Building Applications using Kubernetes APIs

Interacting with Kubernetes APIs to deploy, observe, and manage applications, and building Kubernetes operators to manage all of the things.

What is Kubernetes?

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

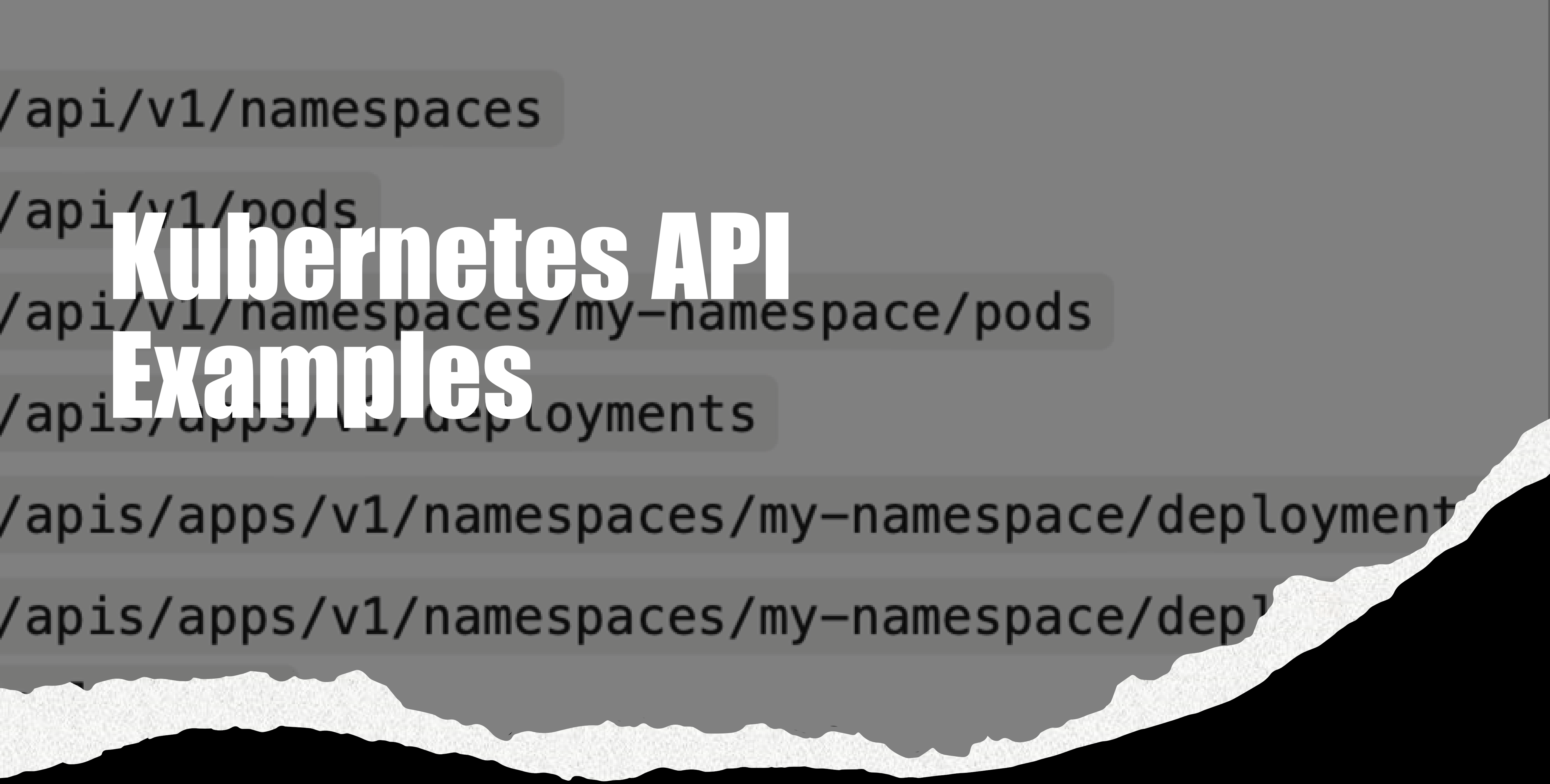
Really... what is Kubernetes?

Kubernetes is a bunch of services working together to make it easier for folks to build distributed applications.

Kubernetes API Server

- Also known as the Kubernetes control plane.
- Aggregation of HTTP APIs that expose typed resources described using OpenAPI v3.
- Admission webhooks for validation.
- Mutating and defaulting webhooks.
- Handles version negotiation and translation.
- Provides watch functionality for when a resource changes.

