# Next Generation of Platform Engineering Using Kcp and Crossplane

*Lovro Sviben & Simon Bein*

## Simon Bein

Senior Software Engineer @ Kubermatic

github.com/SimonTheLeg

linkedin.com/in/simon-bein

twitch.tv/simonbuilds

I also maintain konf, the “best kubeconfig manager out there” (citation needed)

## Lovro Sviben

Senior Distributed Systems Engineer @ Upbound

github.com/lsviben

linkedin.com/in/lovro-sviben

## Common Understanding of Platform Engineering

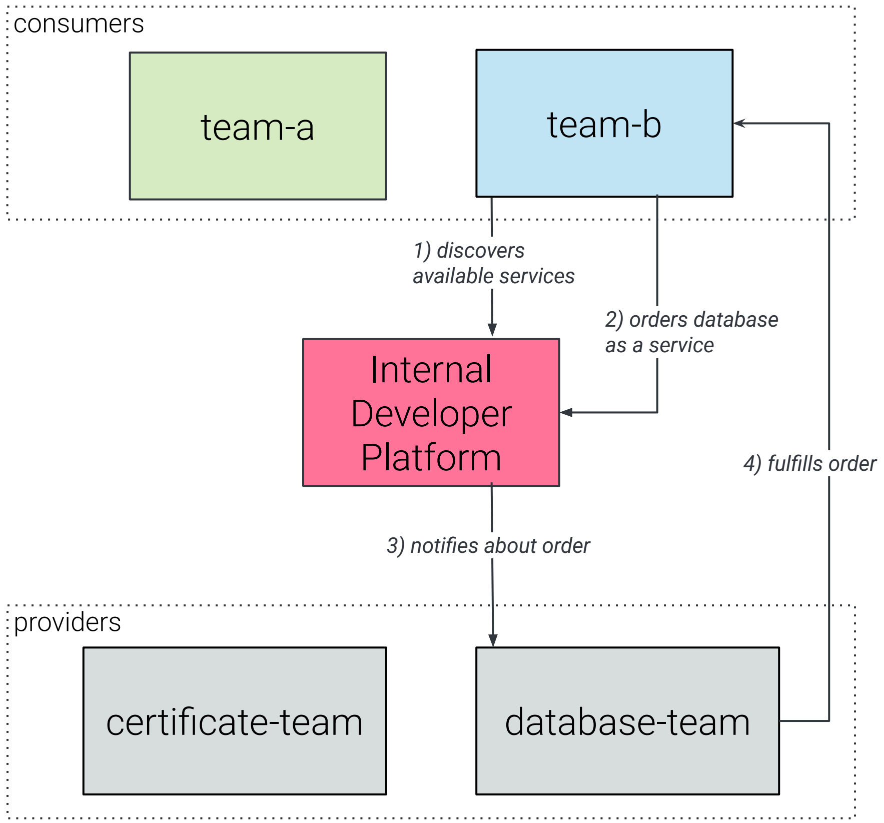
## Within this talk, Platform Engineering is defined as

Creation and Maintenance of an automated,

self-service platform for services inside a company

## Enough Theory - Here’s an Example

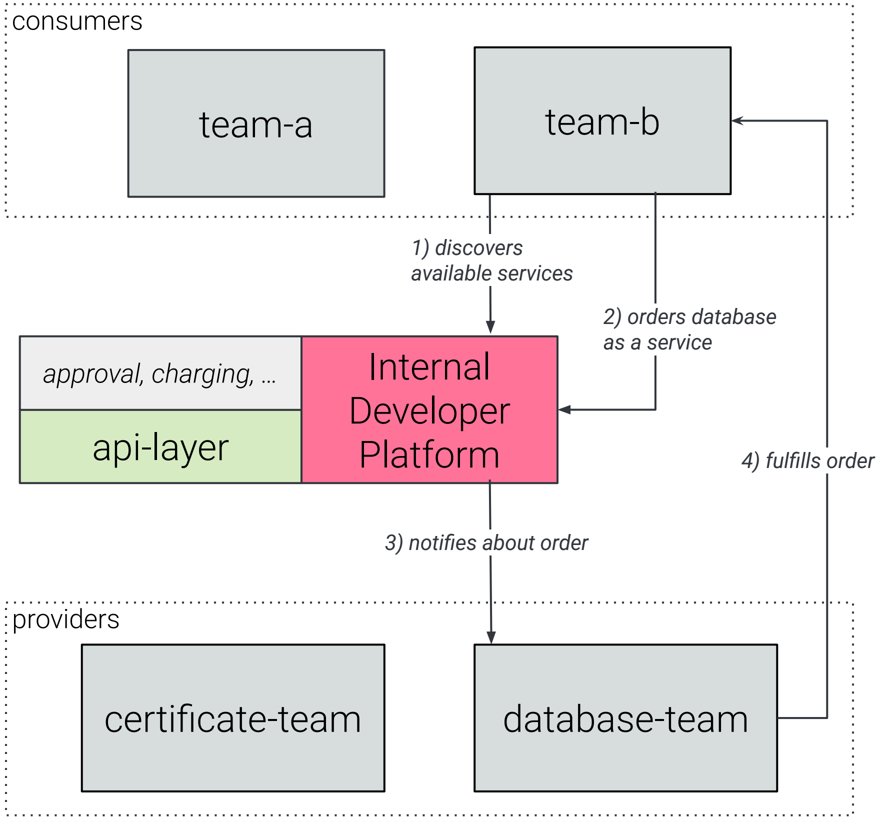
Creation and Maintenance of an automated, self-service platform for services inside a company



# This Talk Focuses on the API Layer of such a Platform

API Layer is responsible for:

- Publishing and discovery of services
- Lifecycle and ownership of Service APIs
- Standardized Data interface for other components



## Why Kubernetes is interesting as an Api-Layer

# (1) Sophisticated API Conventions within Kubernetes

```
/apis/<group>/<version>/[namespaces/<namespace>/]<resourcetype>[/<name>]
```

APIs in Kubernetes are **grouped**.

Each API group is also **versioned**.

Resources are optionally **namespaced**.

Resources have a specific **resource type** that defines their schema.

Resources are uniquely **named**.

