Building an Enterprise Control Plane on Kubernetes

Daniel Mangum, upbound.io

Steven Borrelli, Mastercard

Goals

kubernetes

Crossplane

Explain why the controller pattern is the best way of managing infrastructure.

Demonstrate how to write an infrastructure controller using Crossplane and Kubebuilder.

Agenda

Introduction (~5 minutes)

Tutorial: Developing a Controller to Manage Github Orgs (65 minutes)

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Infrastructure Automation

What

Virtual Machines

Storage

DNS Records

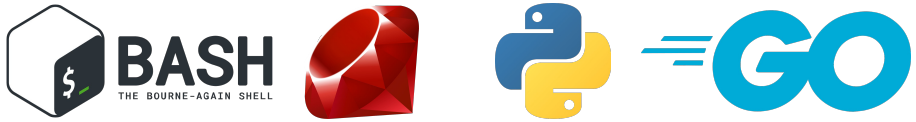
Load Balancers

Software

User Accounts

Firewall Rules

How



Typical Infrastructure Deployment

Spec

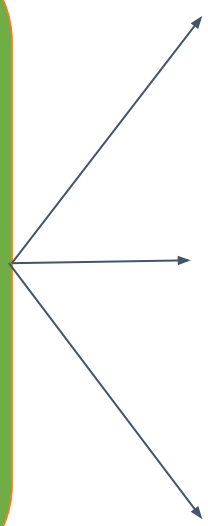
```
{
"type": "vm",
"count": 2,
"image": "debian",
"network","DMZ"
}
```

CI Pipeline

```
if branch =
"master" {
    LOGIN=$ecret
    run(curl vm.sh)
    run(vm.sh)
}
```

Logic

```
if exists(VM){
compareToDesired(VM,
Desired) }
else {
  createVM(VM)
  }
}
```



amazon web services

Microsoft Azure

vmware

f5

Problems

Spec

```
{
“type”: “vm”,
“count”: 2,
“image”: “debian”,
“network”,“DMZ”
}
```

Validation

API Support

Tooling

CI Pipeline

```
if branch =
“master” {
    LOGIN=$ecret
    run(curl vm.sh)
    run(vm.sh)
}
```

Logic in Pipelines

Fire and Forget

Based on GIt commits

Logic

```
if exists(VM){
compareToDesired(VM
, Desired) }
else {
  createVM(VM)
  }
}
```

Reconciliation

State Management

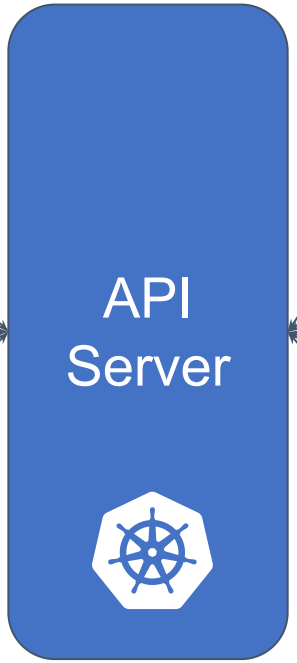
Operations/Logging/Monitoring

The Controller Approach


```
kind: SQLDatabase
metadata:
 name: web-backend
spec:
 region: eu-west
 version: 9
 memorySize: 10Gb
```

