Packt Book Review: Kubernetes and Docker – An Enterprise Guide

By Scott Surovich | Marc Boorshtein

Summary – An EXCELLENT Enterprise grade Kubernetes book! **Must Read and Strongly Recommend it.** It will help you understand how to INTEGRATE Kubernetes with your existing Enterprise systems, e.g. ***Authentication/Authorization with Microsoft Active Directory, Logging/Observability with Splunk/Elasticsearch/Fluentd, Security Compliance/Auditing with Sysdig Falco, RBAC, OIDC, OPA, PodSecurityPolicies, Networking with DNS, Layer 4/7 LB, Backup and Restore with VMware/Heptio Velero/ARK, CI/CD GitOps pipeline with GitLab, Tekton, ArgoCD, OpenUnison, build a DEV env with KIND and Deploy Apps with Helm*** !

A lot of detail steps/yaml/code samples to help you setup, install, configure and test ALL these above important Enterprise Add-On critical to onboard Kubernetes platform while meeting the strict Enterprise Grade requirements in Security, ID Federation, Compliance, Auditing, Business Continuity Plan (BCP), High Availability, CI/CD, GitOps !

3 sections:

1/ Chap 1-3: Docker and Container Essentials – basics and you can skip if you know it already

2/ Chap 4-6: k8s Dev clusters with KIND and k8s basic objects and services. KIND is better than mini-kube, so try it if you are not familiar with KIND!

3/ Chap 7-END: VERY IMPORTANT. Try to read carefully and setup ALL of these important Add-ons to "ENTERPRISE GRADE" your k8s env! I have CKA and read/write many of these in the past few years. Great to see them GET ORGANIZED ALL in this GREAT book. I learn a lot NEW add-ons/tools in this section. There are many great tools, e.g. audit2rbac, Sysdig kube-psp-advisor, etc... The MOST IMPORTANT and DIFFICULT is the INTEGRATION with your EXISTING ENTERPRISE SYSTEMS! This section is SURE a good HELP!

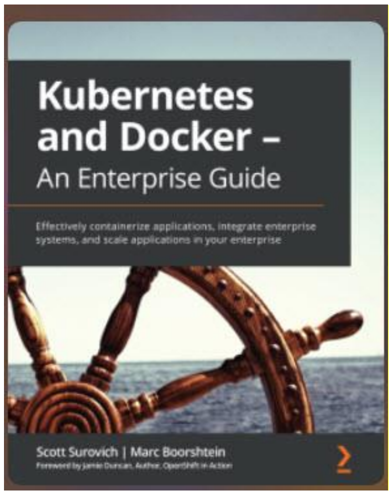
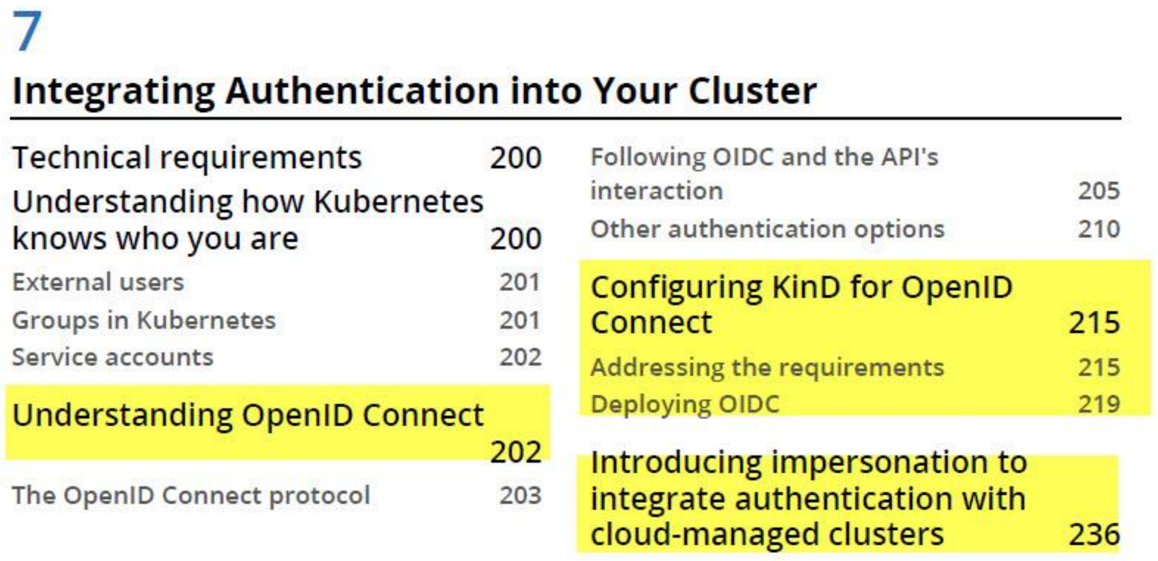
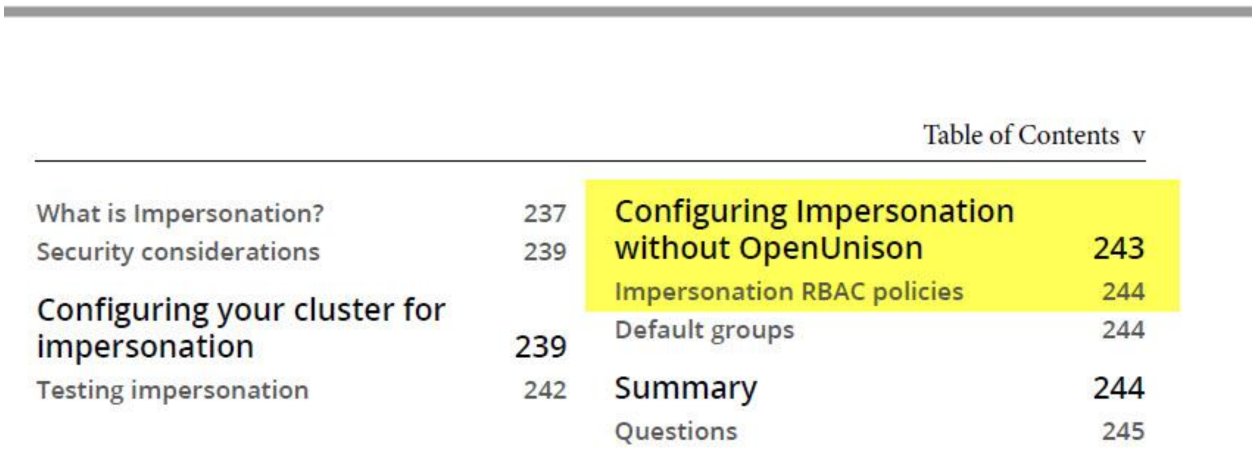