## (2) The Kubernetes API is extendable

We can extend APIs available in the kube-apiserver using CRDs

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: shirts.stable.example.com
spec:
  group: stable.example.com
  scope: Namespaced
  names:
    kind: Shirt
  versions:
  - name: v1
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              color:
                type: string
              size:
                type: string
                ...
```

## Some Examples

Crossplane

ROOK

Fluid

KubeDB

KubeVirt

Kyverno

## The Kubernetes API is pretty awesome!

(that’s it. That’s the ~~tweet~~ ~~post~~ slide)

## But …

- APIs (CRDs) are cluster-scoped, so everyone shares them

## Let’s Give Everyone a Cluster!

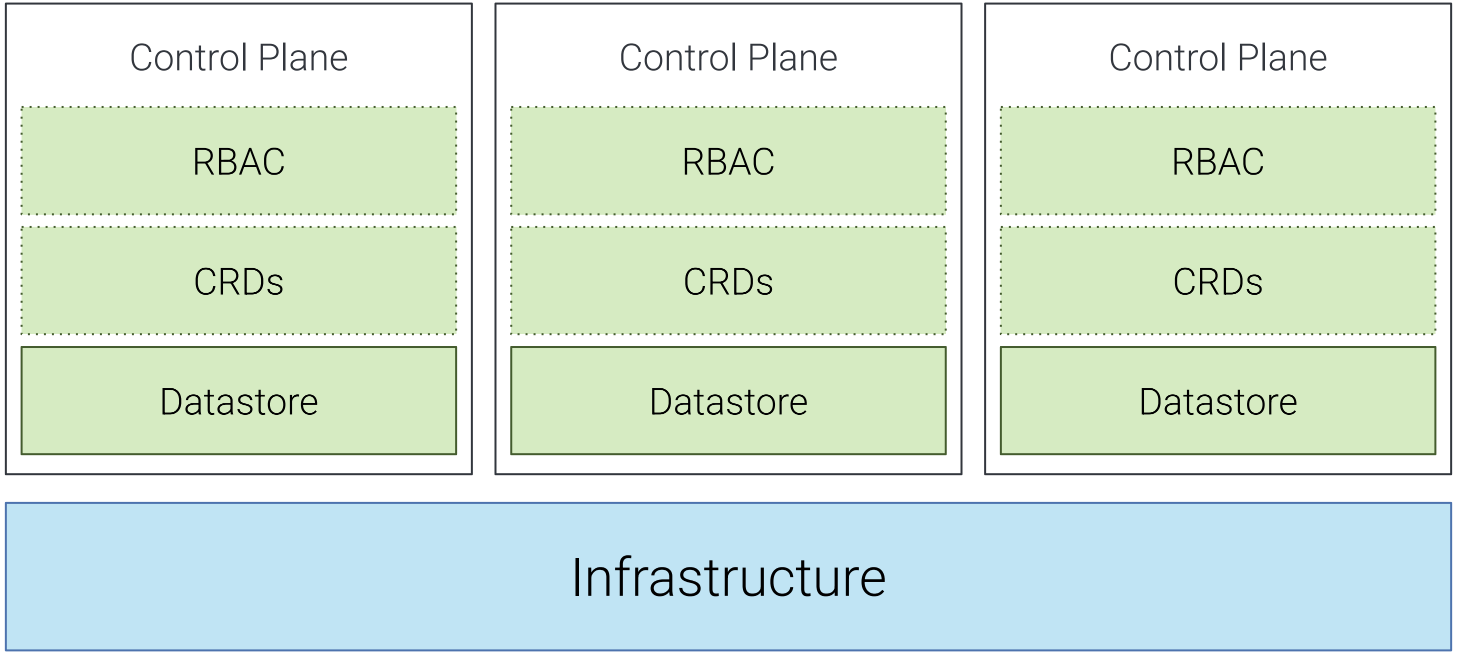
## But …

- APIs (CRDs) are cluster-scoped, so everyone shares them
- Starting a new cluster with its own api-server is time and resource intensive

## Lightweight Clusters

to the rescue?

## Hosted Control Planes

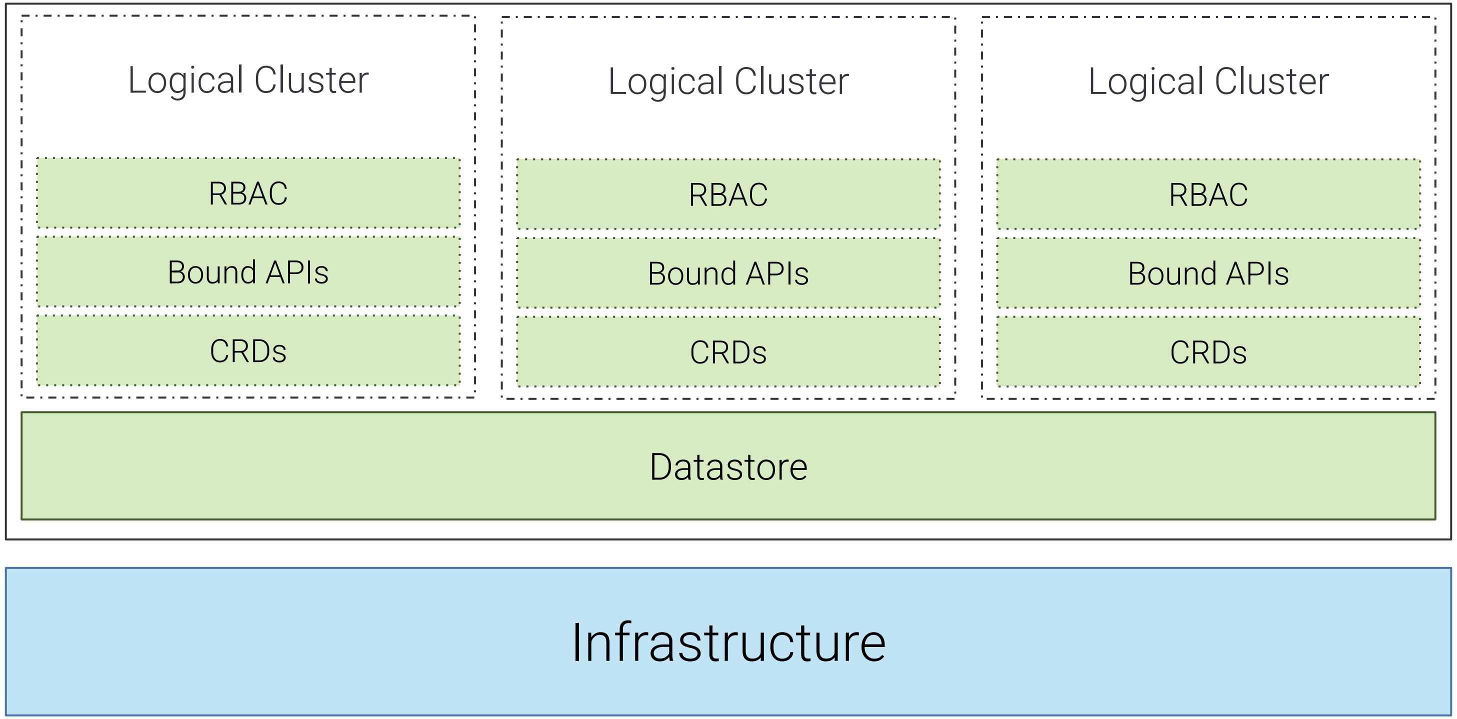


## But ...

- APIs (CRDs) are cluster-scoped, so everyone shares them
- Starting a new cluster with its own api-server is time and resource intensive
- Sharing apis between a large number of clusters is cumbersome