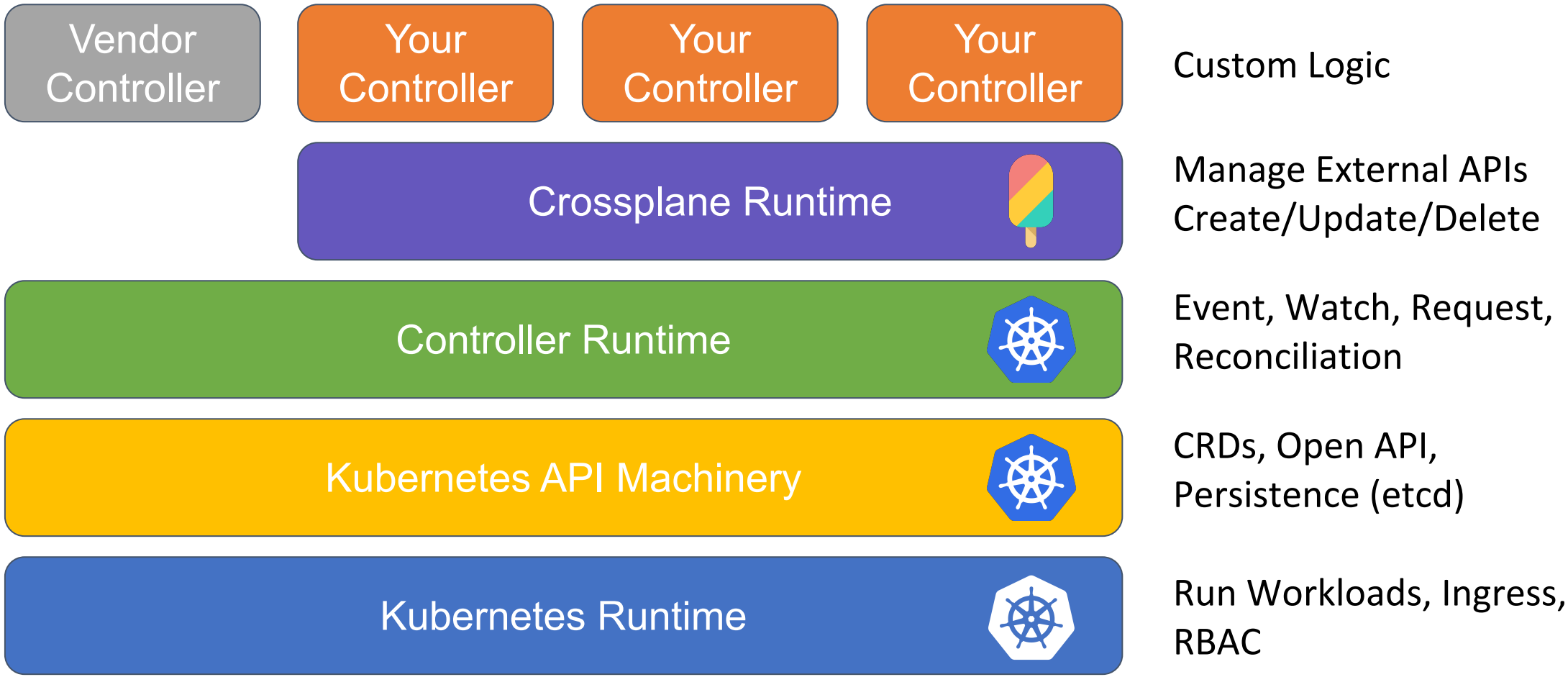
```
kind: StorageBucket
metadata:
 name: static-assets
spec:
 region: eu-west
 versioning: enabled
```

```
kind: GitHubTeam
metadata:
 name: webdev
spec:
 members: Harriet, Ida
 organization: Project
 Role: admin
```