Kubernetes and Docker – An Enterprise Guide
Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise
Scott Surovich | Marc Boorshtein
Foreword by Jamie Duncan, Author, OpenShift in Action

Table of Contents in Section 3 only





8

RBAC Policies and Auditing

9

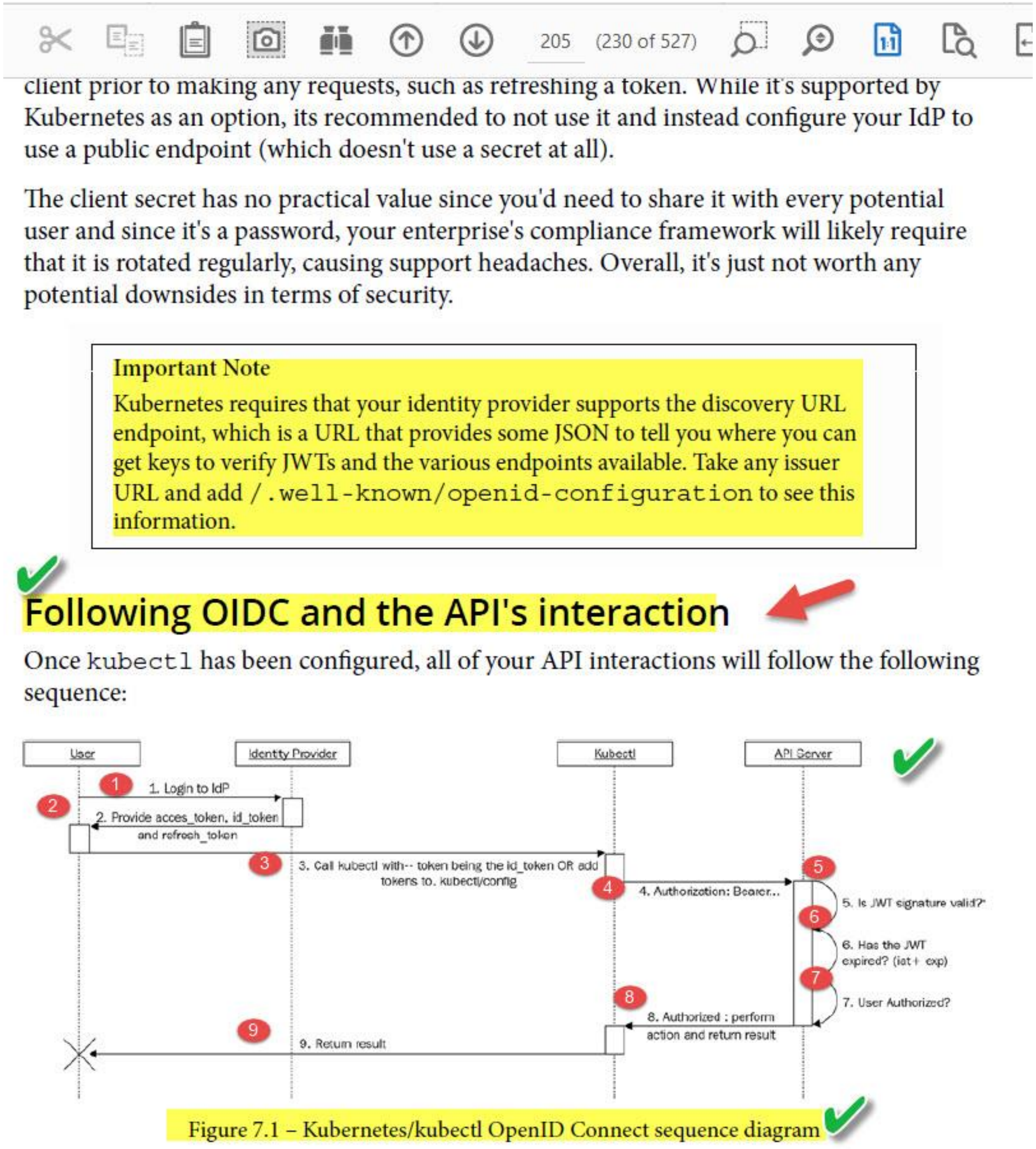
Deploying a Secured Kubernetes Dashboard

12
Auditing using Falco and EFK

13
Backing Up Workloads

14
Provisioning a Platform

A lot of Great and easy to read architecture and diagram



client prior to making any requests, such as refreshing a token. While it's supported by Kubernetes as an option, its recommended to not use it and instead configure your IdP to use a public endpoint (which doesn't use a secret at all).

The client secret has no practical value since you'd need to share it with every potential user and since it's a password, your enterprise's compliance framework will likely require that it is rotated regularly, causing support headaches. Overall, it's just not worth any potential downsides in terms of security.

> **Important Note**
> Kubernetes requires that your identity provider supports the discovery URL endpoint, which is a URL that provides some JSON to tell you where you can get keys to verify JWTs and the various endpoints available. Take any issuer URL and add `/.well-known/openid-configuration` to see this information.

