikube` (small local install) or `kops` (large AWS EC2 install)
- The helm binary

Install & Upgrade

- `helm init` ⇒ sets environment and installs Tiller on the k8s cluster
- `helm init --upgrade`
- `helm --help` ⇒ all you can do with Helm

TOSCA Custom Type

tosca_router.yaml 378 Bytes

```
1  tosca_definitions_version: tosca_simple_yaml_1_0
2  node_types:
3    tosca.nodes.network.Router:
4      derived_from: tosca.nodes.Root
5      properties:
6        public_network:
7          type: string
8      requirements:
9        - link:
10           capability: tosca.capabilities.network.Linkable
11           relationship: tosca.relationships.network.LinksTo
12           node: tosca.nodes.network.Network
```

Note:

- `Linkable`
- `LinksTo`

TOSCA Concepts

TOSCA Concepts in OOP Terms:

- **Service Template** = class
- **Service Template Instance** = Instance of a template (Object)
- **Inputs** = Constructor Arguments
- **Node Templates** = Variables/Content of the class
- **Outputs** = Getter Methods/Attributes