## What if Control Planes share data?

## “Logical” Clusters



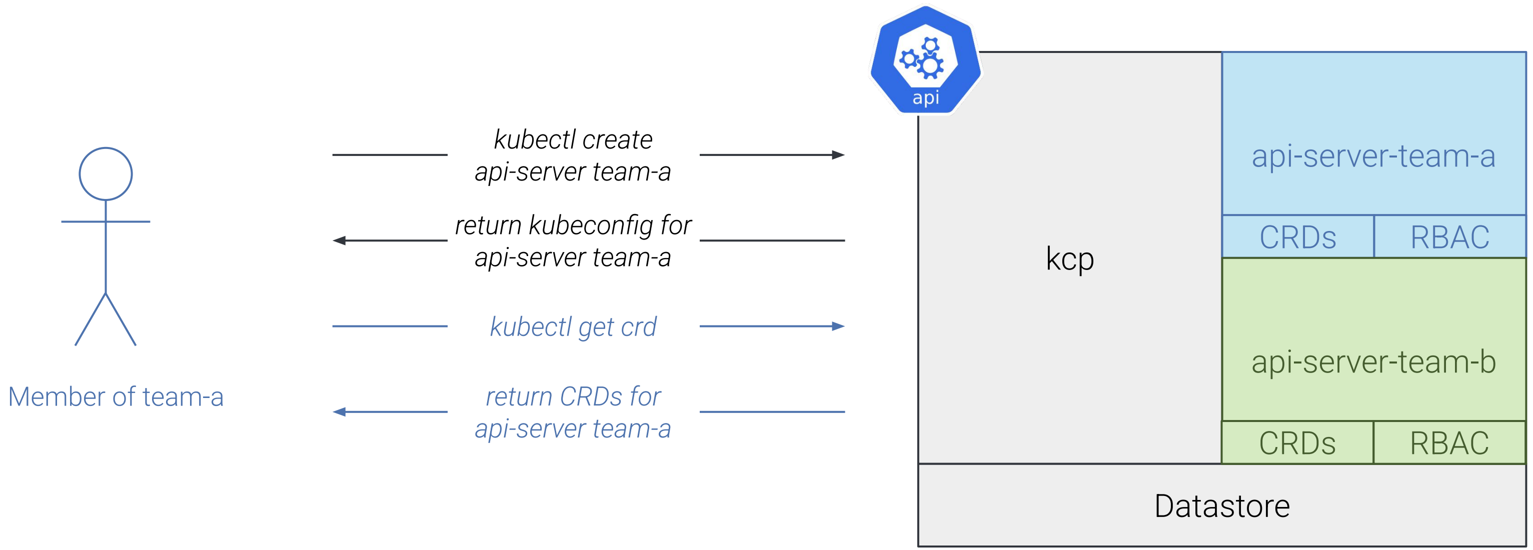
## This brings us to kcp

CLOUD NATIVE
COMPUTING FOUNDATION

**Sandbox project**
(since end of 2023)

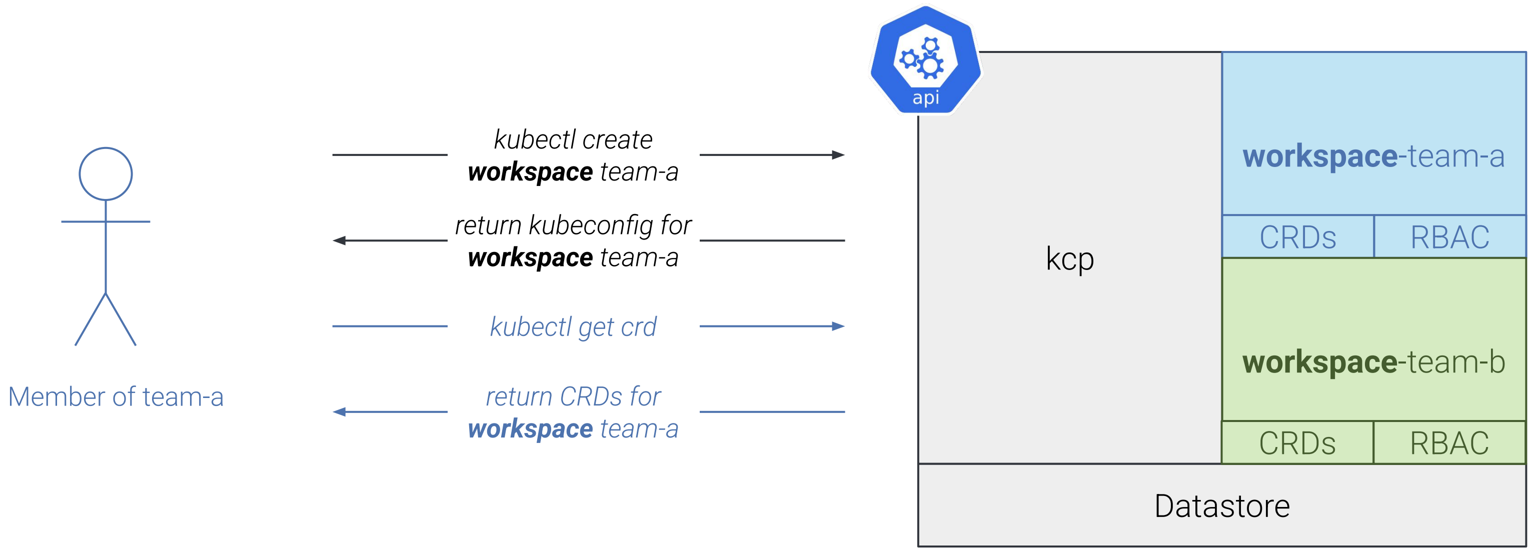
“A horizontally scalable control-plane for Kubernetes-style APIs”

# A practical example of interacting with kcp



Member of team-a
kubectl create api-server team-a
return kubeconfig for api-server team-a
kubectl get crd
return CRDs for api-server team-a
api
kcp
api-server-team-a
CRDs
RBAC
api-server-team-b
CRDs
RBAC
Datastore

# Instead of “api-servers”, we call them “workspaces”



api
Member of team-a
*kubectl create **workspace** team-a*
*return kubeconfig for **workspace** team-a*
*kubectl get crd*
*return CRDs for **workspace** team-a*
kcp
**workspace**-team-a
CRDs
RBAC
**workspace**-team-b
CRDs
RBAC
Datastore

## Workspace

A multi-tenancy **unit of isolation** in kcp.

Each workspaces has its own available **API resource types**.

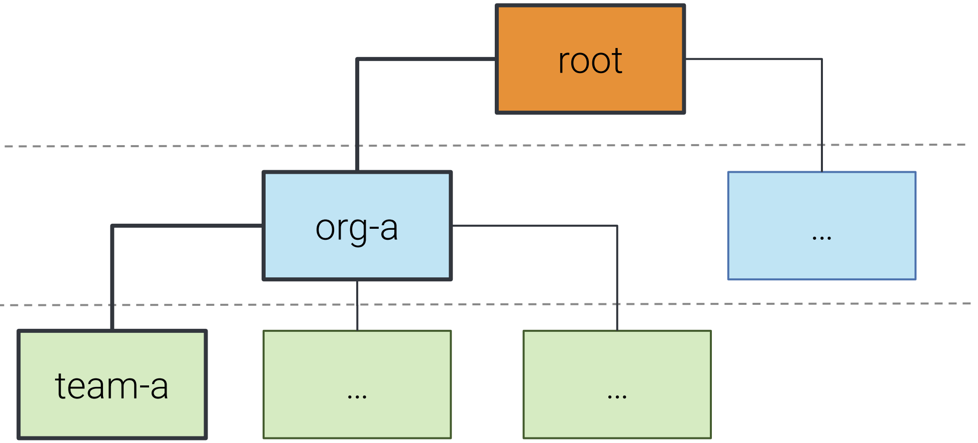
API **objects** are not shared across workspaces.

Delegation of **administrative permissions** to workspace owners.

Workspaces are **cheap**.

## Workspaces are organized in a tree