Following OIDC and the API's interaction

Once `kubectl` has been configured, all of your API interactions will follow the following sequence:

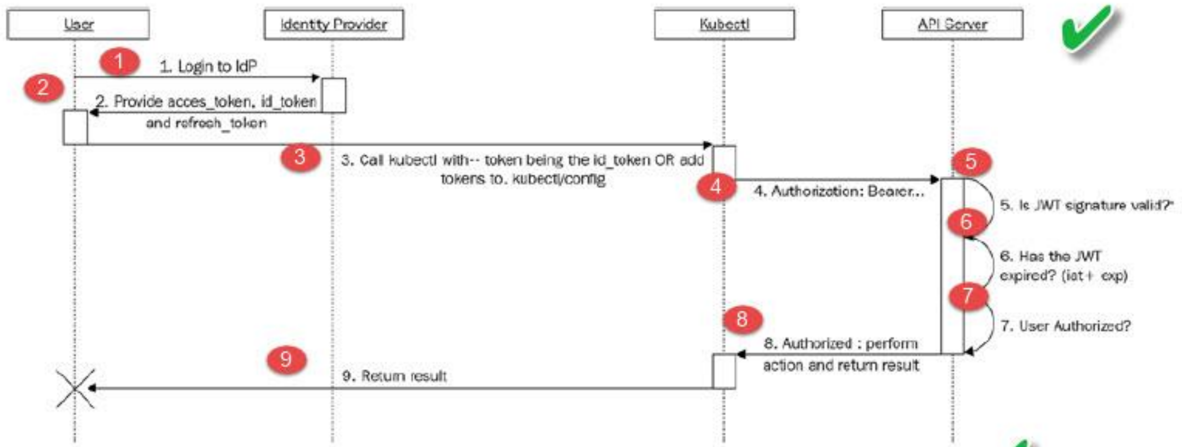

User
Identity Provider
Kubectl
API Server
1
1. Login to IdP
2
2. Provide acces_token, id_token and refresh_token
3
3. Call kubectl with-- token being the id_token OR add tokens to. kubectl/config
4
4. Authorization: Bearer...
5
5. Is JWT signature valid?
6
6. Has the JWT expired? (iat + exp)
7
7. User Authorized?
8
8. Authorized : perform action and return result
9
9. Return result

Figure 7.1 – Kubernetes/kubectl OpenID Connect sequence diagram

Pulling it all together

To fulfill these requirements, we're going to use OpenUnison. It has prebuilt configurations to work with Kubernetes, the dashboard, the CLI, and SAML2 identity providers such as ADFS. It's also pretty quick to deploy, so we don't need to concentrate on provider-specific implementation details and instead focus on Kubernetes' configuration options. Our architecture will look like this:

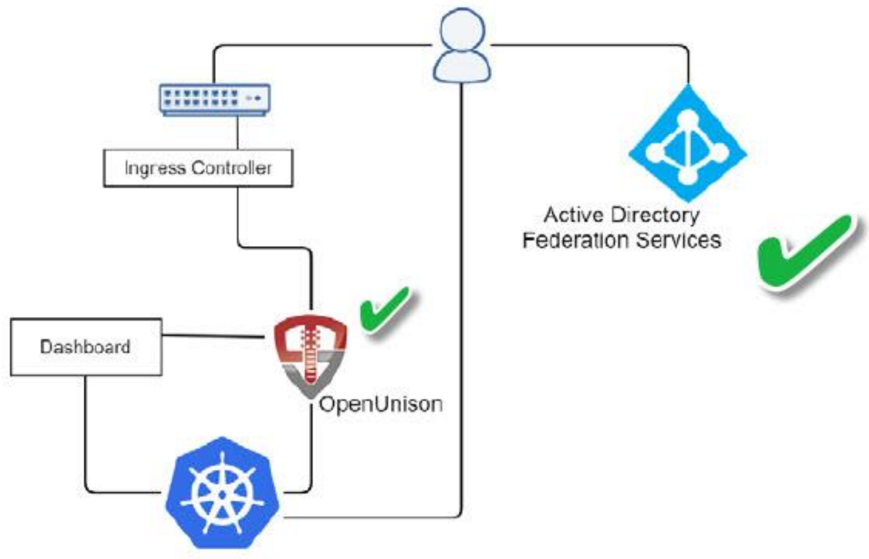


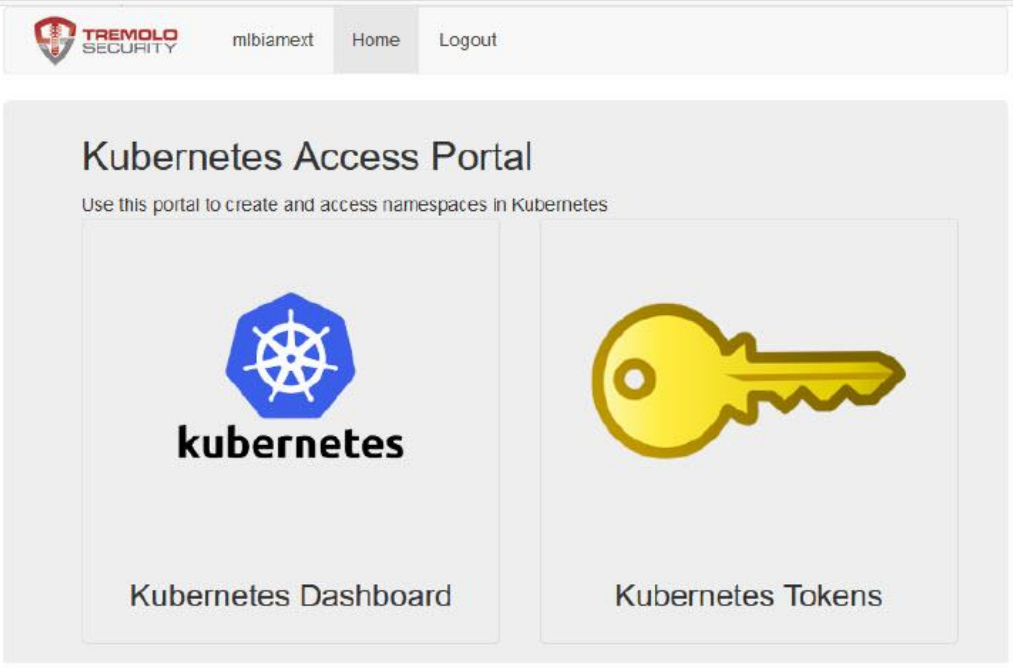
Figure 7.2 – Authentication architecture

