DTREE

Identify relationships between resources in a geo-distributed microservices scenario.

TABLE OF CONTENTS

01

CONTEXT & CHOICES

02

IMPLEMENTATION

03

RESULTS

04

DEMO

1. CONTEXT & CHOICES

Edge computing:

Solves two main challenges of cloud computing

- High latency
- Disconnection between sites

Cheops: Generic and non intrusive solution, service mesh composed of cheops core and cheops glue

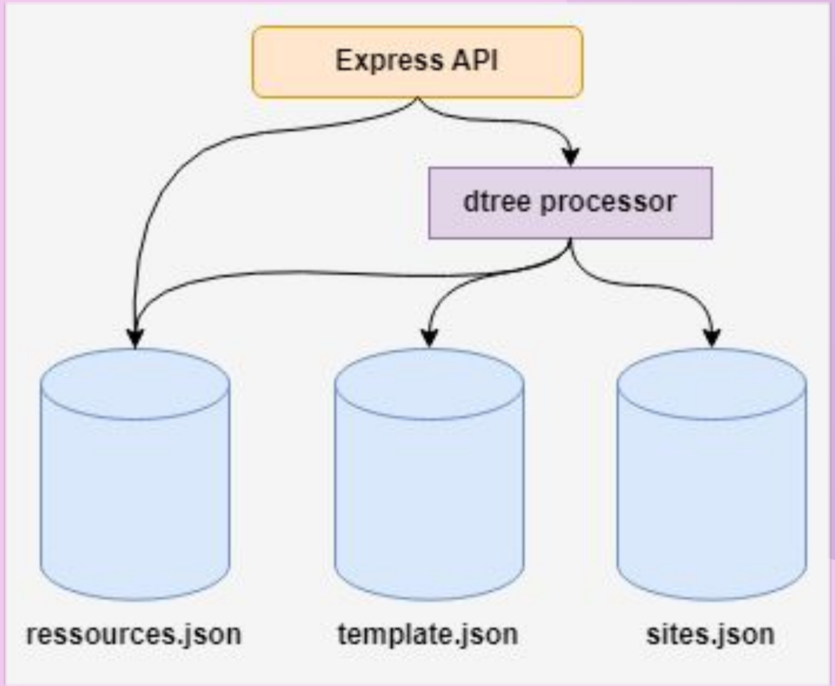
Goal of our project: Based on the JSON K8S definition, be able to identify the objects involved in a geo distributed request in order to determine if a request can be executed or not.

Requirement: The code must be as generic as possible. No hard dependency to K8S

Started with go -> Development was tedious and velocity was low.

Switched to javascript -> Parsing and validation is trivial, high velocity of development for a POC. The language is optimized by design to deal with JSON files.