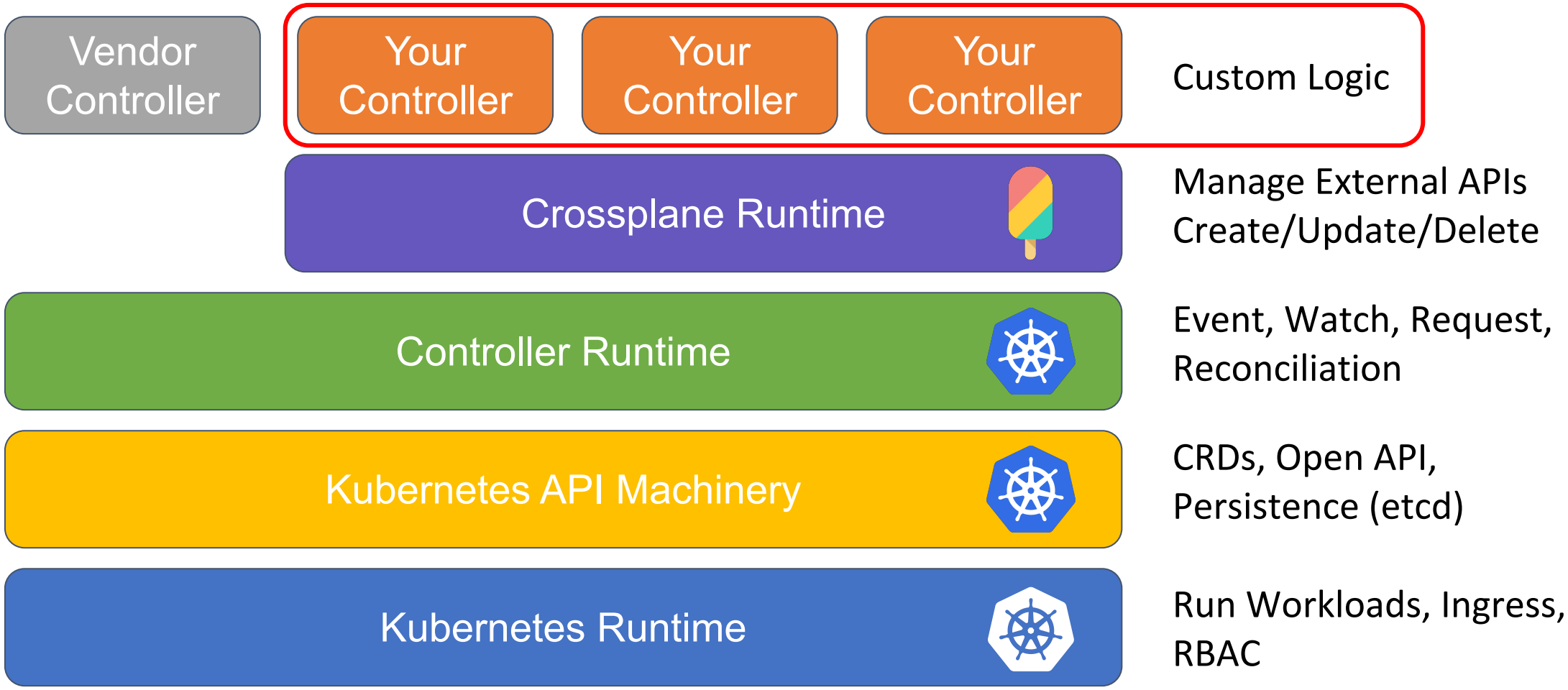


- Cloud Controller
- Network Controller
- Storage Controller
- Git Repo Controller

Foundations



Focus on the Solution



Ecosystem

Using Kubernetes as a platform means you can build upon a huge ecosystem

Open Policy Agent

Summary

- Massive Ecosystem Support
- CRDs let you expose infrastructure as Kubernetes objects
- Libraries let you build full-featured control planes

Controllers are the ideal platform for managing almost any infrastructure

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Links

Crossplane Docs:
https://crossplane.io/docs/v0.14/

Tutorial Code:
https://github.com/hasheddan/kc-provider-github

Kind:
https://github.com/kubernetes-sigs/kind

Tutorial Prerequisites

Install kind & kubectl according to the [crossplane installation documentation]

For example, on macOS:

```
$ brew upgrade
$ brew install kind
$ brew install kubectl
$ brew install helm
```

Create a cluster


```
$ kind create cluster

Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.19.1) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a nice day! 👋
```

Create the namespace

Set your context and create the `crossplane-system` namespace:

```
$ kubectl cluster-info --context kind-kind

Kubernetes master is running at https://127.0.0.1:60307
KubeDNS is running at
https://127.0.0.1:60307/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

```
$ kubectl create namespace crossplane-system

namespace "crossplane-system" created
```

Install Crossplane with Helm


```
$ helm repo add crossplane-alpha

https://charts.crossplane.io/alpha
"crossplane-alpha" has been added to your repositories

$ helm install crossplane --namespace crossplane-system
crossplane-alpha/crossplane --set alpha.oam.enabled=true

NAME: crossplane
LAST DEPLOYED: Wed Nov 18 13:46:50 2020
NAMESPACE: crossplane-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Release: crossplane
```

Demo: AWS DB

This is not required for the tutorial (as it requires AWS credentials), but provides a walkthrough on how to manage resources via Crossplane.

First, set up a provider and create a Postgresql database in AWS. We will be following the instructions at:

https://crossplane.github.io/docs/v0.14/getting-started/install-configure.html#get-aws-account-keyfile

Demo: AWS DB

Install the crossplane CLI and install the AWS provider

```
$ curl -sL
https://raw.githubusercontent.com/crossplane/crossplane/
release-0.14/install.sh | sh

$ kubectl-crossplane install provider
crossplane/provider-aws:alpha
provider.pkg.crossplane.io/provider-aws created
```

Demo: AWS DB

The `ProviderConfig` holds connection information (endpoints, secrets) for a remote API. The credentials are stored in a kubernetes secret, and accessed via a `secretRef`

```yaml
apiVersion: aws.crossplane.io/v1beta1
kind: ProviderConfig
metadata:
  name: default
spec:
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: aws-creds
       key: key
```

The provider could have other Spec Parameters: define the API endpoints, or manage TLS certificate validation.

API's can be authenticated in many ways: username & passwords, tokens, x509 certificates, or IAM.

When writing a provider you will have the ability to fetch the secret data from k8s to use with an API client.

Demo: AWS DB

Crossplane resources are k8s objects. The resource configuration is set in the `forProvider` section.

