

# Implementing a Scheduler Framework Plugin

Learn how to implement a Kubernetes scheduler framework plugin.

We'll cover the following



- Implement a scheduler framework plugin
- Creating a custom plugin
- Hook it up
- Configuration and deployment
- Test the plugin

## Implement a scheduler framework plugin

Both the kube-scheduler and scheduler framework have been written in Go. Therefore, if we want to have our own scheduling plugins, we need to implement them with Go as well.

The Kubernetes community does have a repository [scheduler-plugins](#) that contains out-of-tree scheduler plugins based on the scheduler framework. These high-quality scheduler plugins are exercised in large companies. They are good examples to follow and use in our production environments. If we're creating our scheduler plugins, we can just fork this repo and build on top of it.

In this lesson, we will create an empty repository for a better demonstration.

Now, let's start to implement our custom scheduler plugin.

## Creating a custom plugin

The default scheduler provides a pretty good interface to hook up new out-of-tree plugins. Thus, we don't need to fork the [k8s.io/kubernetes](#) repository.

When writing a new scheduler plugin, we only need to implement the extension points' interfaces defined in the framework. Below is the interface for the extension point `PreFilter` (**line 7**). The interfaces may vary with the Kubernetes version. Here, we're using Kubernetes v1.23.8 as an example.

```
1 // PreFilterPlugin is an interface that must be implemented by the PreFilter plugins.
2 // These plugins are called at the beginning of the scheduling cycle.
3 type PreFilterPlugin interface {
4     Plugin
5     // PreFilter is called at the beginning of the scheduling cycle.
6     // All PreFilter plugins must return success or the Pod will be rejected
7     PreFilter(ctx context.Context, state *CycleState, p *v1.Pod) *Status
8     // PreFilterExtensions returns a PreFilterExtensions interface if the plugin implements one,
9     // or nothing if it does not. A PreFilter plugin can provide extensions to incrementally
10    // modify its preprocessed info. The framework guarantees that the extensions
```

```

10 // modify. AcceptPreFilterExtensions() The framework guarantees that the extensions
11 // AddPod/RemovePod will only be called after PreFilter, possibly on a cloned
12 // CycleState, and may call those functions more than once before calling
13 // Filter again on a specific node.
14 PreFilterExtensions() PreFilterExtensions
15 }

```

The PreFilter interface

Now, we're going to implement a plugin for the extension point PreFilter:

```

1 package myprefilter
2
3 import (
4     "context"
5
6     corev1 "k8s.io/api/core/v1"
7     "k8s.io/apimachinery/pkg/runtime"
8     "k8s.io/kubernetes/pkg/scheduler/framework"
9 )
10
11 var _ framework.PreFilterPlugin = &Plugin{}
12
13 const (
14     PluginName = "MyPreFilter"
15     annFoo     = "mycompany.com/environment"
16 )
17
18 type Plugin struct {
19     confinedAnnotation string
20     handle              framework.Handle
21 }
22
23 func (p *Plugin) Name() string {
24     return PluginName
25 }
26
27 func (p *Plugin) PreFilter(ctx context.Context, state *framework.CycleState, pod *corev1.Pod) *framework.Status {
28     // TODO: Add your own logic here. // In this demo, we use PreFilter to check the preconditions
29     // of Pods before filtering nodes.
30     anns := pod.GetAnnotations()
31     if anns != nil {

```

Implement myPreFilter

We declare `var _ framework.PreFilterPlugin = &Plugin{}` here to make sure our custom plugin implements all the methods in the extension point PreFilter. The method `PreFilter(ctx context.Context, state *CycleState, p *v1.Pod) *Status` is the core function, where we can add our own logic. In the implementation above, we filter the Pods and disallow Pods with the annotation `mycompany.com/environment` for scheduling.

## Hook it up

Both in-tree and out-of-tree scheduling plugins are compiled together into the scheduler. Now, let's hook up our out-of-tree custom plugin. The vanilla default scheduler does provide a hook to register

out-of-tree plugins. To do that, our main function will import `k8s.io/kubernetes/cmd/kube-scheduler/app`, and use the `NewSchedulerCommand` to register our custom plugins by providing the plugin name and constructor function, as shown below. The development environment in which we can add and modify our programs is provided in the code widget below. We can click the “Run” button to initialize it:

search

Search in directory...