2. IMPLEMENTATION - ARCHITECTURE



Site definition

Exposition

- /ressources
- /verify

Dtree computation

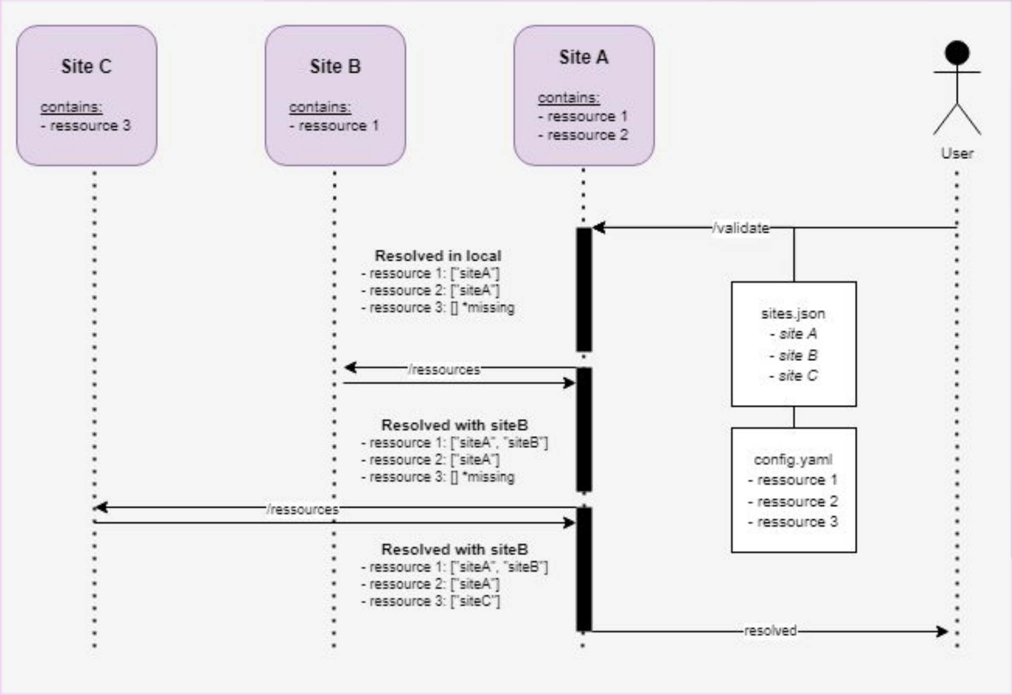
- Generic depth first search algorithm based on the JSON schema spec
- Validation using a battle tested library (Ajv)

Assets

- Ressources: DB of the site, JSON file for POC
- Template: K8S definitions
- Sites: Addresses and ids of the other sites

2. IMPLEMENTATION - ALGORITHM

1. Verify config conformity to template (ajv)
2. Check in the local database for ressources
3. If not fulfilled, check for remote locations until all requirements are fulfilled



Resolved in local
- ressource 1: ["siteA"]
- ressource 2: ["siteA"]
- ressource 3: [] *missing

Resolved with siteB
- ressource 1: ["siteA", "siteB"]
- ressource 2: ["siteA"]
- ressource 3: [] *missing

Resolved with siteB
- ressource 1: ["siteA", "siteB"]
- ressource 2: ["siteA"]
- ressource 3: ["siteC"]

2. IMPLEMENTATION - ALGORITHM

Definitions & User config files

config.yaml

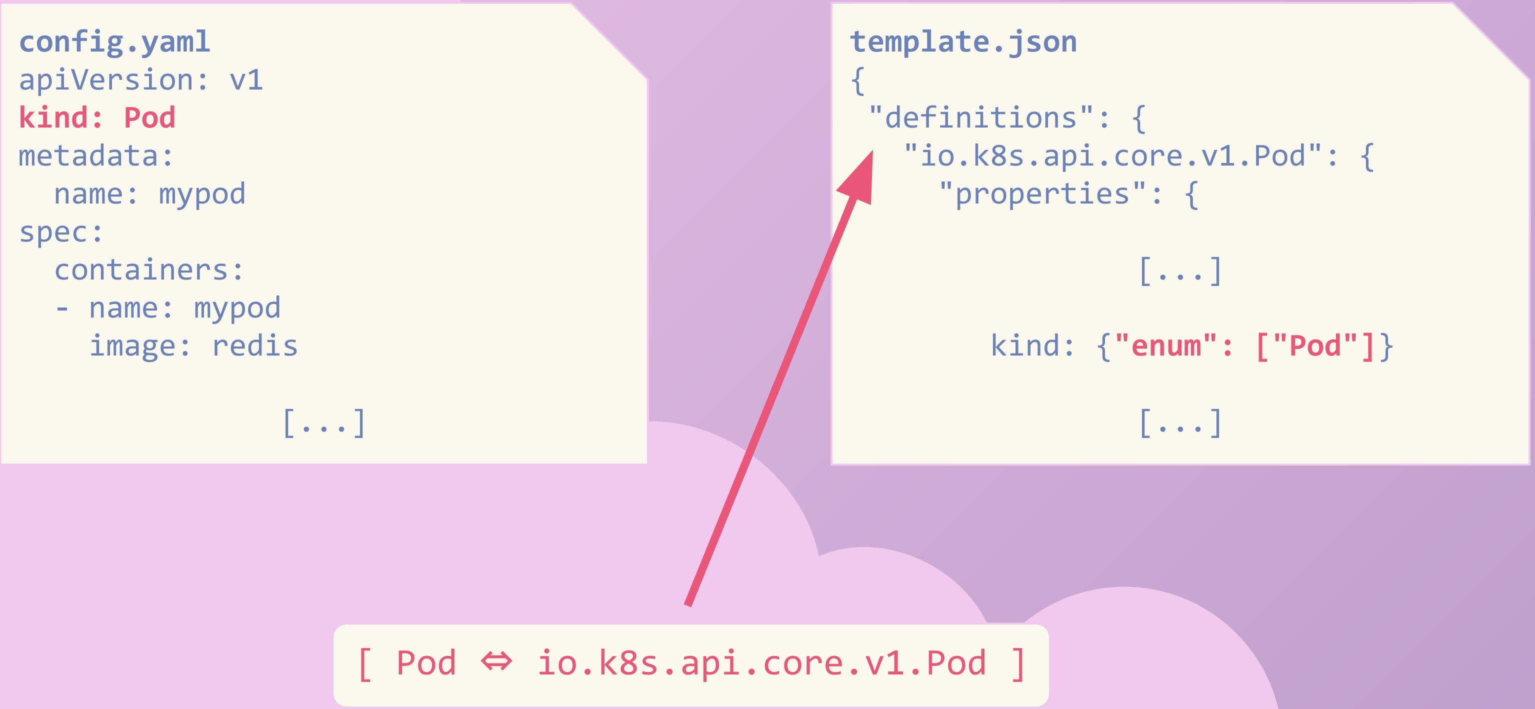
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      optional: false
```

template.json