```
apiVersion: database.aws.crossplane.io/v1beta1
kind: RDSInstance
metadata:
  name: rdspostgresql
spec:
  forProvider:
    dbInstanceClass: db.t2.small
    masterUsername: masteruser
    allocatedStorage: 20
    engine: postgres
    engineVersion: "9.6"
    skipFinalSnapshotBeforeDeletion: true
  writeConnectionSecretToRef:
    namespace: crossplane-system
    name: aws-rdspostgresql-conn
```

Infrastructure CRDs are full K8s objects and support Versions, Labels and Metadata.

One of the first tasks when writing a Provider is defining a Spec. We will cover this in the Tutorial.

If the new resource creates credentials, we can write these back to a k8s secret.

Tutorial

Today we will be building a controller to manage Github Teams.

Clone and build the controller

Clone the source for our Github controller and build it (make sure you have Go installed). The first build may take a while as all the dependencies are downloaded:

```
$ git clone https://github.com/hasheddan/kc-provider-github

$ cd kc-provider-github

$ make
```

To build the controller, install the CRDs, and run the controller, type:

```
$ make run
```

Source Code Overview

`apis/`
`  v1alpha1/`
`  v1beta1/`
	Go types: CRD definitions, Crossplane managed resource definitions, Kubebuilder annotations

`cluster/`	Dockerfiles to package controllers
`cmd/` `  provider/`	Controller main.go
`hack/`	License text to apply to generated code
`package/` `  crds/`	Generated CRD .yaml files to deploy to a cluster

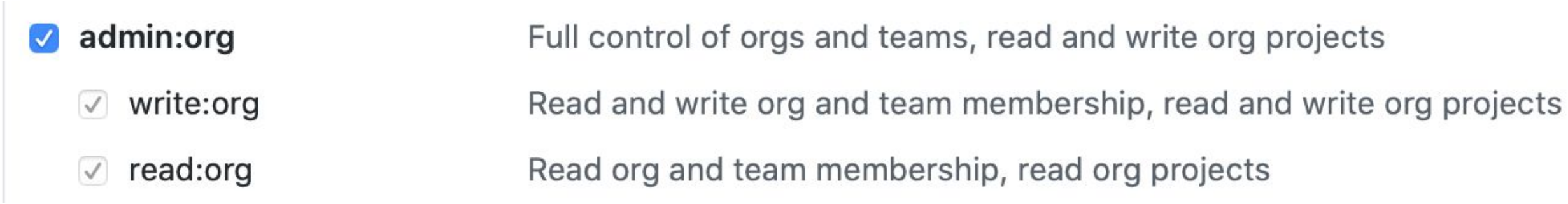
`pkg/`
`  client/`
`  controller/`
	`client/` Communicate with remote API, calculate diffs
	
	`controller/` Reconciliation, Observe/Create/Update/Delete

Create a Github Token

We'll need a token to talk to the Github API. Use the instructions at:

https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/creating-a-personal-access-token

Ensure that **admin:org** scopes are set on the token.



When the scopes are selected, click: Generate token

Create a Kubernetes Secret

We need to store our Github token as a Kubernetes secret for our Provider to use. Our secret will have a key of `credentials`.

Save the token into a file like `token.txt` and create the secret:

```
$ kubectl create secret generic example-provider-secret \
  -n crossplane-system --from-file=credentials=token.txt

secret "example-provider-secret" created
```

Define The ProviderConfig

The ProviderConfig will refer to the secret we just created (make sure that namespace, name, and key match). The tutorial code includes a sample in `examples/provider/config.yaml`:

```
apiVersion: template.crossplane.io/v1alpha1
kind: ProviderConfig
metadata:
 name: default
spec:
 credentials:
   source: Secret
   secretRef:
     namespace: crossplane-system
     name: example-provider-secret
     key: credentials
```

```
$ kubectl apply -f examples/provider/config.yaml
```

Create a Github Team

Our `Team` resource allows us to create a Github team with a name, description and privacy settings. The `examples/org` directory of the tutorial has an example::

```
apiVersion: org.github.hasheddan.io/v1alpha1
kind: Team
metadata:
    name: was-crossplane
spec:
    forProvider:
        org: kubecon-na
        description: "our description"
        privacy: secret
```