Kubernetes API Examples



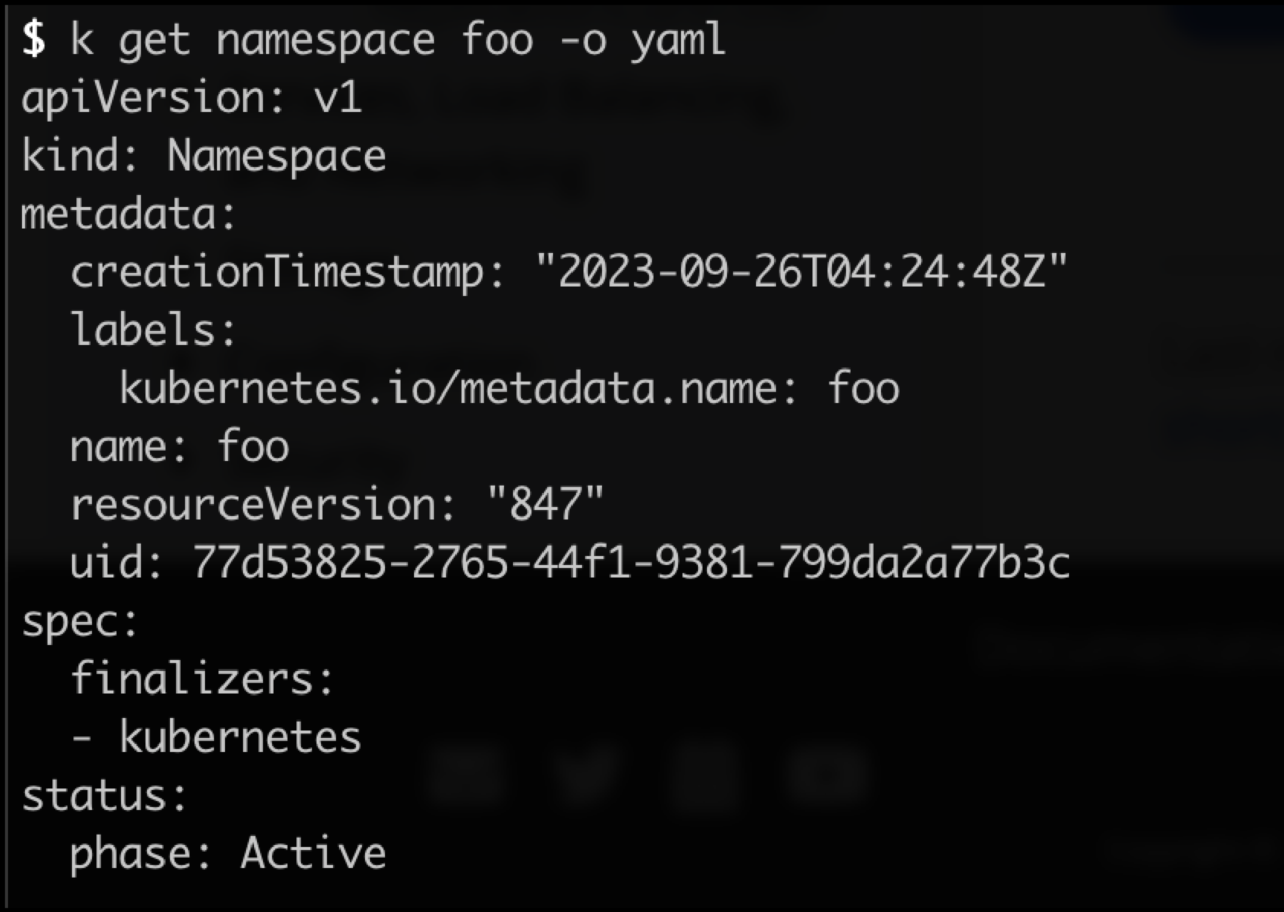
```
/api/v1/namespaces
/api/v1/pods
/api/v1/namespaces/my-namespace/pods
/apis/apps/v1/deployments
/apis/apps/v1/namespaces/my-namespace/deployment
/apis/apps/v1/namespaces/my-namespace/depl
```

Kubernetes Resources

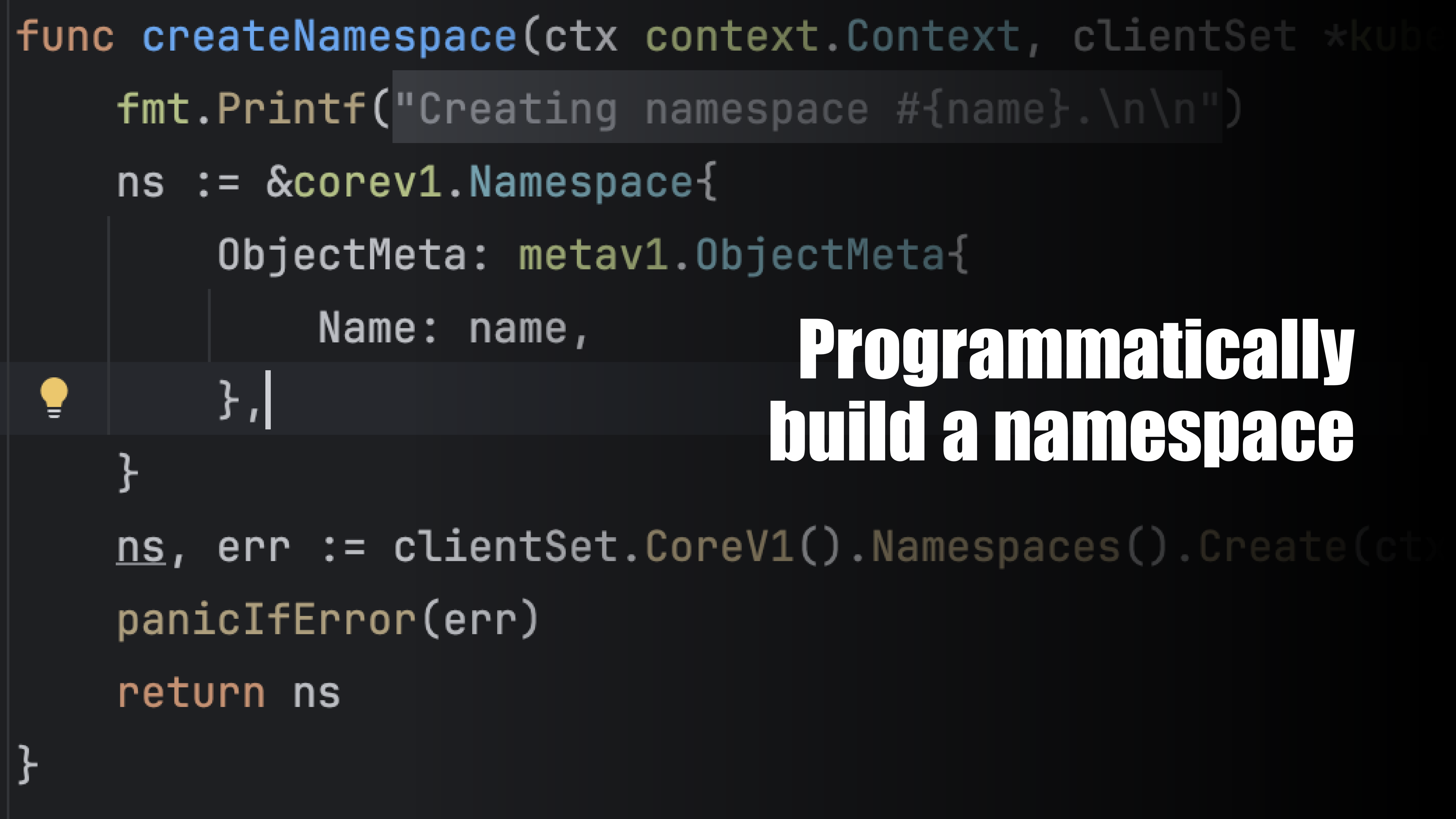
A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state.