```
https://kcp:6443/clusters/root

https://kcp:6443/clusters/root:org-a

https://kcp:6443/clusters/root:org-a:team-a
```



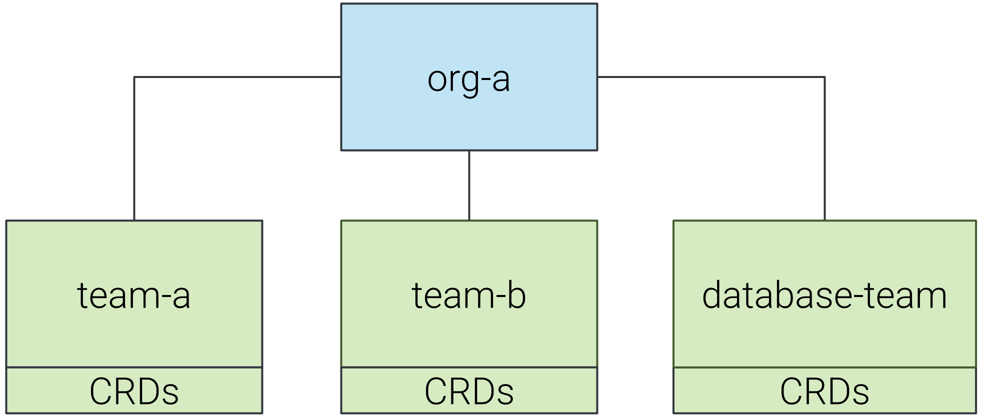
## Practical Example