```
{
 "definitions": {
   "io.k8s.api.core.v1.Pod": {
     "properties": {
       "apiVersion": {
         "type": "string"
       },
       "kind": {
         "type": "string",
         "enum": ["Pod"]
       },
     },
     "metadata": {
     "$ref":"#/defs.[...].ObjectMeta",
   },
   [...] // 18k lines of definition
}
```

2. IMPLEMENTATION - ALGORITHM

Step 1 : Determine the config entry point



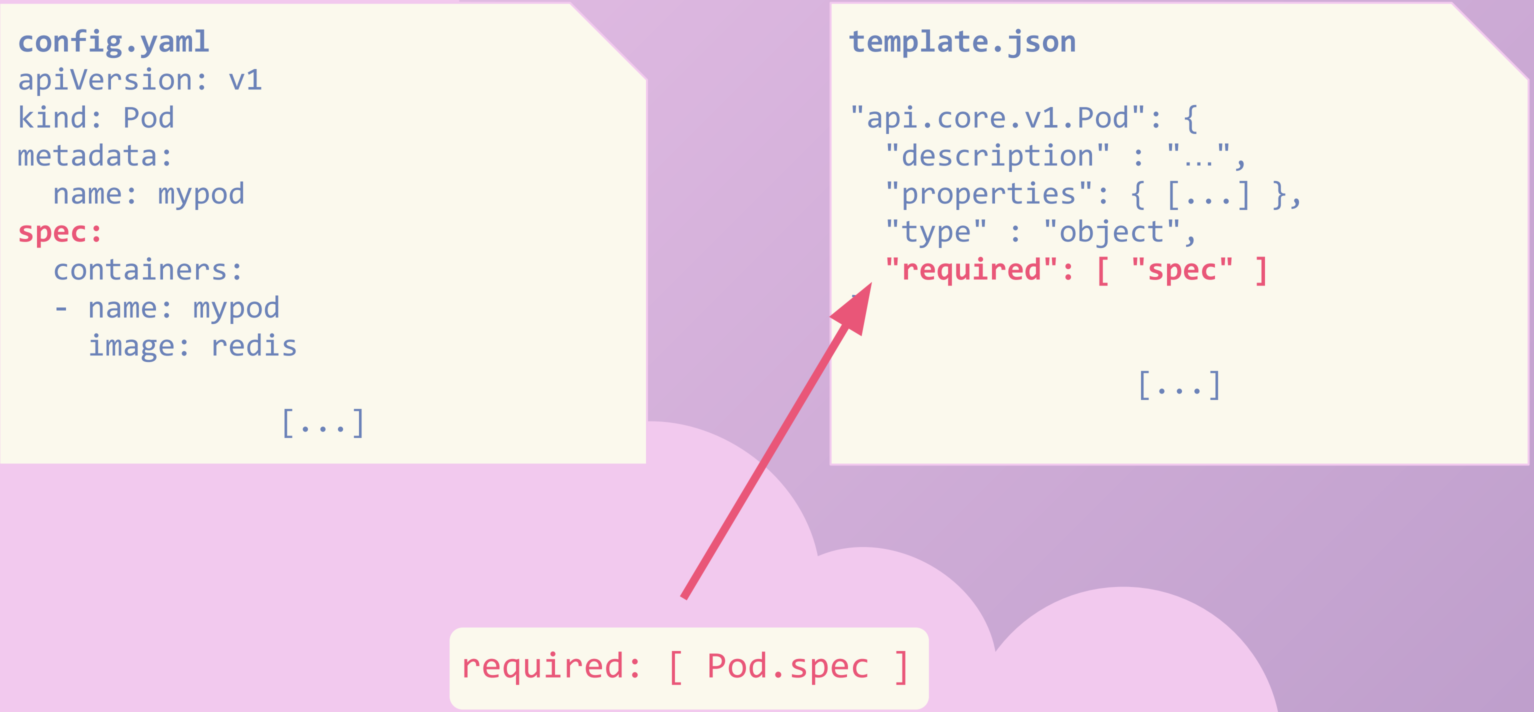
```
config.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis

               [...]
```

```
template.json
{
 "definitions": {
   "io.k8s.api.core.v1.Pod": {
     "properties": {

                 [...]

        kind: {"enum": ["Pod"]}

                 [...]
```

[Pod ⇔ io.k8s.api.core.v1.Pod]

2. IMPLEMENTATION - ALGORITHM

Step 2 : Check for required config fields



```
config.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis

               [...]
```

```
template.json

"api.core.v1.Pod": {
  "description" : "...",
  "properties": { [...] },
  "type" : "object",
  "required": [ "spec" ]
}

                [...]
```

```
required: [ Pod.spec ]
```

2. IMPLEMENTATION - ALGORITHM

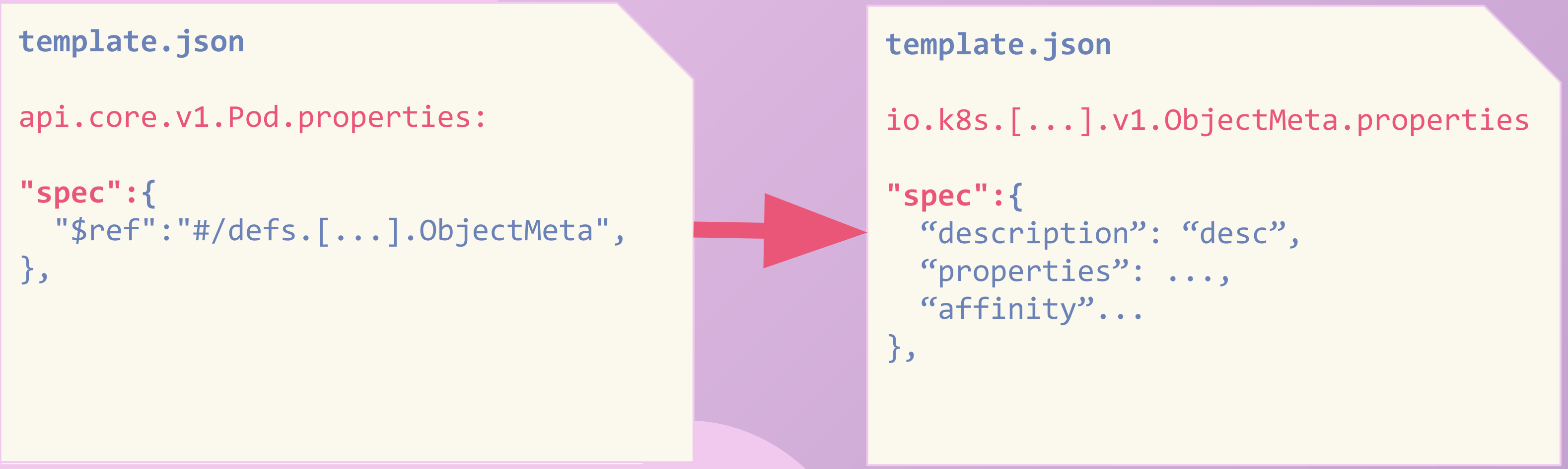
Step 3 : Recursively traverse the tree