```
$ kubectl apply -f examples/org/team.yaml
team.org.github.hasheddan.io/was-crossplane
```

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Custom Resource Definitions

A [Custom Resource Definition](#) (CRD) is a way to extend the Kubernetes API.

We can create any object that is managed like a Kubernetes Pod or Deployment. For example, when we create a `Team` object for Github teams, we could manage it using `kubectl`:

```
$ kubectl get team
NAME                SYNCED      AGE
dbadmin             True        7d
webdev              True        30d
```

Because CRDs are Kubernetes objects, they support OpenAPI V3 validation and can be managed by any tool that supports Kubernetes.

Custom Resource Definitions

CRDs are defined using .yaml files, and are applied to the cluster using kubectl.

In this tutorial, we automatically generate our CRD files, but the CRD definition and controller implementation are independent of one another.

To list the CRDs applied to a cluster using the following command:

```
$ kubectl get crds

NAME                                          CREATED AT
memberships.org.github.hasheddan.io           2020-11-19T23:48:26Z
teams.org.github.hasheddan.io                 2020-11-19T23:48:26Z
providerconfigs.github.hasheddan.io           2020-11-18T23:42:54Z
providerconfigusages.github.hasheddan.io      2020-11-18T23:43:54Z
```

Parameters and Observations

Types are defined in [apis/org/v1alpha1/types.go](). Parameters and Observations are features of the [crossplane-runtime]() library.

```
// Parameters are the desired settings for the resources (i.e. Team name)
type MyTypeParameters struct {
    ConfigurableField string `json:"configurableField"`
}

// Observations are the observed state of the resource (i.e., Team UUID)
type Observation struct {
    ObservableField string `json:"observableField,omitempty"`
}
```

Defining Team Parameters

For our Github team, we want to define the name, description and privacy settings:

```
// TeamParameters are settings we can configure on a Github Team
type TeamParameters struct {
    // Org is the Github organization
    Org string `json:"org"`
    // Description of the team
    Description *string `json:"description,omitempty"`
    // Privacy settings for the team, can be secret or closed
    // Kubebuilder annotations can enforce this in the CRD
    //+kubebuilder:validation:Enum=secret;closed
    Privacy *string `json:"privacy,omitempty"`
}
```

Defining Team Observation

We need to set up what fields we want to observe from the remote API.

We'll use the values that are returned to compare against our desired state.

```
// TeamObservation are observed settings of the team
// There are many fields returned from the Github API,
// see https://developer.github.com/v3/teams/
type TeamObservation struct {
    //NodeID the Github team NodeID
    NodeID string `json:"node_id,omitempty"`
}
```

Spec and Status

The Spec and Status structs are found in `apis/org/v1alpha1/types.go`.

```
// TeamSpec defines the desired state of a Team.
// Includes Metadata and TeamParameters for the provider
type TeamSpec struct {
    runtimev1alpha1.ResourceSpec `json:",inline"`
    ForProvider                  TeamParameters `json:"forProvider"`
}
// TeamStatus represents the observed state of a Team.
// Includes metadata and the TeamObservation Struct
type TeamStatus struct {
    runtimev1alpha1.ResourceStatus `json:",inline"`
    AtProvider                     TeamObservation `json:"atProvider,omitempty"`
}
```

Groups, Versions & Kinds

When you start working with the Kubernetes API, you will come across Groups, Versions and Kinds (often abbreviated GVK in source code).

The **Group** is a set of related resources. It resembles a DNS name. In this case the group will be GIthub Organization objects.

Versions: alpha, beta, stable (v1). See API Versioning.

```
apiVersion: org.github.hasheddan.io/v1alpha1
kind: Team
metadata:
    name: was-crossplane
spec:
    forProvider: ...
```

Kind is the name of a resource that has specific attributes and properties. Once created, the system will work to ensure the resource exists.

Generating CRDs

The file `apis/generate.go` contains code to automatically generate CRD yaml files, DeepCopy methods and Crossplane go files (`zz_generated*`) for managing resources (like the ability to write secrets from a resource). This is done by using controller-gen and angry-jet.

The `generate` target in the `Makefile` will perform the generation of the CRD manifests and `zz_` files.

```
// Generate deepcopy methodsets and CRD manifests
//go:generate go run -tags generate sigs.k8s.io/controller-tools/cmd/controller-gen
object:headerFile=../hack/boilerplate.go.txt paths=./... crd:trivialVersions=true
output:artifacts:config=../package/crds

// Generate crossplane-runtime methodsets (resource.Claim, etc)
//go:generate go run -tags generate github.com/crossplane/crossplane-tools/cmd/angryjet
generate-methodsets --header-file=../hack/boilerplate.go.txt ./...
```

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Github API Client

Client code is defined in [pkg/client/client.go](). We are using the [Go-Github]() library to communicate with the API.

The token is supplied from the `secretRef` in the Provider.