```
$ kcp start --bind-address=127.0.0.1
… lots of log output


$ export KUBECONFIG=.kcp/admin.kubeconfig

$ kubectl create configmap foo
configmap/foo created

$ kubectl create workspace team-a
Workspace "team-a" (type root:organization) is ready to use.

$ kubectl get ws
NAME     TYPE   REGION   PHASE   URL          AGE
team-a   team            Ready   https://…    3m23s

$ kubectl ws :root:team-a
Current workspace is "root:team-a" (type root:team).

$ kubectl get configmap foo
Error from server (NotFound): configmaps "foo" not found
```

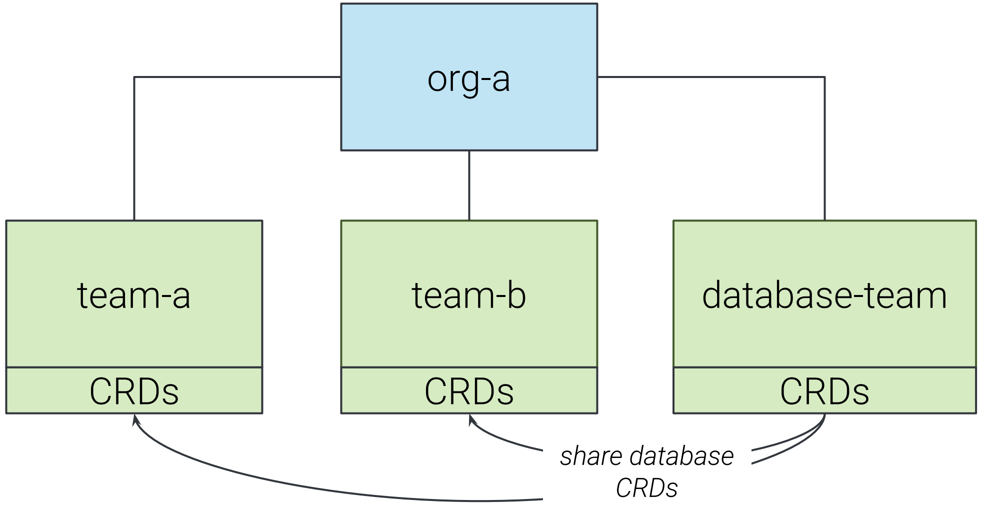
## Workspaces allow filesystem-like navigation

```
$ kubectl ws .
Current workspace is "root".

$ kubectl get ws
NAME    TYPE           REGION   PHASE   URL          AGE
org-a   organization            Ready   https://…    69d
org-b   organization            Ready   https://…    65d

$ kubectl ws org-a
Current workspace is "root:org-a" (type root:organization).

$ kubectl get ws
NAME     TYPE   REGION   PHASE   URL         AGE
team-a   team            Ready   https://…   3m23s
team-b   team            Ready   https://…   3m18s


$ kubectl ws team-a
Current workspace is "root:org-a:team-a" (type root:team).
```

## When we go back to our example...



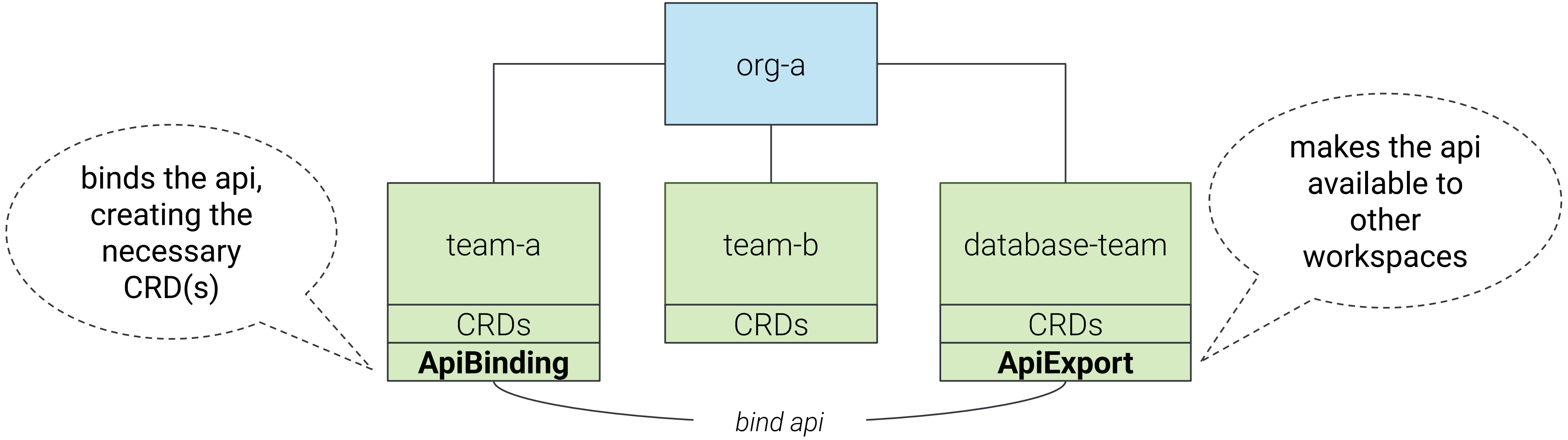
## ... the reality is closer to this



## The API Marketplace

Sharing is Caring

## How this can be achieved in kcp: ApiExport & ApiBinding


binds the api, creating the necessary CRD(s)
org-a
team-a
CRDs
ApiBinding
team-b
CRDs
database-team
CRDs
ApiExport
makes the api available to other workspaces
bind api

Infrastructure + Service teams are not in the business of making APIs discoverable and consumable.

**Platform teams are.**

## Create APIs with APIExports

## APIExport

```yaml
apiVersion: apis.kcp.io/v1alpha1
kind: APIExport
metadata:
  name: databases.demo.example.com
spec:
  latestResourceSchemas:
    - v1.databases.demo.example.com
    - v1.databaseDrivers.demo.example.com
```

Resource schemas define resources, just like CRDs.

## Enable APIs with APIBindings

## Powered by APIBindings