```
config.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis

               [...]
```

```
template.json
api.core.v1.Pod.properties:
"metadata":{
  "$ref":"#/defs.[...].ObjectMeta",
},
"spec":{
  "$ref":"#/defs.[...].ObjectMeta",
},
"status":{
  "$ref":"#/defs.[...].ObjectMeta",
},
```

2. IMPLEMENTATION - ALGORITHM

Step 4 : Replace references by their values

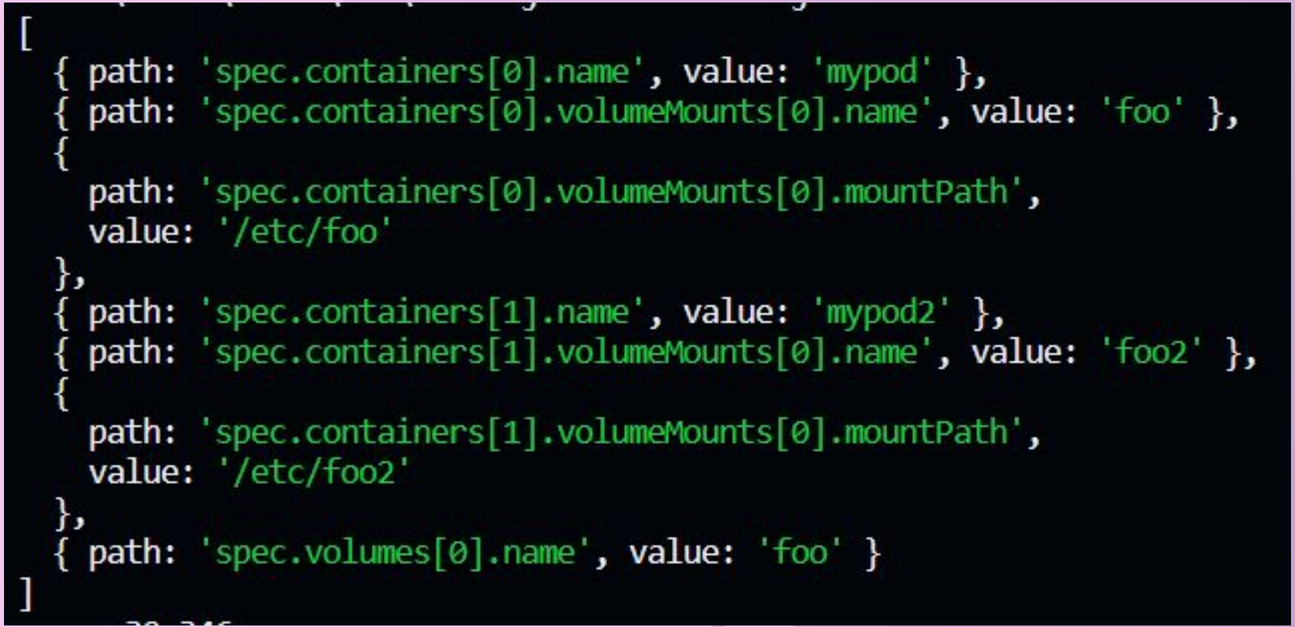