Anatomy of a Kubernetes Resource

- Group, Version, Kind (GVK) + Namespace, Name
- Spec field that describes the desired state of the resource.
- Status field that describes the current state of the resource.
- Metadata
  - Annotations: key / value maps that can’t be used for querying
  - Labels: key / value maps that can be used for object queries



```
$ k get namespace foo -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2023-09-26T04:24:48Z"
  labels:
    kubernetes.io/metadata.name: foo
  name: foo
  resourceVersion: "847"
  uid: 77d53825-2765-44f1-9381-799da2a77b3c
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```



```go
func createNamespace(ctx context.Context, clientSet *kub
    fmt.Printf("Creating namespace #{name}.\n\n")
    ns := &corev1.Namespace{
        ObjectMeta: metav1.ObjectMeta{
            Name: name,
        },
    }
    ns, err := clientSet.CoreV1().Namespaces().Create(ct
    panicIfError(err)
    return ns
}
```

Programmatically build a namespace

Hands on programmatically deploying a load-balanced app on Kubernetes

- Build a Go application to use the Kubernetes APIs to deploy a simple NGINX ”Hello world!” application.
- Use an Ingress, Service, and Deployment resources to create your service stack.
- Follow the logs from each running instance of NGINX.
- Stretch Goals:
  - Use additional Kubernetes APIs to extend the demo.
  - Deploy a different application image rather than the NGINX demo.

Extending the Kubernetes API

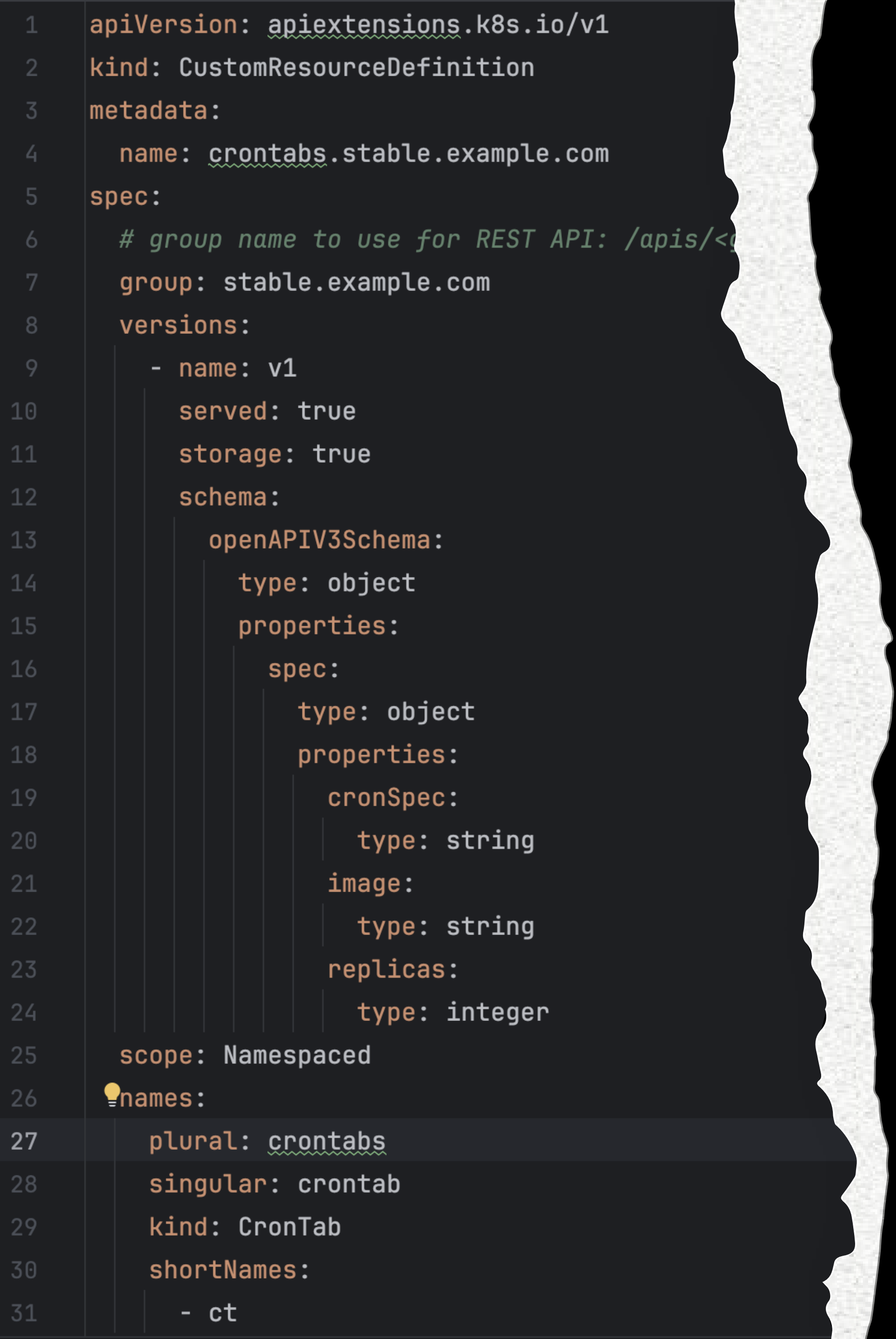
- In the previous hands on experience we learned that Kubernetes is not just a single API, but an aggregation of APIs.
- In this next section we are going to learn how to create our own resources.

Operator Pattern

The operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

Operator Pattern

- Custom Resource Definitions (CRDs)
- Custom Controllers