```
$ kubectl get apibindings
NAME                    AGE    READY
tenancy.kcp.io-3wb5h    30d    True
topology.kcp.io-cua3o   30d    True
```

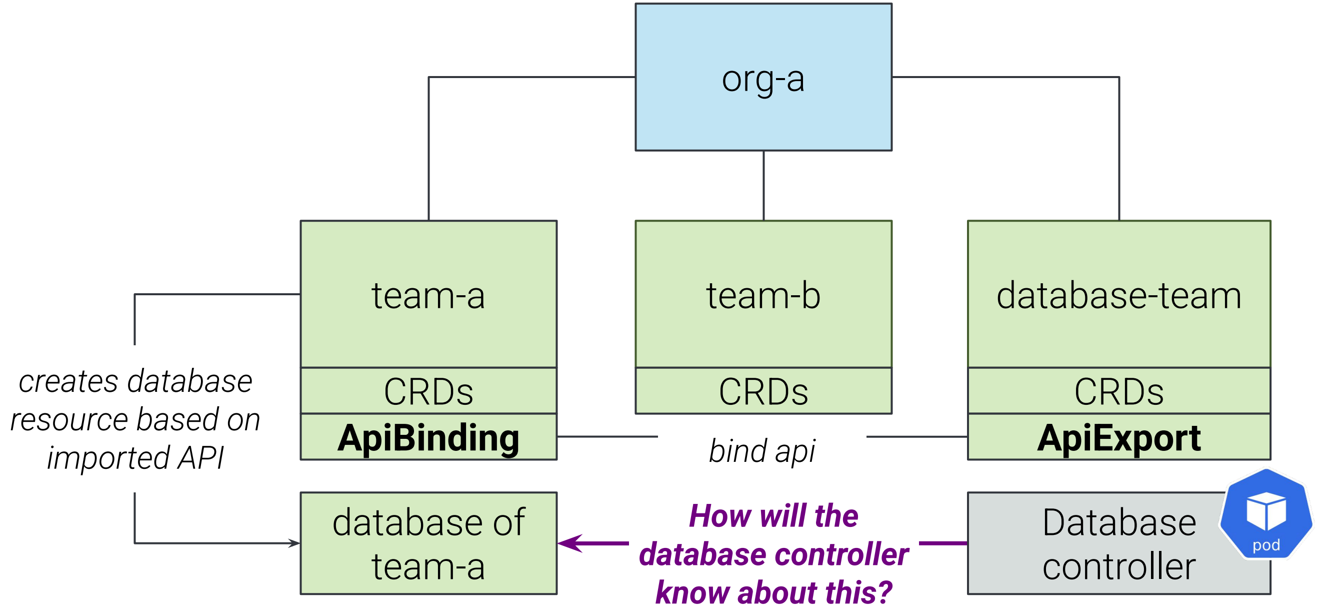
```yaml
apiVersion: apis.kcp.io/v1alpha1
kind: APIBinding
metadata:
  name: tenancy.kcp.io-3wb5h
spec:
  reference:
    export:
      name: databases.demo.example.com
      path: root:database-team
```

**This references an APIExport in a different workspace!**

## Discovery of API-Consumers

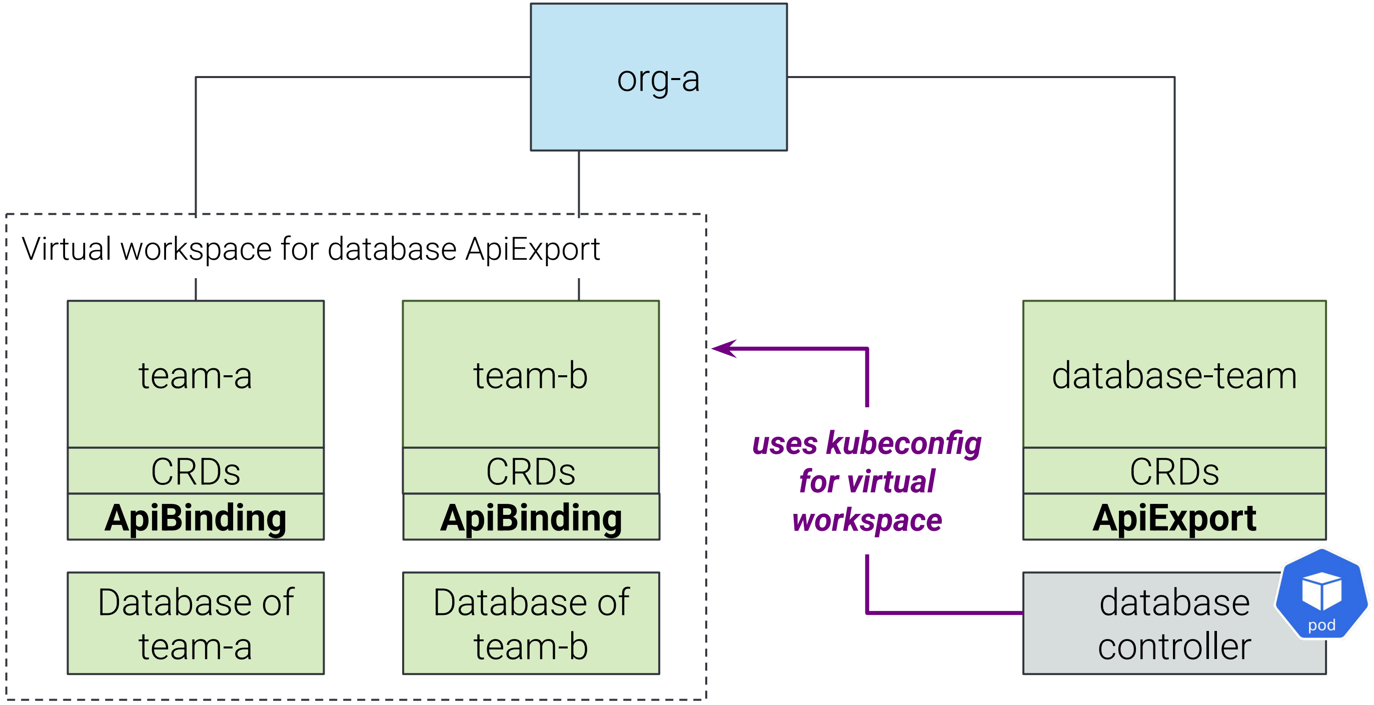
Keeping Track of Orders

# Actually the reality is even more complicated



## The APIExport Virtual Workspace to the Rescue

# A virtual workspace provides a computed view of parts of one or multiple workspaces



## APIExport Virtual Workspace - Good to know

- Virtual Workspaces are not a CustomRessource Type, you cannot directly create one, but kcp provides them for you
- APIExport Virtual Workspace provides a unique URL for its export, which can be used as the server URL in a k8s controller
- There are more virtual workspaces available in kcp

## Intro to Crossplane

## What is Crossplane

- Your cloud native **control plane**
    - Provision/manage **all** of your resources
- **Compose** those resources into high level **abstractions**
    - Give your developers self-service provisioning
- Kubernetes is a great control plane for containers
    - Crossplane teaches it how to manage **everything** else
- Cloud providers have used control planes for years
    - Now it’s your turn to build your own!

## Build your own

- Assemble granular resources. Crossplane has a marketplace with providers managing various resources.
- Expose as higher level self-service API for your app teams
  - **Compose** GKE, NodePool, Network, Subnetwork
  - **Offer** as a simple Cluster abstraction (API) with limited config for developers to self-service
- Hide infrastructure complexity and codify a “golden path”
- All with K8s API - compatible with kubectl, GitOps, etc.
- No code **required**

## Composite

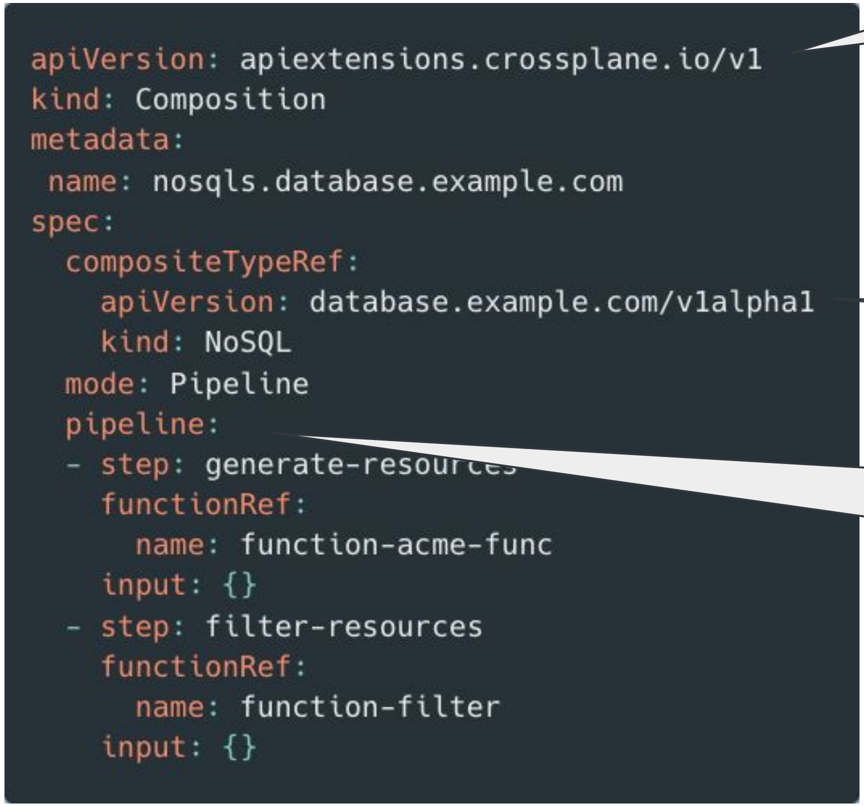
```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: nosqls.database.example.com
spec:
  group: database.example.com
  names:
    kind: NoSQL
    plural: nosqls
  versions:
  - name: v1alpha1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
```

First create Composite Resource Definition (XRD) to declare our custom platform API

API Group

Standard OpenAPI v3 schema

## Composition