```
// NewClient creates a new client
func NewClient(token string) (*github.Client, error) {
    if token == "" {
        return nil, errors.New(errEmptyToken)
    }
    ctx := context.Background()
    ts := oauth2.StaticTokenSource(
        &oauth2.Token{AccessToken: token},
    )
    tc := oauth2.NewClient(ctx, ts)

    return github.NewClient(tc), nil
}
```

Mapping the API to K8s

In more complex controllers the `pkg/client` directory often includes code like `IsUpToDate()` to compare desired vs. existing state.

This example is for the GCP CloudMemorystore:

```
func IsUpToDate(id InstanceID, in *v1beta1.CloudMemorystoreInstanceParameters, observed *redisv1pb.Instance) (bool, error)
{
	generated, err := copystructure.Copy(observed)
	if err != nil { return true, errors.Wrap(err, errCheckUpToDate)}

	desired, ok := generated.(*redisv1pb.Instance)
	if !ok {return true, errors.New(errCheckUpToDate) }

	GenerateRedisInstance(id, *in, desired)

	if desired.MemorySizeGb != observed.MemorySizeGb { return false, nil }
	if !cmp.Equal(desired.RedisConfigs, observed.RedisConfigs) { return false, nil }
	return true, nil
}
```

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

main()

The entrypoint for the controller is [cmd/provider/main.go](), and is built upon [controller-runtime]() libraries.

One Manager can manage multiple controllers and provide them with a shared runtime.

Our `main.go` is fairly short, but performs many functions:

`ctrl.NewManager()`	Create a shared controller manager (cache, leader election, etc)
`apis.AddToScheme()`	Register the API Types we defined in `apis/`
`controller.Setup()`	Set up the controllers defined in `pkg/controller/`
`mgr.Start()`	Start the controllers

Controller

Our [Github Team controller] runs continually, performing the following:

- `Setup()` a `NewReconciler()` to watch the Kubernetes API for any changes to `Team` objects
- `Connect()` to the Github API using the the `Provider` secret
- For every `Team` object:
  - `Observe()` the state of the requested object on the remote API
  - `Create()` resources if they don't exist
  - `Update()` resources if desired state doesn't match observed state
  - `Delete()` resources if the Kubernetes object has been deleted

In the following slides, we will describe each method.

Controller Setup()

Each Controller has a `Setup()` that creates a controller that is added to the shared Manager.

Setting up a controller involves the following:

`NewReconciler()`	Reconcile the API Group/Version/Kind we want to manage
`WithExternalConnecter()`	The Remote API client to use
`WithLogger()/WithRecorder()`	Set up logging and events
`NewControllerManagedBy()`	Add the controller to the manager

Controller Setup()


```
func Setup(mgr ctrl.Manager, l logging.Logger) error {
    name := managed.ControllerName(v1alpha1.TeamGroupKind)

    r := managed.NewReconciler(mgr,
        resource.ManagedKind(v1alpha1.TeamGroupVersionKind),
        managed.WithExternalConnecter(&connector{
            kube:  mgr.GetClient(),
            usage: resource.NewProviderConfigUsageTracker(mgr.GetClient(),
&apisv1alpha1.ProviderConfigUsage{})}),
        managed.WithLogger(l.WithValues("controller", name)),
        managed.WithRecorder(event.NewAPIRecorder(mgr.GetEventRecorderFor(name))))

    return ctrl.NewControllerManagedBy(mgr).
        Named(name).
        For(&v1alpha1.Team{}).
        Complete(r)
}
```

Reconcile our Team Kind, with an ExternalConnector

Add our Controller to the Manager

Controller Connect()

The `Connect()` method takes the Provider configuration and returns an API client by performing the following:

- Connect to the Kubernetes API to get the `Provider` using `kube.Get()`
  - Read the Provider configuration (API endpoint, TLS settings, `secretRef`)
  - Fetch the credential data from the provided `secretRef` using `kube.Get()`
  - Use the provider information and credentials to create an API client

Controller Connect()