```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
```

Custom Resource Definitions (CRDs)

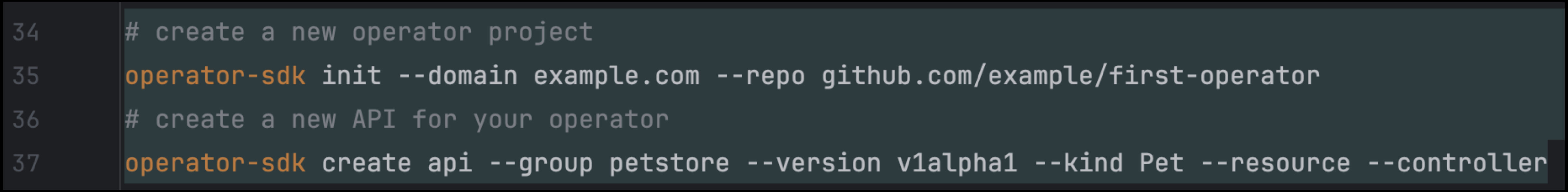
Controllers

- In robotics and automation, a control loop is a non-terminating loop that regulates the state of a system.
- A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

This sounds complex...

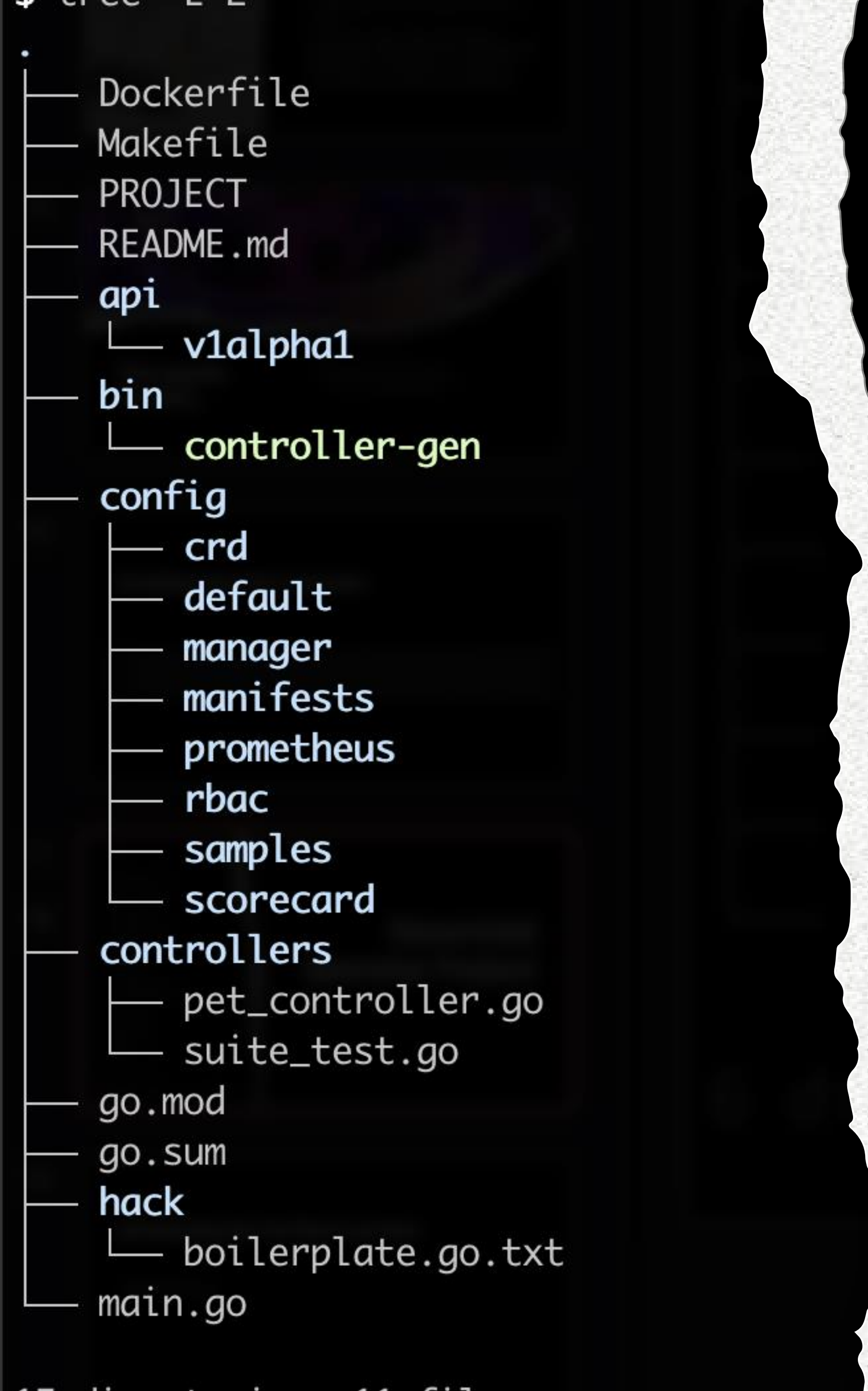
- There are great tools to help build your operators.
  - Operator SDK: https://sdk.operatorframework.io
  - Kubebuilder: https://book.kubebuilder.io

Creating your first operator



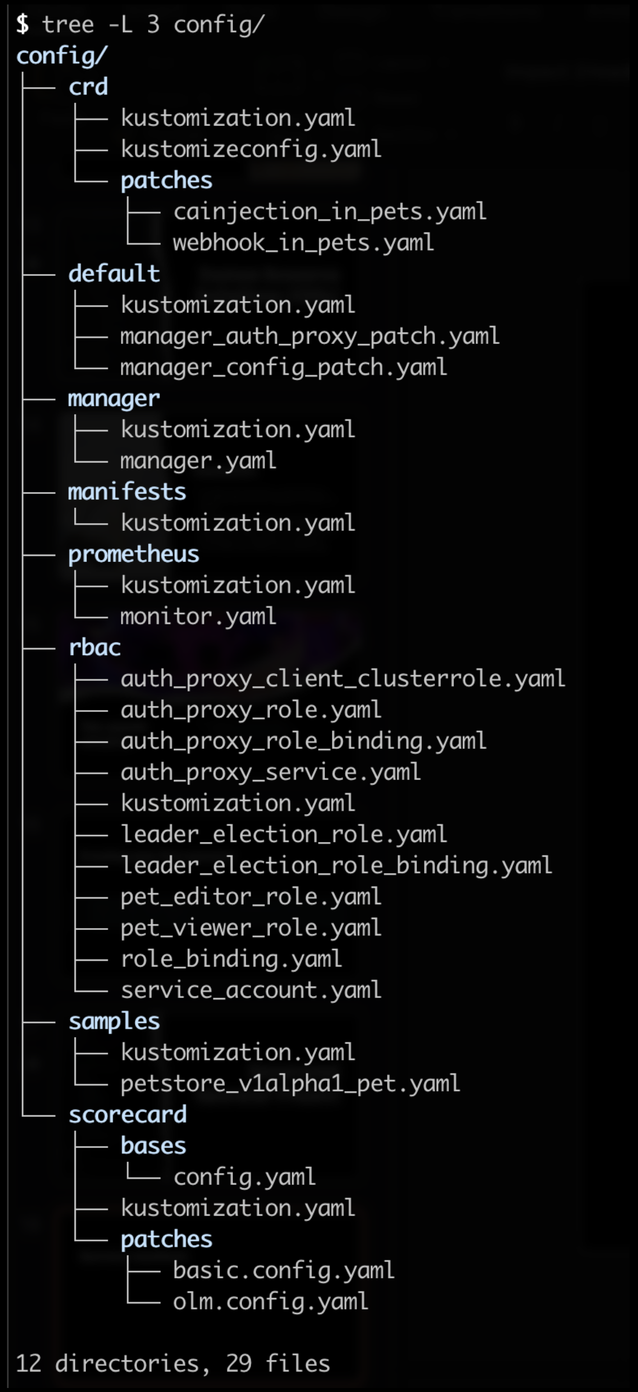
```
# create a new operator project
operator-sdk init --domain example.com --repo github.com/example/first-operator
# create a new API for your operator
operator-sdk create api --group petstore --version v1alpha1 --kind Pet --resource --controller
```

Generated Operator Project



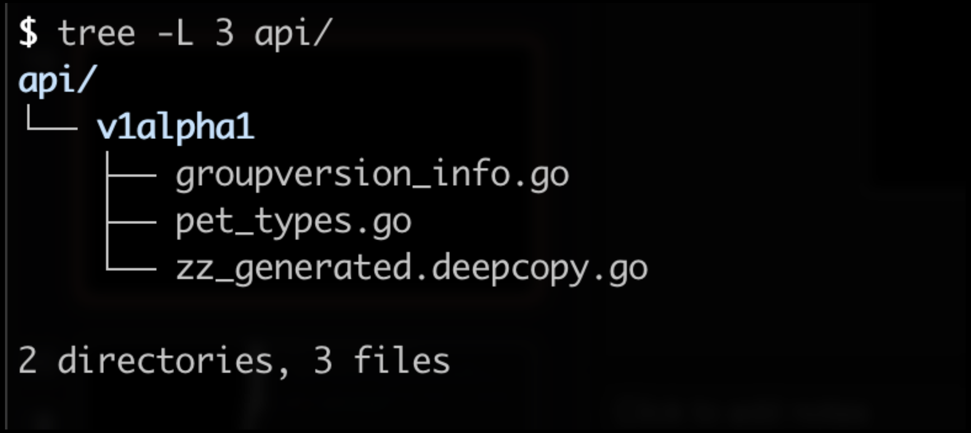
```
$ tree -L 2
.
├── Dockerfile
├── Makefile
├── PROJECT
├── README.md
├── api
│   └── v1alpha1
├── bin
│   └── controller-gen
├── config
│   ├── crd
│   ├── default
│   ├── manager
│   ├── manifests
│   ├── prometheus
│   ├── rbac
│   ├── samples
│   └── scorecard
├── controllers
│   ├── pet_controller.go
│   └── suite_test.go
├── go.mod
├── go.sum
├── hack
│   └── boilerplate.go.txt
└── main.go