```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
 name: nosqls.database.example.com
spec:
  compositeTypeRef:
    apiVersion: database.example.com/v1alpha1
    kind: NoSQL
  mode: Pipeline
  pipeline:
  - step: generate-resources
    functionRef:
      name: function-acme-func
    input: {}
  - step: filter-resources
    functionRef:
      name: function-filter
    input: {}
```

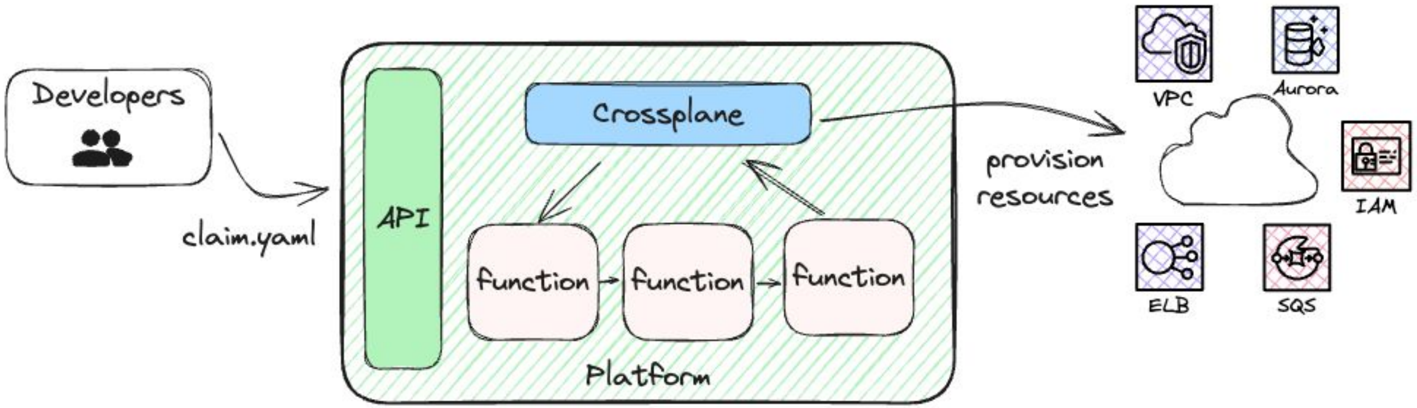
Then we define a Composition that defines the XRD

The XRD this Composition is for

Pipeline of functions to execute that weil generate the managed resources

## How do Functions work

- Run a pipeline of simple functions to compose resources
- Written in your language of choice
- Focus only on your unique logic
- Crossplane does the heavy lifting of resources CRUD, reconciling, finalizers, owner refs, etc.

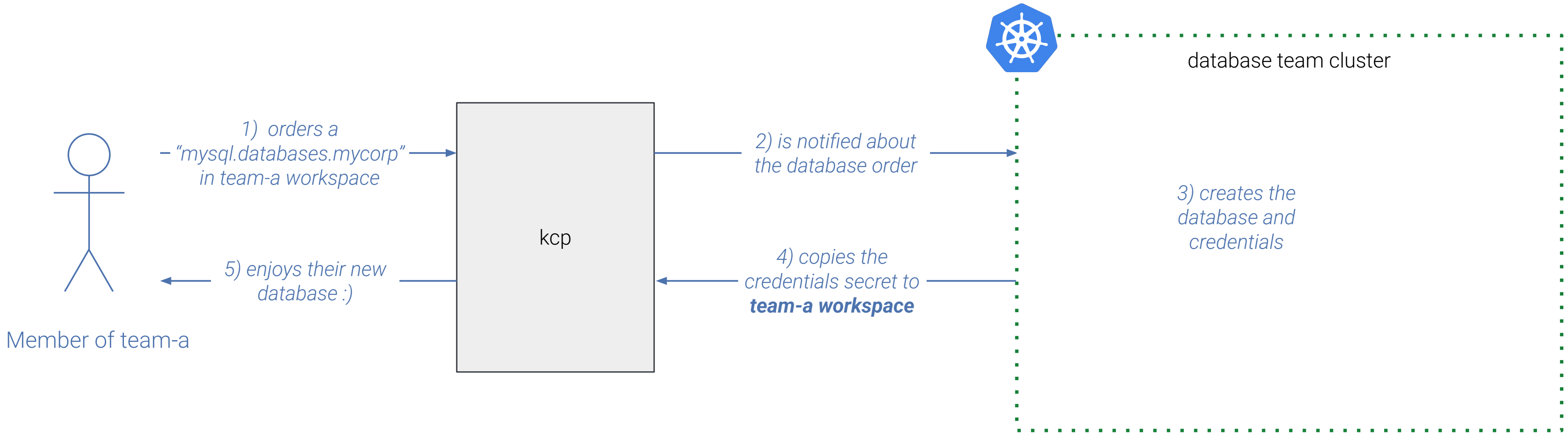


# Crossplane v2 is here

- **Crossplane v2 is more useful, more intuitive, and less opinionated**
- Three major changes:
  - Composite resources can now be namespaced
  - Managed resources are now namespaced
  - Composition supports any Kubernetes resource
- Operations - a new way to run operational tasks
- ManagedResourceDefinitions - a new way to control CRD sprawl
- Crossplane v2 is better suited to building control planes for **applications**, not just infrastructure

## Demo Time

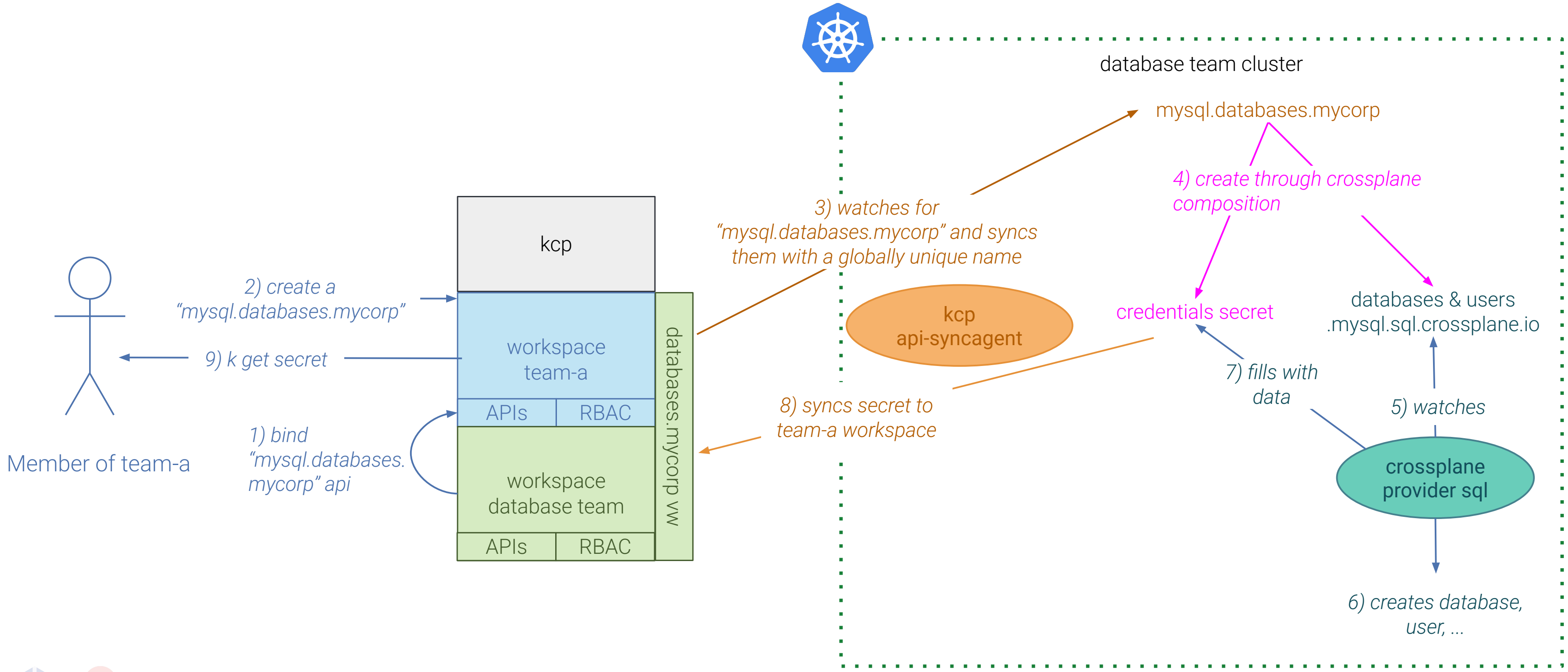
# What do we want to achieve? - High Level


database team cluster
1) orders a "mysql.databases.mycorp" in team-a workspace
kcp
2) is notified about the database order
3) creates the database and credentials
4) copies the credentials secret to **team-a workspace**
5) enjoys their new database :)
Member of team-a

## How can we achieve this?

+

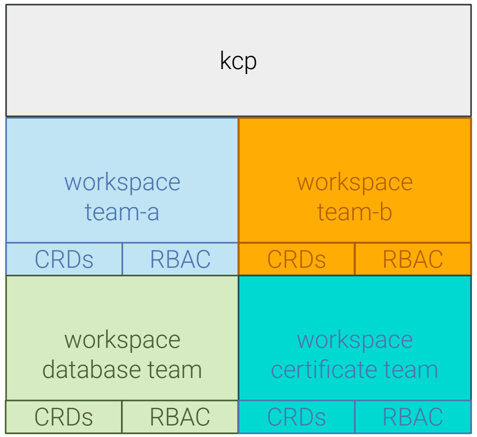
# Putting the Pieces Together

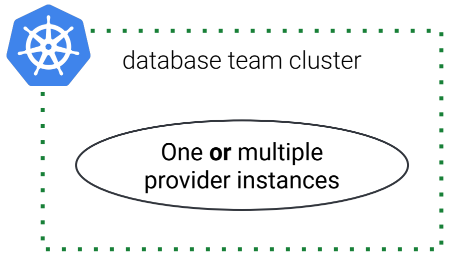


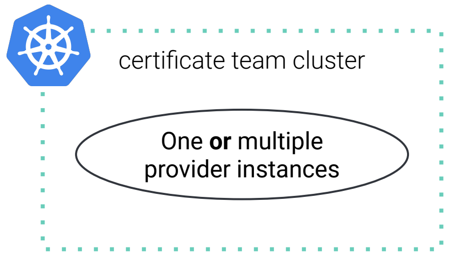
database team cluster
mysql.databases.mycorp
4) create through crossplane composition
3) watches for "mysql.databases.mycorp" and syncs them with a globally unique name
kcp
2) create a "mysql.databases.mycorp"
workspace team-a
9) k get secret
APIs
RBAC
1) bind "mysql.databases. mycorp" api
workspace database team
APIs
RBAC
databases.mycorp vw
Member of team-a
kcp api-syncagent
credentials secret
databases & users .mysql.sql.crossplane.io
7) fills with data
5) watches
8) syncs secret to team-a workspace
crossplane provider sql
6) creates database, user, ...

## Enough Slides, Let’s Hop In

# Why so complicated?
# Multi Tenancy Degrees of Freedom







## Multi Tenancy Degrees of Freedom

- Multiple consumer teams can have their own bound APIs via kcp workspaces
- Multiple providers can be consumed per workspace
- Multiple consumers can be served by a single crossplane provider instance

## One last thing about kcp

## So you have decided you want to try kcp out