For our implementation, we're going to use two hostnames:

- **k8s.apps.X-X-X-X.nip.io**: Access to the OpenUnison portal, where we'll initiate our login and get our tokens
- **k8sdb.apps.X-X-X-X.nip.io**: Access to the Kubernetes dashboard

Important Note

As a quick refresher, `nip.io` is a public DNS service that will return an IP address from the one embedded in your hostname. This is really useful in a lab



Kubernetes Access Portal

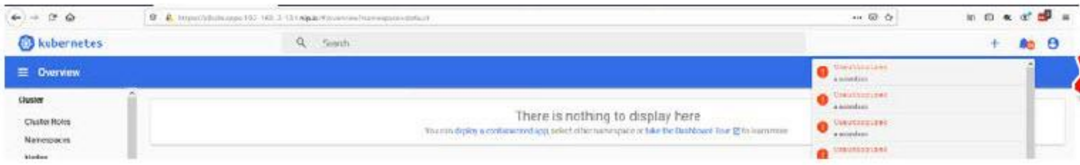
Use this portal to create and access namespaces in Kubernetes

Kubernetes Dashboard

Kubernetes Tokens

Figure 7.7 – OpenUnison home screen

17. Let's test the OIDC provider by clicking on the **Kubernetes Dashboard** link. Don't panic when you look at the initial dashboard screen – you'll see something like the following:



There is nothing to display here

Figure 7.8 – Kubernetes Dashboard before SSO integration has been completed with the API server

That looks like a lot of errors! We're in the dashboard, but nothing seems to be authorized. That's because the API server doesn't trust the tokens that have been generated by OpenUnison yet. The next step is to tell Kubernetes to trust OpenUnison as its OpenID Connect Identity Provider.

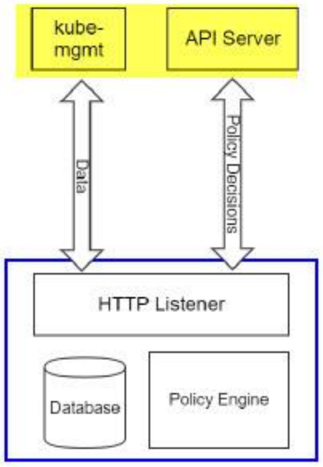


Figure 11.2 – OPA in Kubernetes

When used in Kubernetes, OPA populates its database using a side car, called *kube-mgmt*, which sets up watches on the objects you want to import into OPA. As objects are created, deleted, or changed, *kube-mgmt* updates the data in its OPA instance. This means that OPA is "eventually consistent" with the API server, but it won't necessarily be a real-time representation of the objects in the API server. Since the entire etcd database is essentially being replicated over and over again, great care needs to be taken in order to refrain from replicating sensitive data, such as `Secrets`, in the OPA database.

Rego, the OPA policy language

We'll cover the details of Rego in the next section in detail. The main point to mention here is that Rego is a policy evaluation language, not a generic programming language. This can be difficult for developers who are used to languages such as Golang, Java, or JavaScript, which support complex logic such as iterators and loops. Rego is designed to evaluate policy and is streamlined as such. For instance, if you wanted to write code in Java to check that all the container images in a `Pod` starting with one of a list of registries, it would look something like the following:

```
public boolean validRegistries(List<Container>
containers,List<String> allowedRegistries) {
  for (Container c : containers) {
      boolean imagesFromApprovedRegistries = false;
    for (String allowedRegistry : allowedRegistries) {
```

Our application in Kubernetes is defined as a series of objects stored in `etcd` that are generally represented as code using YAML files. Why not store those files in a Git repository too? This gives us the same benefits as storing our application code in Git. We have a single source of truth for both the application source and the operations of our application! Now, our pipeline involves some more steps:

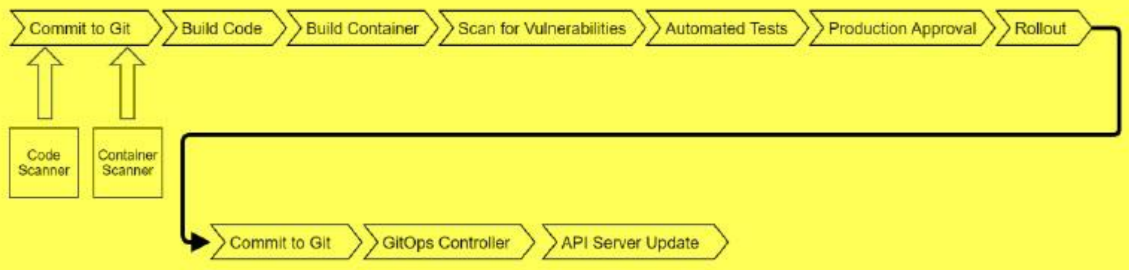


Figure 14.4 – GitOps pipeline

In this diagram, our rollout updates a Git repository with our application's Kubernetes YAML. A controller inside our cluster watches for updates to Git and when it sees them, gets the cluster in sync with what's in Git. It can also detect drift in our cluster and bring it back to alignment with our source of truth.

This focus on Git is called **GitOps**. The idea is that all of the work of an application is done via code, not directly via APIs. How strict you are with this idea can dictate what your platform looks like. Next, we'll explore how opinions can shape your platform.