17 directories, 11 files
```

Operator manifests



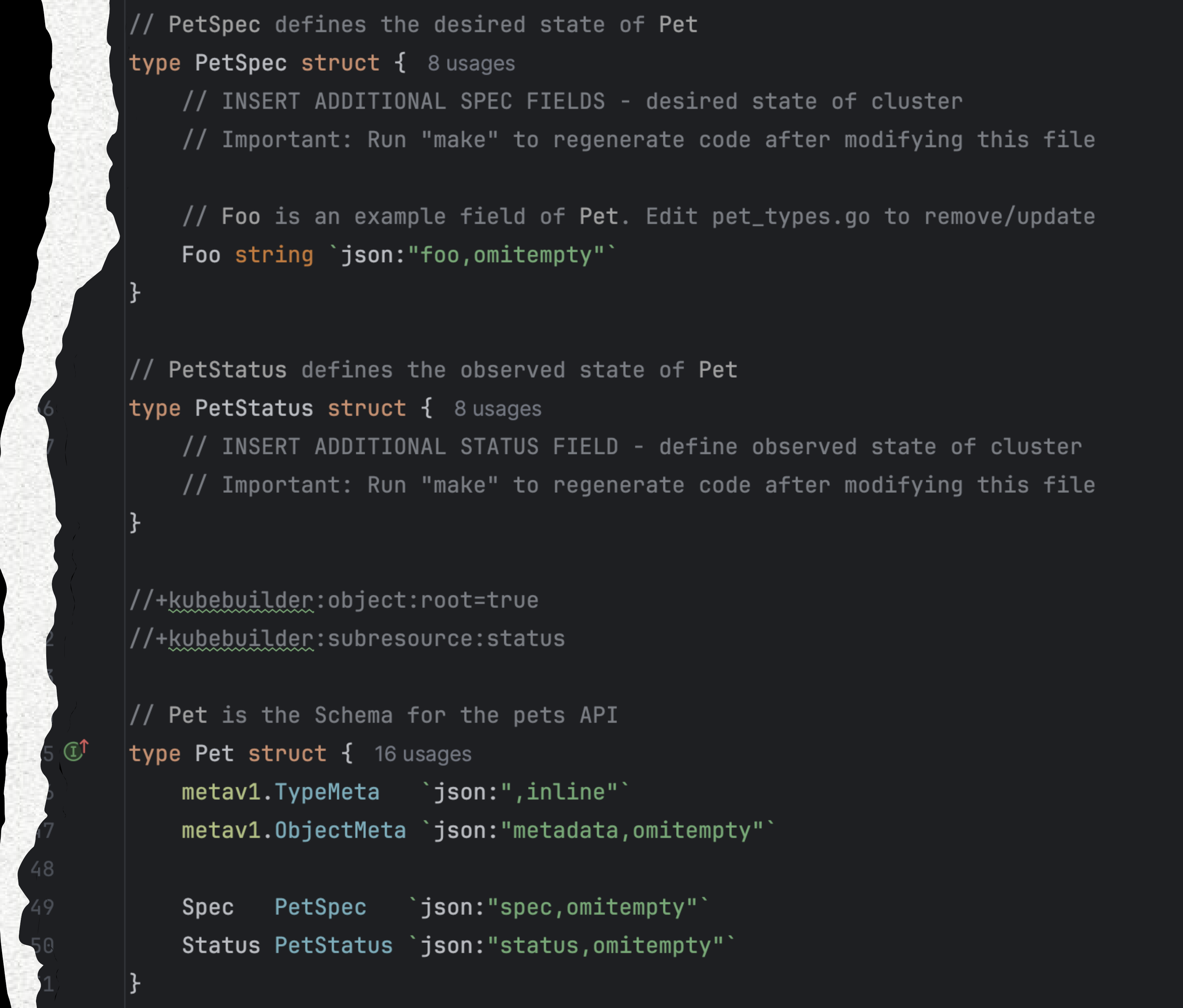
```
$ tree -L 3 config/
config/
├── crd
│   ├── kustomization.yaml
│   ├── kustomizeconfig.yaml
│   └── patches
│       ├── cainjection_in_pets.yaml
│       └── webhook_in_pets.yaml
├── default
│   ├── kustomization.yaml
│   ├── manager_auth_proxy_patch.yaml
│   └── manager_config_patch.yaml
├── manager
│   ├── kustomization.yaml
│   └── manager.yaml
├── manifests
│   └── kustomization.yaml
├── prometheus
│   ├── kustomization.yaml
│   └── monitor.yaml
├── rbac
│   ├── auth_proxy_client_clusterrole.yaml
│   ├── auth_proxy_role.yaml
│   ├── auth_proxy_role_binding.yaml
│   ├── auth_proxy_service.yaml
│   ├── kustomization.yaml
│   ├── leader_election_role.yaml
│   ├── leader_election_role_binding.yaml
│   ├── pet_editor_role.yaml
│   ├── pet_viewer_role.yaml
│   ├── role_binding.yaml
│   └── service_account.yaml
├── samples
│   ├── kustomization.yaml
│   └── petstore_v1alpha1_pet.yaml
└── scorecard
    ├── bases
    │   └── config.yaml
    ├── kustomization.yaml
    └── patches
        ├── basic.config.yaml
        └── olm.config.yaml

12 directories, 29 files
```

Generated Resource Code



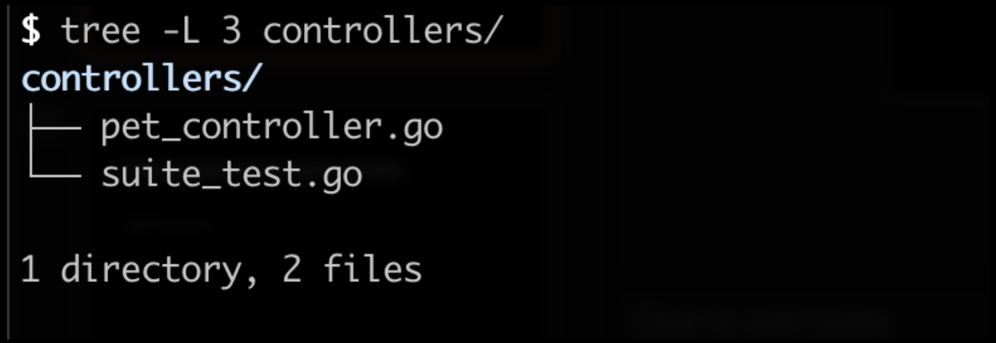
```
$ tree -L 3 api/
api/
└── v1alpha1
    ├── groupversion_info.go
    ├── pet_types.go
    └── zz_generated.deepcopy.go

2 directories, 3 files
```

Stub Pet Resource



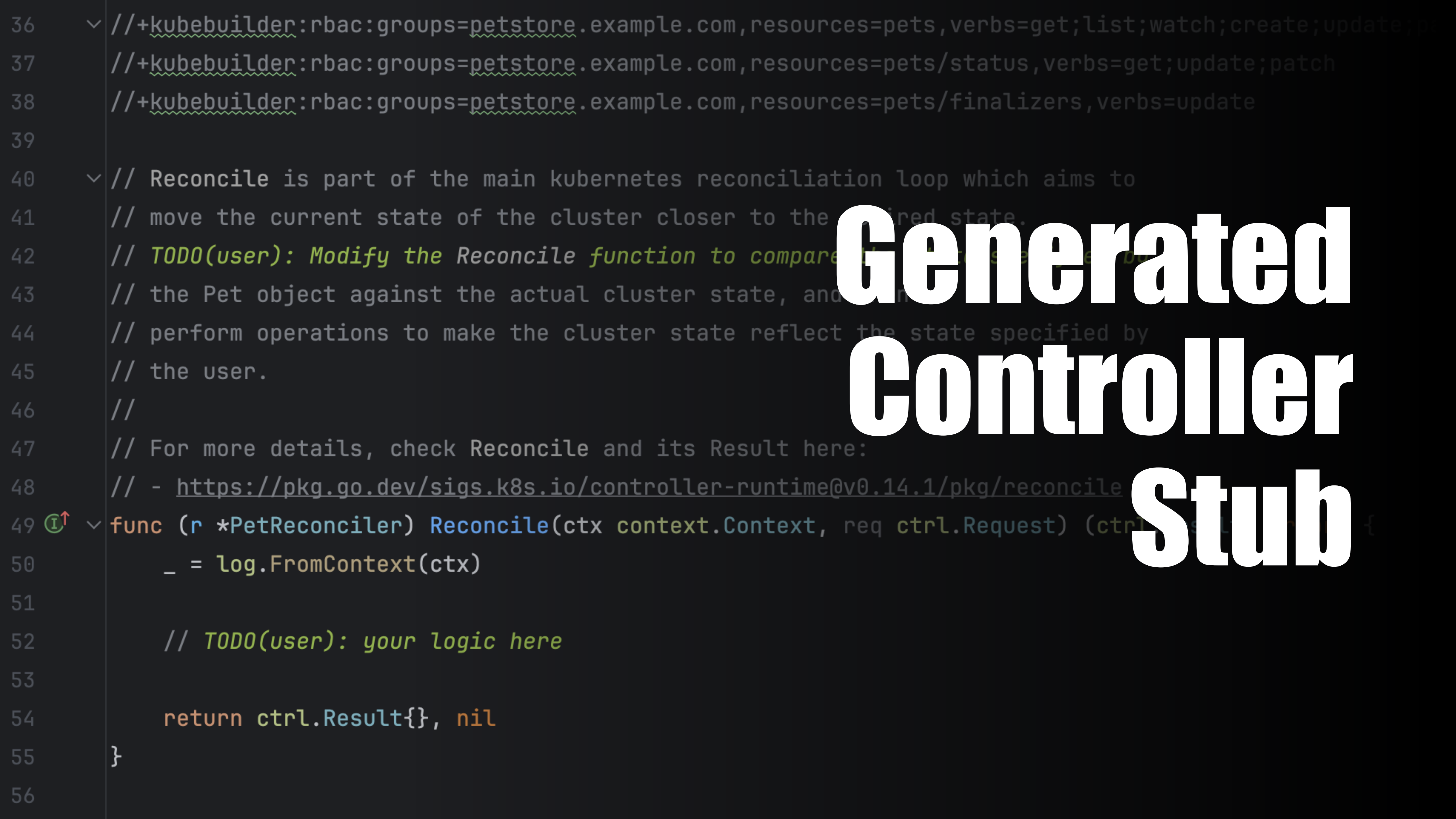
```go
// PetSpec defines the desired state of Pet
type PetSpec struct {  8 usages
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    // Foo is an example field of Pet. Edit pet_types.go to remove/update
    Foo string `json:"foo,omitempty"`
}

// PetStatus defines the observed state of Pet
type PetStatus struct {  8 usages
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status

// Pet is the Schema for the pets API
type Pet struct {  16 usages
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   PetSpec   `json:"spec,omitempty"`
    Status PetStatus `json:"status,omitempty"`
}
```

Generated Controller Code

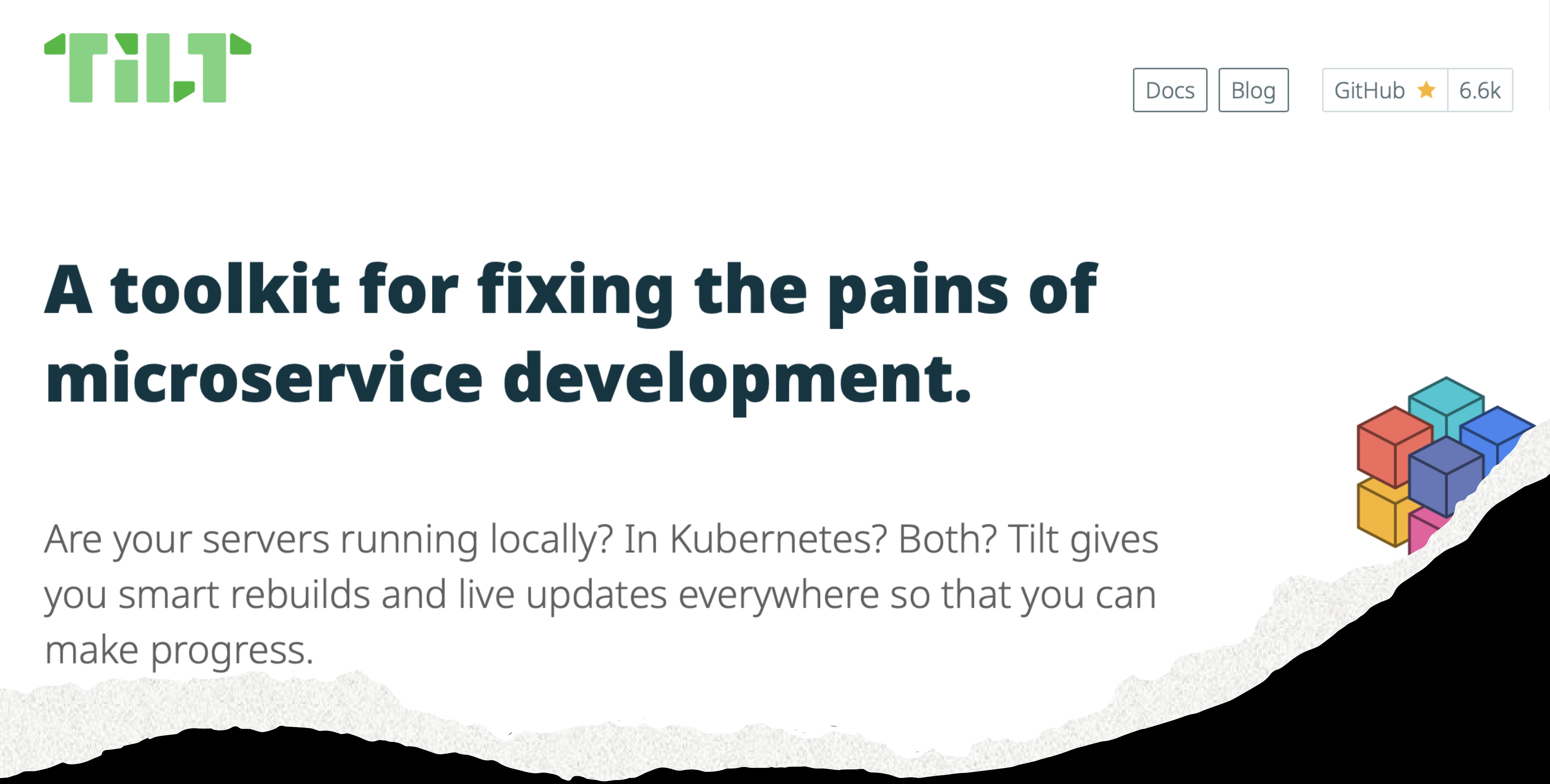


```
$ tree -L 3 controllers/
controllers/
├── pet_controller.go
└── suite_test.go

1 directory, 2 files
```

Generated Controller Stub

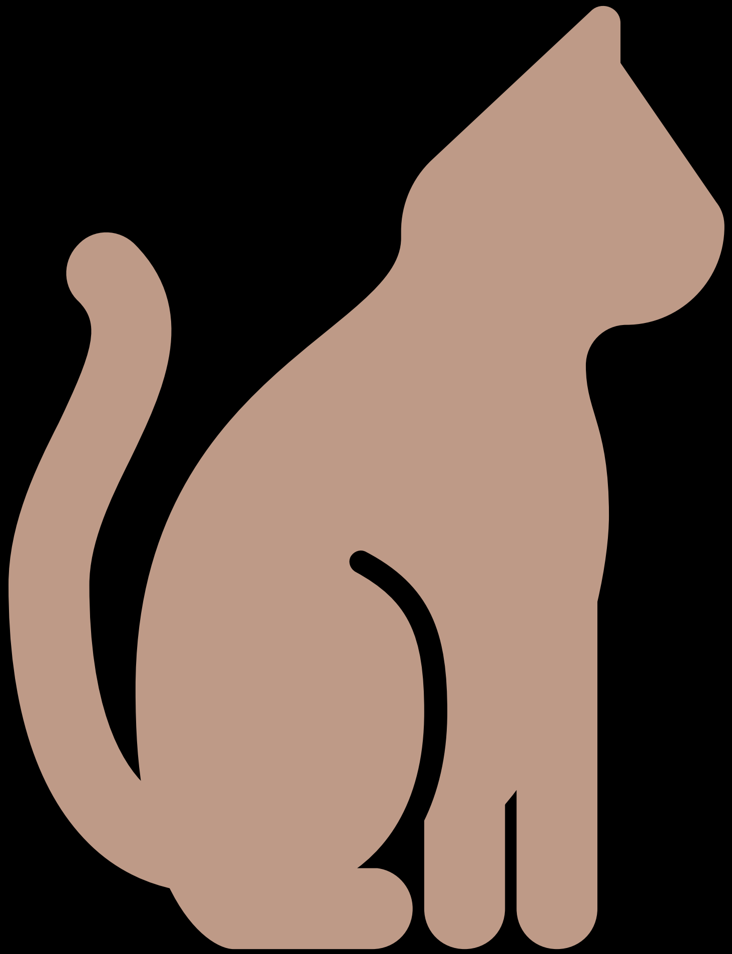


```go
//+kubebuilder:rbac:groups=petstore.example.com,resources=pets,verbs=get;list;watch;create;update;pa
//+kubebuilder:rbac:groups=petstore.example.com,resources=pets/status,verbs=get;update;patch
//+kubebuilder:rbac:groups=petstore.example.com,resources=pets/finalizers,verbs=update

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the Pet object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
//
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.14.1/pkg/reconcile
func (r *PetReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
	_ = log.FromContext(ctx)

	// TODO(user): your logic here

	return ctrl.Result{}, nil
}
```



A toolkit for fixing the pains of microservice development.

Are your servers running locally? In Kubernetes? Both? Tilt gives you smart rebuilds and live updates everywhere so that you can make progress.



Introducing the Pet Store service

- gRPC service to handle CRUD for Pets in our Pet Store.
- Containerized and ready to be deployed within our Kubernetes cluster.

Hands on with Kubernetes CRDs and Controllers – Building a Pet Store

- Build and deploy to a kind cluster an operator for extending Kubernetes to be able to store pets in a pet store service.
- Create an example pet in your cluster using kubectl.
- Inspect the pet you have created using kubectl.
- Stretch goals
  - Create another resource using operator-sdk or kubebuilder, and build a controller to manage that new resource.
  - Add a new version of the Pet CRD that includes a schema change. Ensure that both versions of the Pet CRD can be served at the same time.
  - Add some defaulting or validation logic to a mutating or validating webhook.