```
func (c *connector) Connect(ctx context.Context, mg resource.Managed) (managed.ExternalClient,  error) {

    pc := &apisv1alpha1.ProviderConfig{}
    if err := c.kube.Get(ctx, types.NamespacedName{Name: cr. GetProviderConfigReference().Name}, pc); err !=
nil {
        return nil, errors.Wrap(err, errGetPC)
    }

    ref := pc.Spec.Credentials.SecretRef

    s := &v1.Secret{}
    if err := c.kube.Get(ctx, types.NamespacedName{Namespace: ref.Namespace, Name: ref.Name}, s); err !=
nil {
        return nil, errors.Wrap(err, errGetSecret)
    }

    svc, err := gitclient.NewClient(string(s.Data[ref.Key]))
    if err != nil {
        return nil, errors.Wrap(err, errNewClient)
    }

    return &external{service: svc}, nil
}
```

Get Provider configuration that you defined in `apis/`

Get secret from the `secretRef`

Create a client using token stored in secret

Controller Observe()

The `Observe()` method uses the Remote API client to observe the state of the resource.

Most importantly it returns an ExternalObservation{}, which the Crossplane runtime uses to decide what actions to perform next, such as calling `Create()` or `Update()`.

```
managed.ExternalObservation{
        // Return false when the external resource does not exist. This lets
        // the managed resource reconciler know that it needs to call Create to
        // (re)create the resource, or that it has successfully been deleted.
        ResourceExists: true,

        // Return false when the external resource exists, but it not up to date
        // with the desired managed resource state. This lets the managed
        // resource reconciler know that it needs to call Update.
        ResourceUpToDate: upToDate,
}
```

Controller Observe()


```
func (c *external) Observe(ctx context.Context, mg resource.Managed) (managed.ExternalObservation,  error)
{
    team, _, err := c.service.Teams.GetTeamBySlug(ctx, cr.Spec.ForProvider.Org, meta.GetExternalName(cr))
    if err != nil {
        return managed.ExternalObservation{
            ResourceExists: false,
        }, nil
    }

    upToDate := true
    if team != nil {
        if cr.Spec.ForProvider.Description != nil {
            if team.Description == nil || *team.Description != *cr.Spec.ForProvider.Description {
                upToDate = false
            }
        }
    }
    return managed.ExternalObservation{
        ResourceExists: true,
        ResourceUpToDate: upToDate,
    }, nil
}
```

If resource doesn't exist, return
`ResourceExists: false`

If desired state != observed state, set
`ResourceUpToDate: false`

Observe returns
`ExternalObservation{}`

Controller Create()

The `Create()` method is called if `ExternalObservation{ResourceExists: false}`

It returns an ExternalCreation{}, which could contain newly generated credentials or other connection details that can be propagated to the users who requested the new resource.

If `Create()` returns an error, the controller will reschedule the resource creation to be attempted again in the future. Developers can use the built-in wait durations or define their own wait times to reattempt creation.

```
type ExternalCreation struct {
    // ConnectionDetails required to connect to this resource.
    // Crossplane may publish these credentials to a store (e.g. a Secret).
    ConnectionDetails ConnectionDetails
}
```

Controller Create()


```
func (c *external) Create(ctx context.Context, mg resource.Managed) (managed.ExternalCreation,
error) {
    cr, ok := mg.(*v1alpha1.Team)
    if !ok {
        return managed.ExternalCreation{}, errors.New()
    }

    fmt.Printf("Creating: %+v", cr)

     ,  , err := c.service.Teams.CreateTeam(ctx, cr.Spec.ForProvider.Org, github.NewTeam{
        Name:        meta.GetExternalName(cr),
        Description: cr.Spec.ForProvider.Description,
        Privacy:     cr.Spec.ForProvider.Privacy,
    })

    return managed.ExternalCreation{}, err
}
```

Use the Github API client's `CreateTeam()` to create the team

Get the desired settings from the `ForProvider` section of the Team yaml

Return `ExternalCreation{}`. This could contain Database credentials from the remote API}

Controller Update()

The `Update()` method is called if `ExternalObservation{ResourceIsUpToDate: false}`

`Update()` is similar to the `Create()` method and returns an ExternalUpdate{}, which contains `ConnectionDetails`.

If `Update()` returns an error, the controller will reschedule the resource creation to be attempted again in the future.