Opinionated platforms

Kelsey Hightower, a developer advocate for Google and leader in the Kubernetes world, once said: "Kubernetes is a platform for building platforms. It's a better place to start; not the endgame." When you look at the landscape of vendors and projects building Kubernetes-based products, they all have their own opinions of how systems should be built. As an example, Red Hat's **OpenShift Container Platform** (**OCP**) wants to be a one-stop-shop for multi-tenant enterprise deployment. It builds in a great deal of the

SPLK-1003.pdf SPLK-2002.pdf Packt - 9781838

468 (493 of 527)

Automating project onboarding using OpenUnison

Earlier in this chapter, we deployed the OpenUnison automation portal. This portal lets users request new namspaces to be created and allows developers to request access to these namespaces via a self-service interface. The workflows built into this portal are very basic but create the namespace and appropriate `RoleBinding` objects. What we want to do is build a workflow that integrates our platform and creates all of the objects we created manually earlier in this chapter. The goal is that we're able to deploy a new application into our environment without having to run the `kubectl` command (or at least minimize its use). This will require careful planning. Here's how our developer workflow will run:

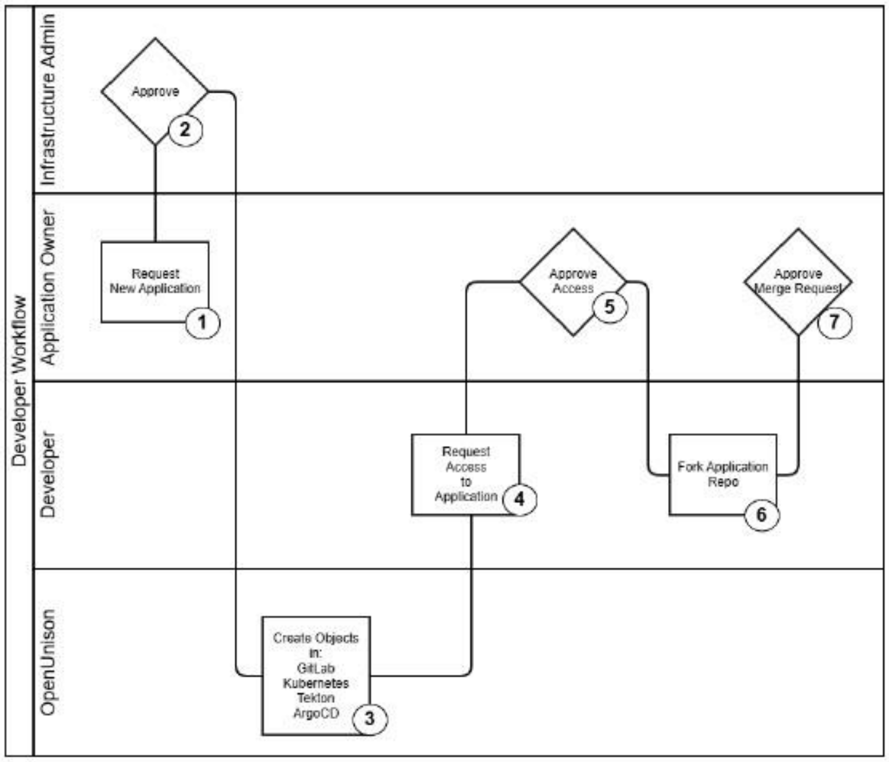


Figure 14.6 – Platform developer workflow

Nowhere in this flow is there a step called "operations staff uses `kubectl` to create a namespace." This is a simple flow and won't totally avoid your operations staff from using `kubectl`, but it should be a good starting point. All this automation requires an extensive set of objects to be created:

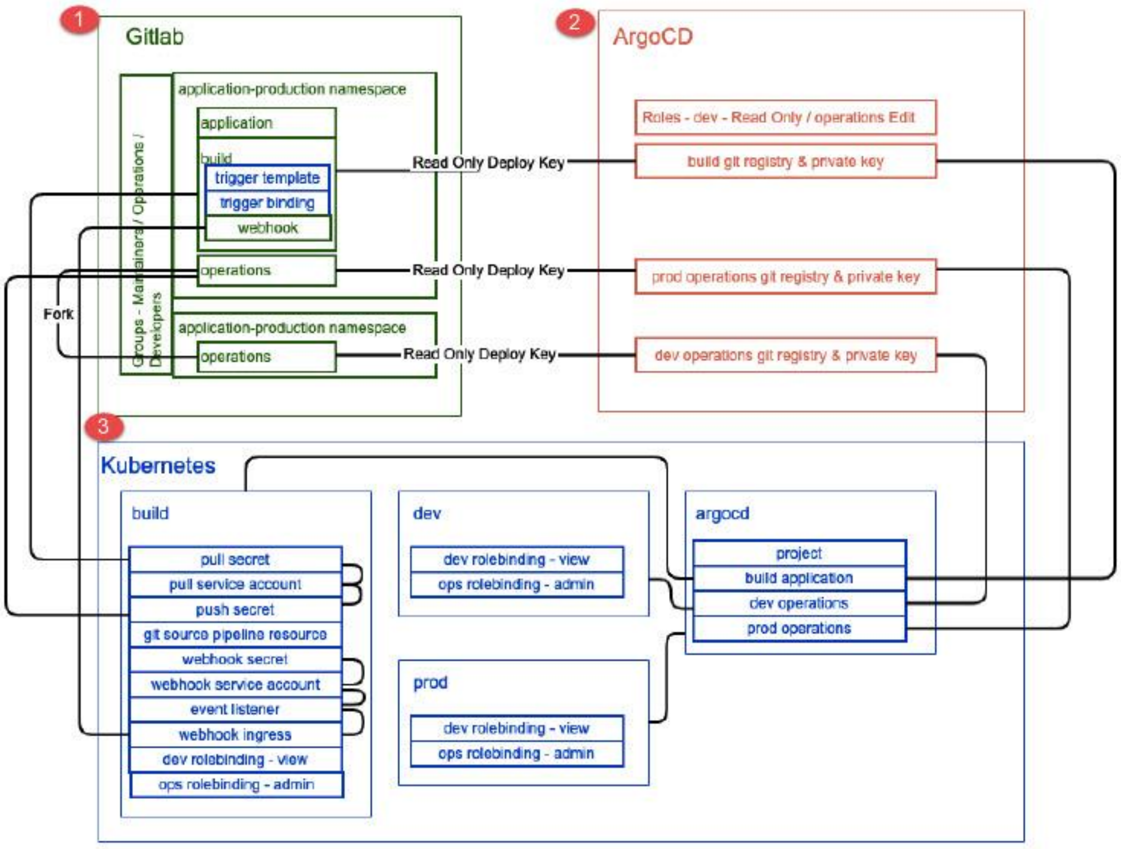
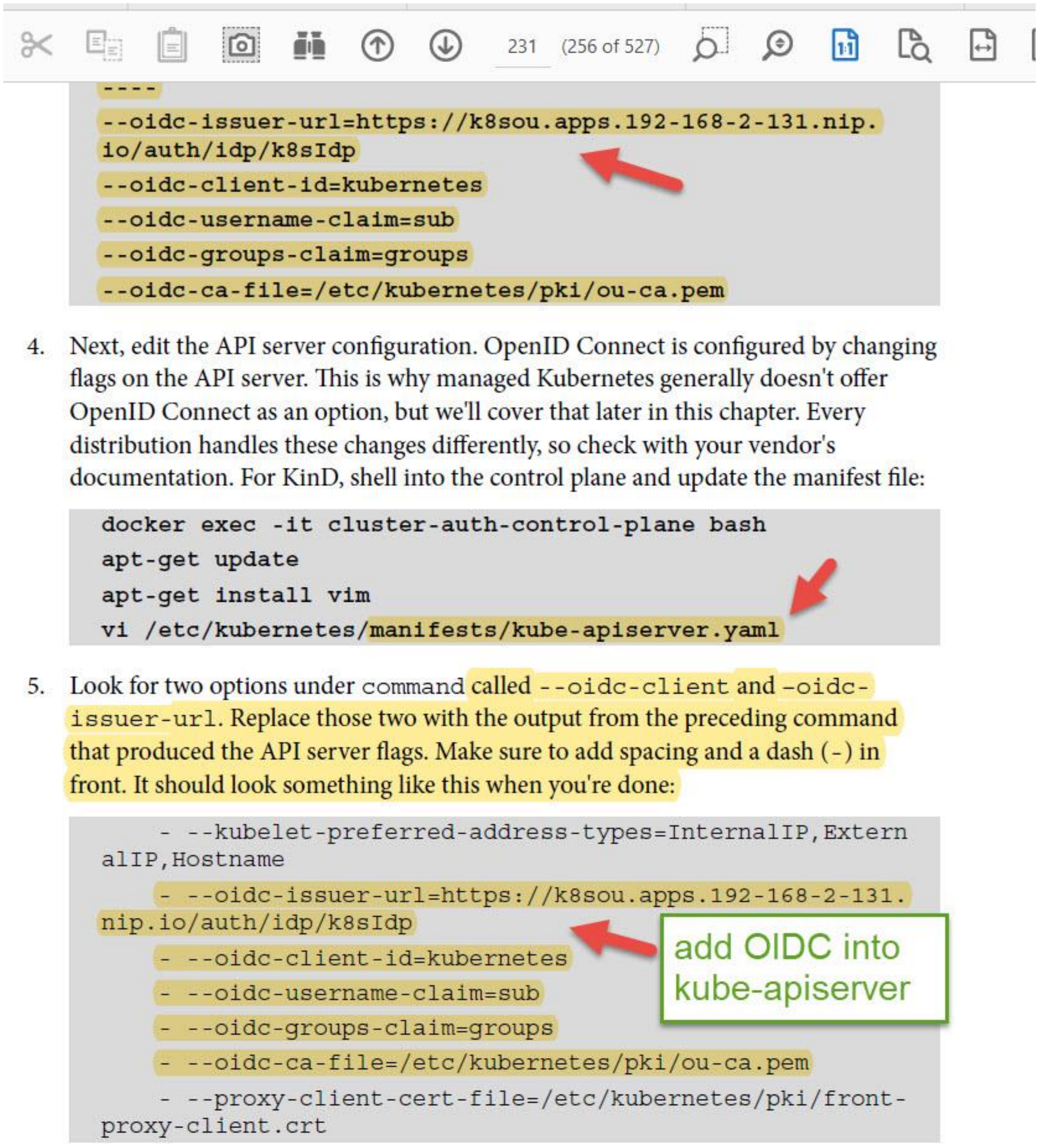


Figure 14.7 – Application onboarding object map

In GitLab, we create a project for our application code, operations, and build pipeline. We also fork the operations project as a development operations project. For each project, we