```
$ kubectl ws .
Current workspace is 'root'.

$ kubectl get crd
No resources found



$ kubectl api-resources
NAME                SHORTNAMES  APIVERSION                 NAMESPACED    KIND
workspaces          ws          tenancy.kcp.io/v1alpha1    false         Workspace
workspacetypes                  tenancy.kcp.io/v1alpha1    false         WorkspaceType
databases                       databases.demo.example.com false         Database
...

$ kubectl explain workspaces
# will return workspaces api definition
```

## Wrapping Up

## Wrapping Up

https://crossplane.io

github.com/lsviben

linkedin.com/in/lovro-sviben

By using Crossplane v2 and kcp we can create true Multi Tenancy with multiple degrees of freedom

**kcp and crossplane are community projects! We welcome everyone to build the future together.**

https://kcp.io

*kcp-users & kcp-dev* on Kubernetes Slack

github.com/SimonTheLeg

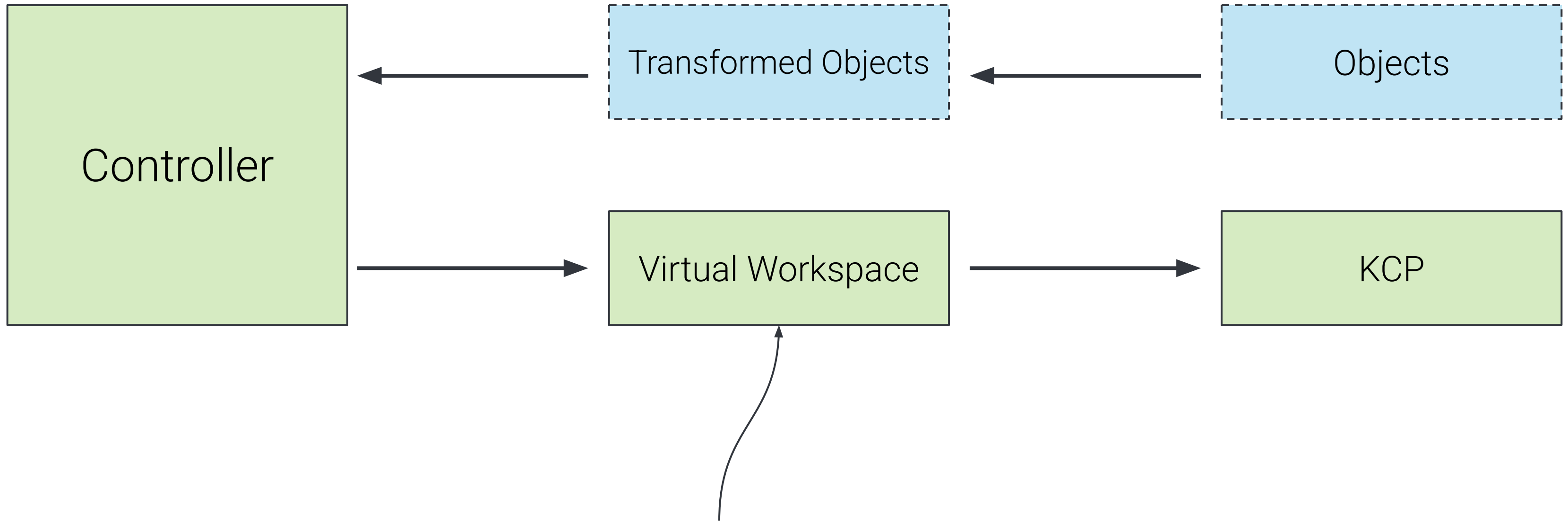
linkedin.com/in/simon-bein

## Backup: RBAC Extension for APIBindings

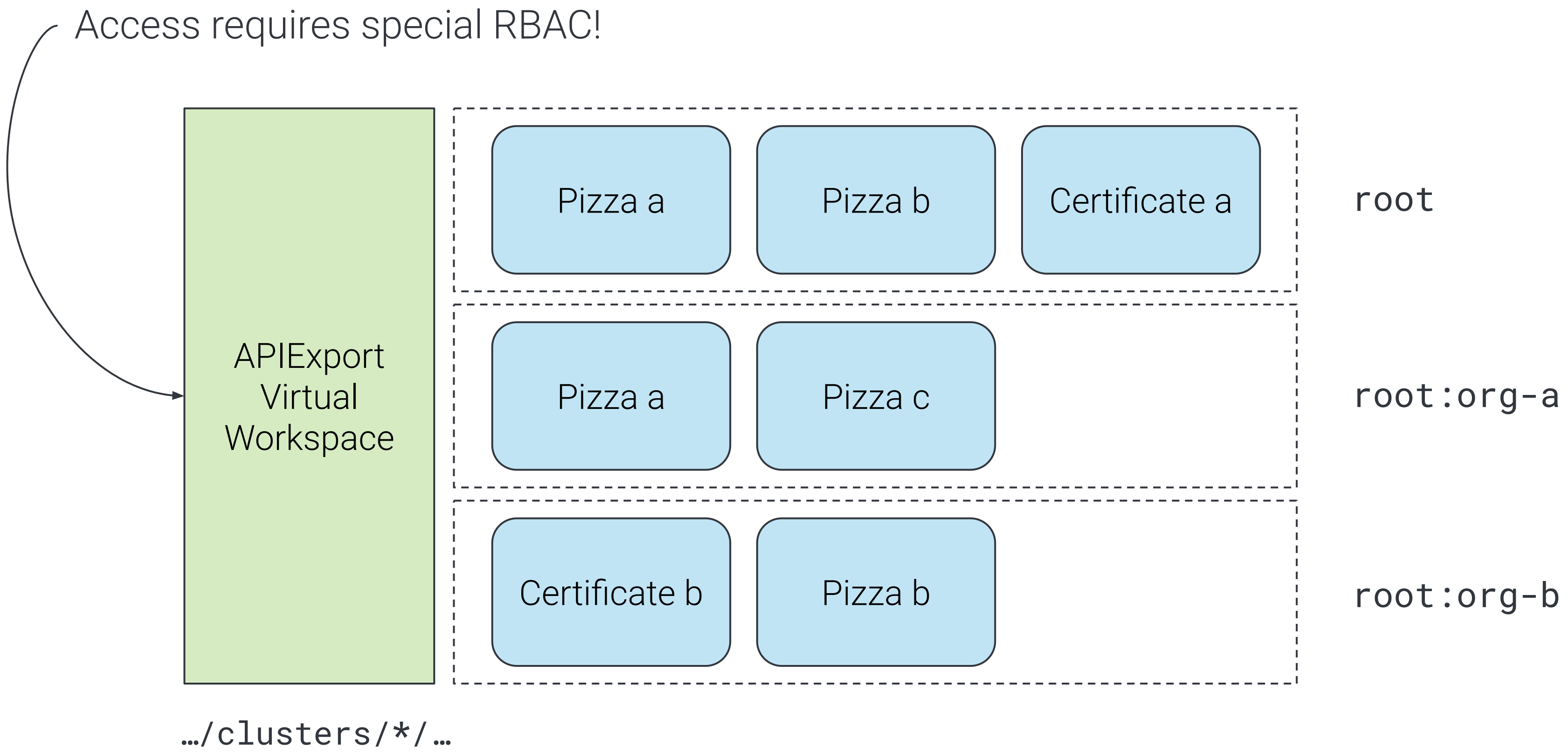
Binding to exported APIs requires RBAC permissions on the **APIExport**.

```
apiVersion:
rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
   name: bind-apiexport
rules:
- apiGroups:
  - apis.kcp.io
  resources:
  - apiexports
  verbs:
  - use
  resourceNames:
  - demo.embik.me
```

## Backup: Virtual Workspaces for Controllers



Proxy that provides a computed view



Access requires special RBAC!
APIExport Virtual Workspace
Pizza a
Pizza b
Certificate a
root
Pizza a
Pizza c
root:org-a
Certificate b
Pizza b
root:org-b
…/clusters/*/…

## Backup: How to Build a KCP-aware Controller

1 Use kcp-aware client and cache

```
MapperProvider: kcp.NewClusterAwareMapperProvider,
NewClient:      kcp.NewClusterAwareClient,
NewCache:       kcp.NewClusterAwareCache,
NewAPIReader:   kcp.NewClusterAwareAPIReader,
```

2 Reconcile in Virtual Workspace via **Cluster**

```
sigs.k8s.io/controller-runtime/pkg/cluster.Cluster
```

3 Reconcile with logical cluster in context

```
ctx = kontext.WithCluster(ctx, logicalcluster.Name(request.ClusterName))
```

github.com/kcp-dev/controller-runtime