/

test-pod.yaml

scheduler-config.yaml

kube-scheduler.yaml

plugins

go.sum

go.mod

Dockerfile

main.go

scheduler-config.yaml

```
1 apiVersion: kubescheduler.config.k8s.io/v1beta3
2 kind: KubeSchedulerConfiguration
3 leaderElection:
4   leaderElect: true
5 clientConnection:
6   kubeconfig: /etc/kubernetes/scheduler.conf
7 profiles:
8   - schedulerName: default-scheduler
9     plugins:
10       multiPoint:
11         enabled:
12           - name: MyPreFilter
13
```

Implementing custom scheduler plugin

## Configuration and deployment

However, we still need to register the plugin and configuration in the scheduler framework. Now, let's create a configuration file, as shown below, to enable our plugin:

```
1 apiVersion: kubescheduler.config.k8s.io/v1beta3
2 kind: KubeSchedulerConfiguration
3 leaderElection:
4   leaderElect: true
5 clientConnection:
6   kubeconfig: /etc/kubernetes/scheduler.conf
7 profiles:
8   - schedulerName: default-scheduler
9     plugins:
```

```
10      multiPoint:
11        enabled:
12          - name: MyPreFilter
```

Configuration for the kube-scheduler

We've already built an image `dixudx/pwk:scheduler-plugin`. If you want to build your own, run the following command in the terminal above:

```
1 docker build -t dixudx/pwk:scheduler-plugin ./
```

Build the container image

Let's modify `/etc/kubernetes/manifests/kube-scheduler.yaml` to run the scheduler. Before that, we make a copy of it and see the changes of the new `kube-scheduler.yaml` with the commands below:

```
1 cp /etc/kubernetes/manifests/kube-scheduler.yaml /usercode/kube-scheduler-old.yaml
2 diff /usercode/kube-scheduler.yaml /usercode/kube-scheduler-old.yaml
```

Show the diffs of kube-scheduler.yaml

Now, we copy it to `/etc/kubernetes/manifests/` to run the new scheduler:

```
1 cp /usercode/scheduler-config.yaml /etc/kubernetes/scheduler-config.yaml
2 cp /usercode/kube-scheduler.yaml /etc/kubernetes/manifests/kube-scheduler.yaml
3 docker ps -a | grep kube-scheduler | grep -v "pause" | awk '{print $1}' | xargs docker rm -f
```

Modify static Pod kube-scheduler

## Test the plugin

Now, let's test our scheduler plugin. We create the test-pod below with the annotation `mycompany.com/environment`.

```
1 kubectl apply -f /usercode/test-pod.yaml
```

Create a Pod for scheduling

After creating it successfully, let's describe it to find the scheduling result.

```
1 kubectl describe pod test-pod
```

View the Pod scheduling result

The output will be as follows:

```
1 Name:      test-pod
2 Namespace: default
3 Priority:   0
4 Node:      <none>
```

```

4  node:      <none>
5  Labels:    <none>
6  Annotations: mycompany.com/environment: abc
7  Status:    Pending
8  IP:        <none>
9  IPs:       <none>
10 Containers:
11   nginx:
12     Image:      nginx
13     Port:       80/TCP
14     Host Port:  0/TCP
15     Environment: <none>
16     Mounts:
17       /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-7kmdq (ro)
18 Conditions:
19   Type                Status
20   PodScheduled        False
21 Volumes:
22   kube-api-access-7kmdq:
23     Type:              Projected (a volume that contains injected data from multiple sources)
24     TokenExpirationSeconds: 3607
25     ConfigMapName:      kube-root-ca.crt
26     ConfigMapOptional:  <nil>
27     DownwardAPI:        true
28   QoS Class:           BestEffort
29   Node-Selectors:      <none>
30   Tolerations:         key=value:NoSchedule
31   kube-api-access-7kmdq:
32     Type:              Projected (a volume that contains injected data from multiple sources)
33     TokenExpirationSeconds: 3607
34     ConfigMapName:      kube-root-ca.crt
35     ConfigMapOptional:  <nil>
36     DownwardAPI:        true
37   QoS Class:           BestEffort
38   Node-Selectors:      <none>
39   Tolerations:         key=value:NoSchedule

```

The output

The events above show that our test-pod is rejected for scheduling. That's exactly what we've set for our plugin.

Ta-da! Our scheduling plugin works!

[← Back](#)

Introduction to the Scheduler Framework



[Next →](#)

Quiz on Schedulers