A lot of GREAT and SIMPLE Sample Code/Yaml so you can learn/setup easily



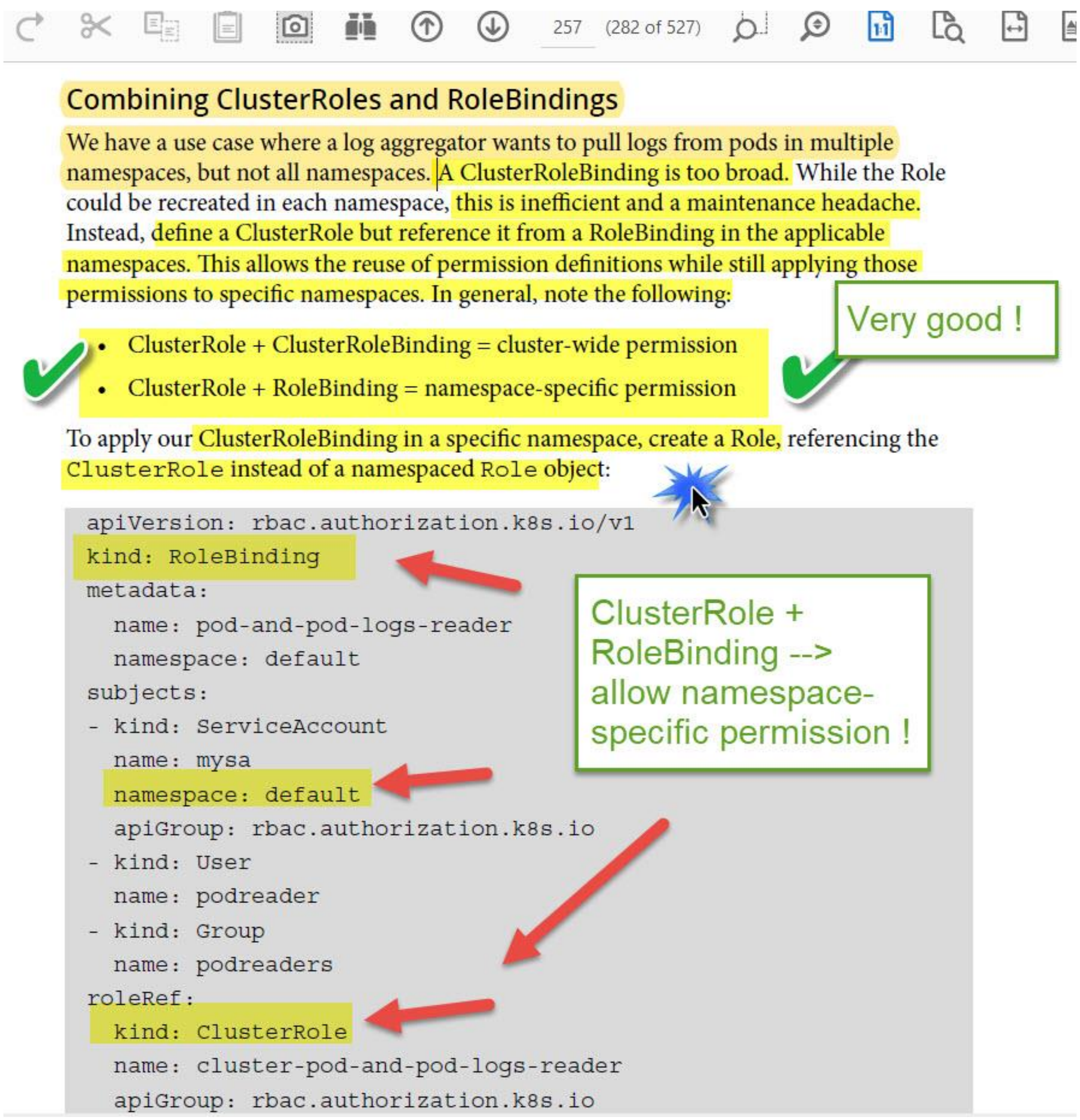
```
----
--oidc-issuer-url=https://k8sou.apps.192-168-2-131.nip.
io/auth/idp/k8sIdp
--oidc-client-id=kubernetes
--oidc-username-claim=sub
--oidc-groups-claim=groups
--oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
```

4. Next, edit the API server configuration. OpenID Connect is configured by changing flags on the API server. This is why managed Kubernetes generally doesn't offer OpenID Connect as an option, but we'll cover that later in this chapter. Every distribution handles these changes differently, so check with your vendor's documentation. For KinD, shell into the control plane and update the manifest file:

```
docker exec -it cluster-auth-control-plane bash
apt-get update
apt-get install vim
vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

5. Look for two options under `command` called `--oidc-client` and `-oidc-issuer-url`. Replace those two with the output from the preceding command that produced the API server flags. Make sure to add spacing and a dash (-) in front. It should look something like this when you're done:

```
    - --kubelet-preferred-address-types=InternalIP,Extern
alIP,Hostname
    - --oidc-issuer-url=https://k8sou.apps.192-168-2-131.
nip.io/auth/idp/k8sIdp
    - --oidc-client-id=kubernetes
    - --oidc-username-claim=sub
    - --oidc-groups-claim=groups
    - --oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-
proxy-client.crt
```


257
(282 of 527)
Very good !
ClusterRole + RoleBinding --> allow namespace-specific permission !

Combining ClusterRoles and RoleBindings

We have a use case where a log aggregator wants to pull logs from pods in multiple namespaces, but not all namespaces. A ClusterRoleBinding is too broad. While the Role could be recreated in each namespace, this is inefficient and a maintenance headache. Instead, define a ClusterRole but reference it from a RoleBinding in the applicable namespaces. This allows the reuse of permission definitions while still applying those permissions to specific namespaces. In general, note the following:

- ClusterRole + ClusterRoleBinding = cluster-wide permission
- ClusterRole + RoleBinding = namespace-specific permission

To apply our ClusterRoleBinding in a specific namespace, create a Role, referencing the `ClusterRole` instead of a namespaced `Role` object:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-and-pod-logs-reader
  namespace: default
subjects:
- kind: ServiceAccount
  name: mysa
  namespace: default
  apiGroup: rbac.authorization.k8s.io
- kind: User
  name: podreader
- kind: Group
  name: podreaders
roleRef:
  kind: ClusterRole
  name: cluster-pod-and-pod-logs-reader
  apiGroup: rbac.authorization.k8s.io
```

7. Similar to the `audit2rbac` tool used when debugging RBAC policies, Sysdig has published a tool that will inspect the pods in a namespace and generate a recommended policy and RBAC set. Download the latest version from `https://github.com/sysdiglabs/kube-psp-advisor/releases`:

```
./kubectl-advise-psp inspect  --namespace=ingress-nginx
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  creationTimestamp: null
  name: pod-security-policy-ingress-nginx-20200611232031
spec:
  defaultAddCapabilities:
  - NET_BIND_SERVICE
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - max: 80
    min: 80
  - max: 443
    min: 443
  requiredDropCapabilities:
  - ALL
  runAsUser:
    ranges:
    - max: 101
      min: 101
    rule: MustRunAs
  seLinux:
    rule: RunAsAny
```

Using Rego to write policies

Rego is a language specifically designed for policy writing. It is different to most languages you have likely written code in. Typical authorization code will look something like the following:

```
//assume failure
boolean allowed = false;
//on certain conditions allow access
if (someCondition) {
  allowed = true;
}
//are we authorized?
if (allowed) {
  doSomething();
}
```

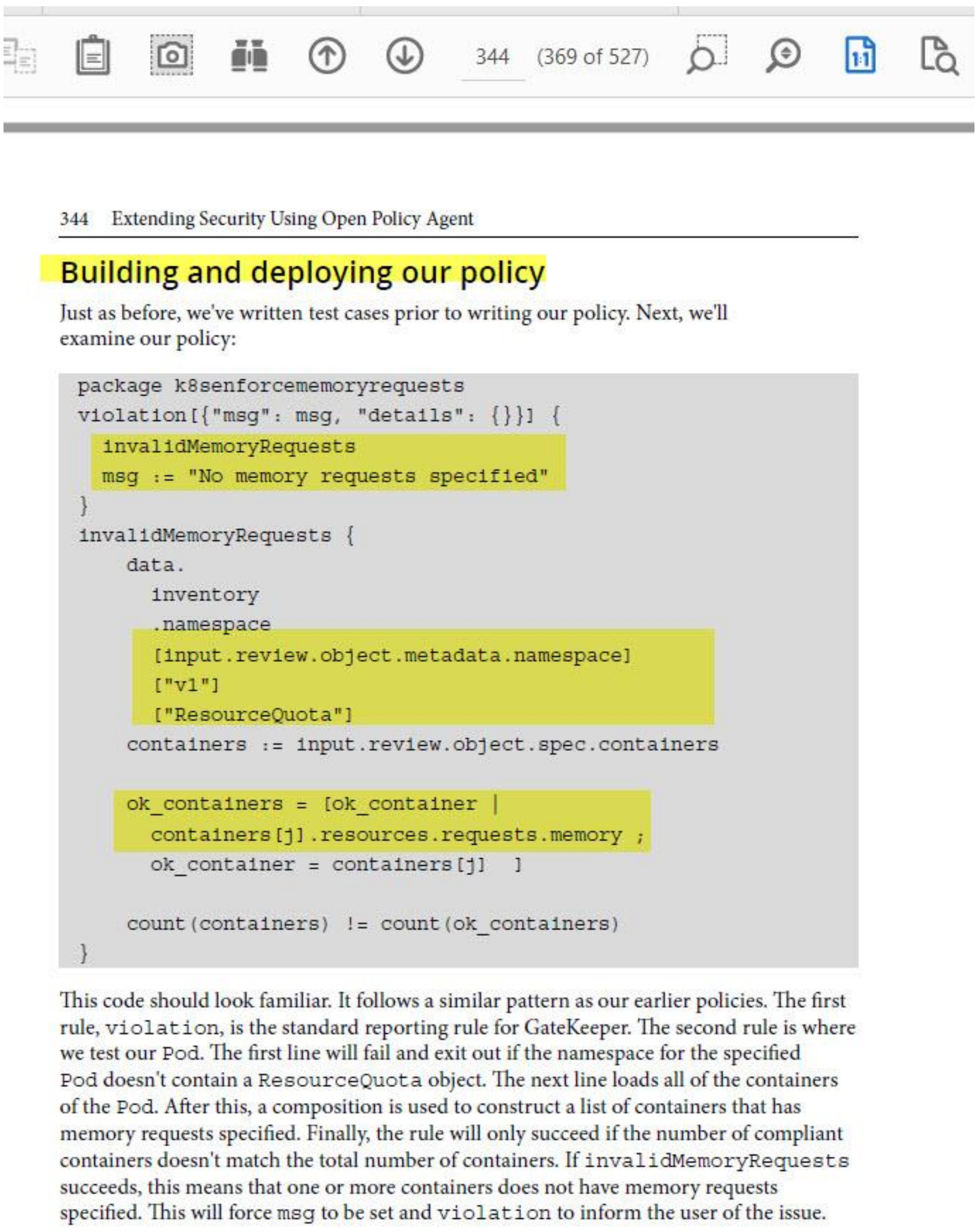
Authorization code will generally default to unauthorized, with a specific condition having to happen in order to allow the final action to be authorized. Rego takes a different approach. Rego is generally written to authorize everything unless a specific set of conditions happens.

Another major difference between Rego and more general programming languages is that there are no explicit "`if/then/else`" control statements. When a line of Rego is going to make a decision, the code is interpreted as "if this line is false, stop execution." For instance, the following code in Rego says "if the image starts with `myregistry.lan/`, then stop execution of the policy and pass this check, otherwise generate an error message":

```
not startsWith(image,"myregistry.lan/")
msg := sprintf("image '%v' comes from untrusted registry",
[image])
```

The same code in Java might look as follows:

```
if (! image.startsWith("myregistry.lan/")) {
   throw new Exception("image " + image + " comes from
untrusted registry");
}
```



Building and deploying our policy

Just as before, we've written test cases prior to writing our policy. Next, we'll examine our policy:

```
package k8senforcememoryrequests
violation[{"msg": msg, "details": {}}] {
  invalidMemoryRequests
  msg := "No memory requests specified"
}
invalidMemoryRequests {
    data.
      inventory
      .namespace
      [input.review.object.metadata.namespace]
      ["v1"]
      ["ResourceQuota"]
    containers := input.review.object.spec.containers

    ok_containers = [ok_container |
      containers[j].resources.requests.memory ;
      ok_container = containers[j]  ]

    count(containers) != count(ok_containers)
}
```

This code should look familiar. It follows a similar pattern as our earlier policies. The first rule, `violation`, is the standard reporting rule for GateKeeper. The second rule is where we test our `Pod`. The first line will fail and exit out if the namespace for the specified `Pod` doesn't contain a `ResourceQuota` object. The next line loads all of the containers of the `Pod`. After this, a composition is used to construct a list of containers that has memory requests specified. Finally, the rule will only succeed if the number of compliant containers doesn't match the total number of containers. If `invalidMemoryRequests` succeeds, this means that one or more containers does not have memory requests specified. This will force `msg` to be set and `violation` to inform the user of the issue.