```
type ExternalUpdate struct {
    // ConnectionDetails required to connect to this resource.
    ConnectionDetails ConnectionDetails
}
```

Controller Update()


```
func (c *external) Update(ctx context.Context, mg resource.Managed) (managed.ExternalUpdate,  error) {
    cr, ok := mg.(*v1alpha1.Team)
    if !ok {
        return managed.ExternalUpdate{}, errors. New(errNotMyType)
    }

    fmt.Printf("Updating: %+v", cr)

    _, _, err := c.service.Teams. EditTeamBySlug(ctx, cr.Spec.ForProvider.Org, meta. GetExternalName(cr),
github.NewTeam{
        Name:         meta. GetExternalName(cr),
        Description: cr.Spec.ForProvider.Description,
        Privacy:      cr.Spec.ForProvider.Privacy,
    }, false)

    return managed.ExternalUpdate{}, err
}
```

Use `EditTeamBySlug()` to update the team

Get the desired settings from the `ForProvider` section of the Team yaml

Return `ExternalUpdate{}`, which is similar to the `ExternalCreation{}` struct.

Controller Delete()

The `Delete()` method is called if there is a request to delete the Kubernetes object.

An important concept when writing Kubernetes controllers is managing finalizers.

When a delete request comes in for an object with a finalizer, the `metadata.deletionTimestamp` field is set. It is the responsibility of each controller to delete their finalizer once they have finished deleting their resource.

Crossplane-runtime automatically manages finalizers during Reconciliation, so you won't need to do anything to your code.

Controller Delete()


```
func (c *external) Delete(ctx context.Context, mg resource.Managed) error {
    cr, ok := mg.(*v1alpha1.Team)
    if !ok {
        return errors.New(errNotMyType)
    }

    fmt.Printf("Deleting: %+v", cr)

     , err := c.service.Teams.DeleteTeamBySlug(ctx, cr.Spec.ForProvider.Org,
meta.GetExternalName(cr))

    return err
}
```

Use `DeleteTeamBySlug()` to delete the team

Unlike the other methods, we only return errors

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Packaging the Controller

Crossplane packages are OCI-compliant images that can be stored in a registry that supports the Docker Registry v2 API.

The package definition is located in `package/crossplane.yaml`, and can be built using the crossplane kubectl plugin :

```
cd package
kubectl crossplane build provider
```

This will generate an `.xpkg` package that can be pushed to a registry along with the controller docker image built using `make image`.

One the image and package are pushed, you can install the controller onto any cluster using `kubectl crossplane install provider`.

Update the Controller

To publish a new controller, update the Docker image tags in the Makefile, then run:

```
make image

make image-push
```

We then rebuild our package definition in `package/crossplane.yaml`

```
kubectl crossplane build provider

kubectl crossplane push provider <provider>
```

Agenda

Introduction

Tutorial: Developing a Crossplane Controller to Manage Github Orgs

- Defining your Custom Resource Definitions (CRDs)
- Creating a client for the remote API
- Writing a controller for CRUD operations
- Packaging and shipping your controller
- Higher level abstractions: composing apps and infrastructure

Q&A

Composition

Composition is a Crossplane feature that allows you to build new custom resources upon other CRDs.

A Composite Resource Definition (XRD) can be used to combine any number of Kubernetes CRDs to build a service catalog.

We are going to demonstrate how to compose Github Teams and Memberships.

Composition

In our composition.yaml, we define a `CompositeUserTeam` that has a list of resources, including our `Team` and `Membership` CRDs:

```
apiVersion: apiextensions.crossplane.io/v1alpha1
kind: Composition
metadata:
 name: compositeuserteams.github.hasheddan.io
spec:
 compositeTypeRef:
   apiVersion: hasheddan.io/v1alpha1
   kind: CompositeUserTeam
 resources:
   - base:
       apiVersion: org.github.hasheddan.io/v1alpha1
       kind: Team
       spec:
         forProvider:
           org: kubecon-na
           description: "A composed team."
           privacy: secret
...
```

Composition

In a composition, XRD fields can be mapped to the underlying CRD specification using patches.

For example, mapping the `CompositeUserTeam` organization to the `Team forProvider` organization.

```
patches:
 - fromFieldPath: "metadata.annotations[crossplane.io/external-name]"
   toFieldPath: "spec.forProvider.team"
 - fromFieldPath: "spec.org"
   toFieldPath: "spec.forProvider.org"
 - fromFieldPath: "spec.user"
   toFieldPath: "spec.forProvider.user"
```

Composition

Compositions are built like Crossplane packages

```
kubectl crossplane build configuration
```

```
kubectl crossplane push configuration <configuration>
```

When installed to a cluster, compositions can be retrieved:

```
kubectl get xrd
```

Q&A

https://slack.crossplane.io/


KEEP CLOUD NATIVE
EVERYWHERE
KubeCon
CloudNativeCon
North America 2020
Virtual