```
template.json

api.core.v1.Pod.properties:

"spec":{
  "$ref":"#/defs.[...].ObjectMeta",
},
```

```
template.json

io.k8s.[...].v1.ObjectMeta.properties

"spec":{
  “description”: “desc”,
  “properties”: ...,
  “affinity”...
},
```

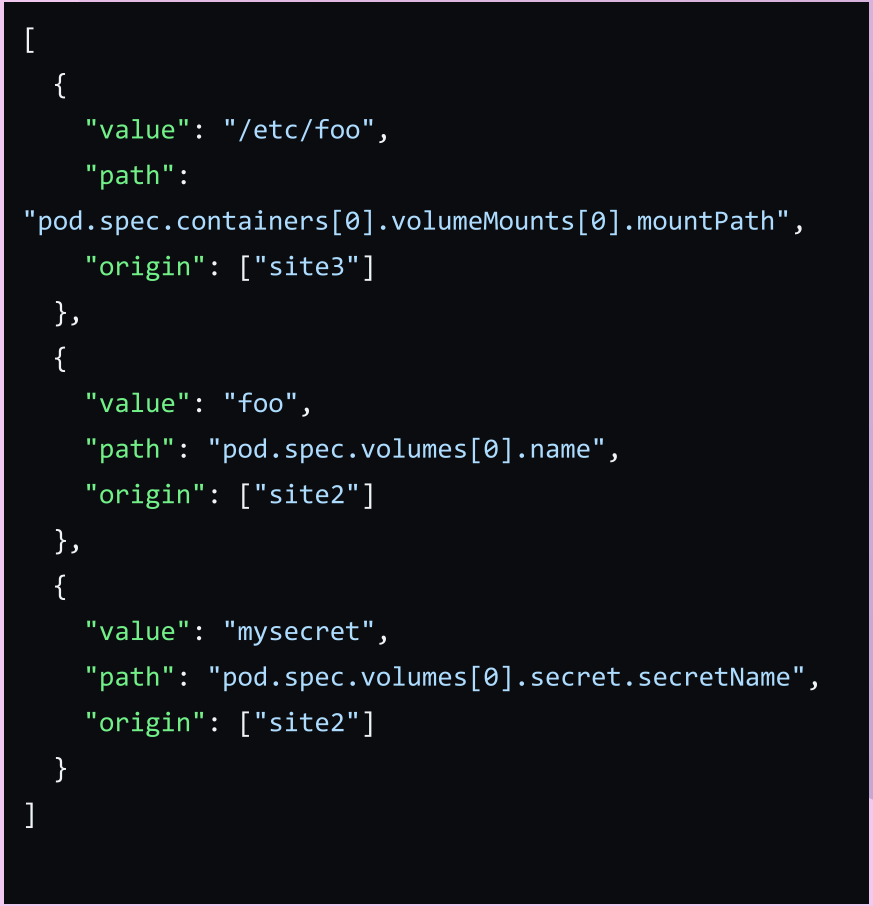
Iterate on references and go back to step 2

Example of config output



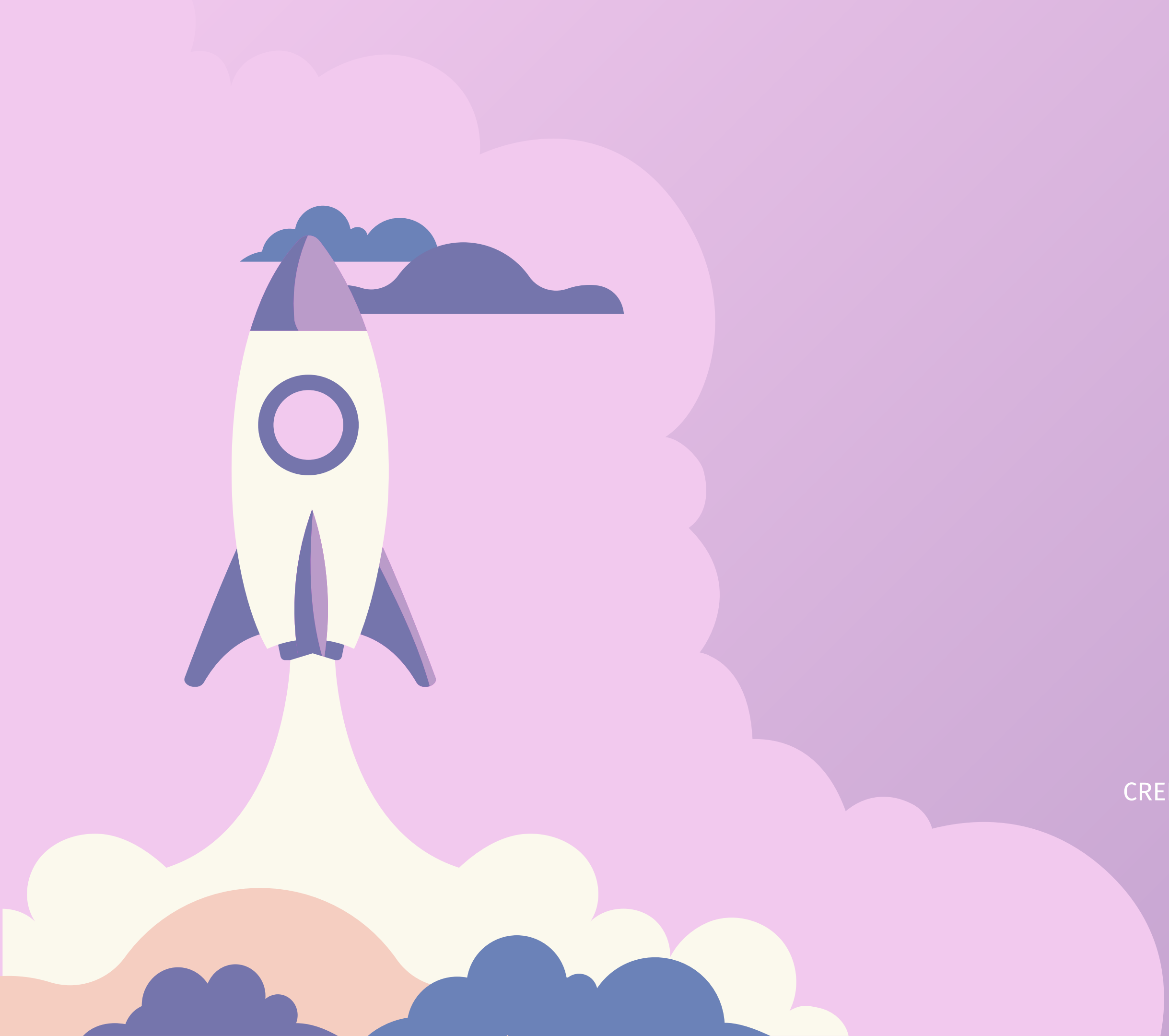
```
[
  { path: 'spec.containers[0].name', value: 'mypod' },
  { path: 'spec.containers[0].volumeMounts[0].name', value: 'foo' },
  {
    path: 'spec.containers[0].volumeMounts[0].mountPath',
    value: '/etc/foo'
  },
  { path: 'spec.containers[1].name', value: 'mypod2' },
  { path: 'spec.containers[1].volumeMounts[0].name', value: 'foo2' },
  {
    path: 'spec.containers[1].volumeMounts[0].mountPath',
    value: '/etc/foo2'
  },
  { path: 'spec.volumes[0].name', value: 'foo' }
]
```

3. RESULTS



```
[
  {
    "value": "/etc/foo",
    "path":
"pod.spec.containers[0].volumeMounts[0].mountPath",
    "origin": ["site3"]
  },
  {
    "value": "foo",
    "path": "pod.spec.volumes[0].name",
    "origin": ["site2"]
  },
  {
    "value": "mysecret",
    "path": "pod.spec.volumes[0].secret.secretName",
    "origin": ["site2"]
  }
]
```

Demo



Thanks!

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution