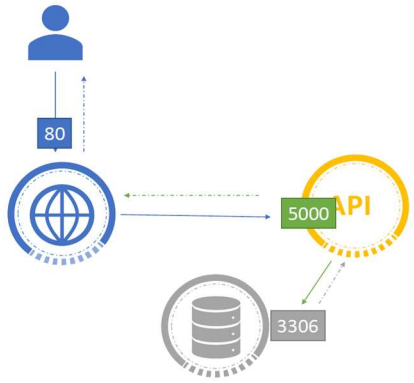
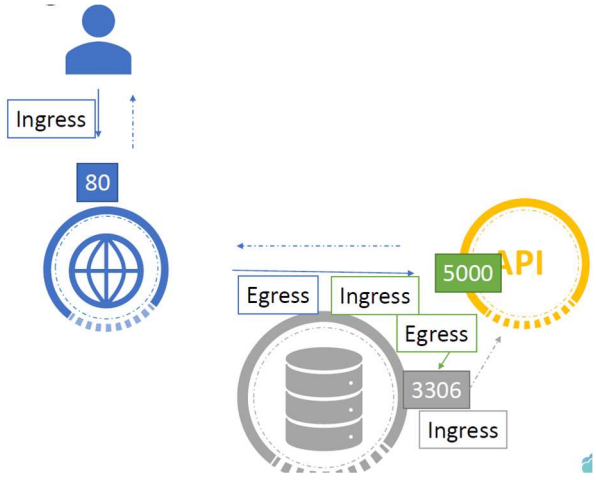
Network Policy

So let us first get our networking and security basics right. We will start with a simple example of a traffic flowing through a web app and database server. So you have a web server serving front end to users an app server serving backend APIs, and a database server. The user sends in a request to the web server at port 80. The web server then sends a request to the API server at port 5000 in the backend. The API server then fetches data from the database server at port 3306, and then sends the data back to the user. A very simple setup.



So there are two types of traffic here, ingress and egress. For example, for a web server, the incoming traffic from the users is an ingress traffic and the outgoing request to the app server is egress traffic and that is denoted by the straight arrow. When you define ingress and egress, remember you're only looking at the direction in which the traffic originated. The response back to the user denoted by the dotted lines do not really matter.

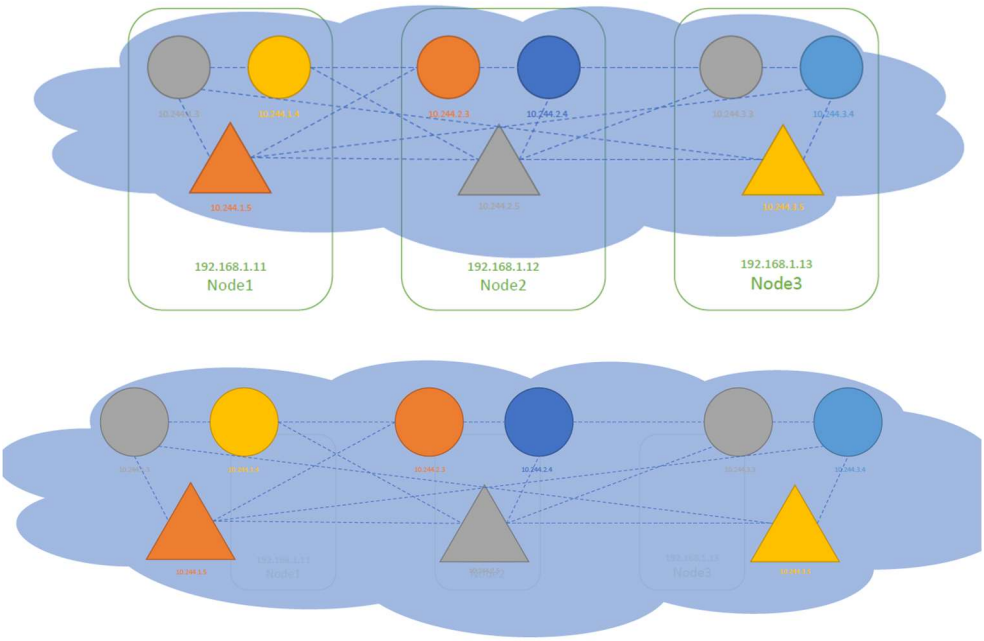
Similarly, in case of the backend API server, it receives ingress traffic from the web server on port 5000 and has egress traffic to port 3306 to the database server. And from the database server's perspective, it receives ingress traffic on port 3306 from the API server.



If we were to list the rules required to get this working, we would have an ingress rule that is required to accept HTTP traffic on port 80 on the web server, an egress rule to allow traffic from

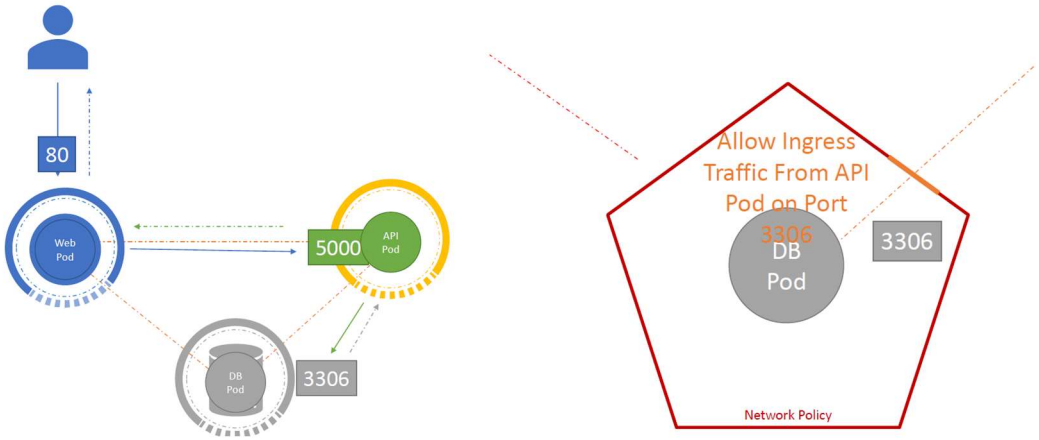
the web server to port 5000 on the API server, an ingress rule to accept traffic on port 5000 on the API server and an egress rule to allow traffic to port 3306 on the database server. And finally, an ingress rule on the database server to accept traffic on port 3306. So that's the basic of traffic flow and rules.

Let us now look at network security in Kubernetes. So we have a cluster with a set of nodes hosting a set of pods and services. Each node has an IP address and so does each pod as well as service. One of the prerequisite for networking in Kubernetes is whatever solution you implement; the pods should be able to communicate with each other without having to configure any additional settings like routes. For example, in this network solution, all pods are on a virtual private network that spans across the node in the Kubernetes cluster and they can all by default reach each other using the IPs or pod names or services configured for that purpose. Kubernetes is configured by default with an all allow rule that allows traffic from any pod to any other pod or services within the cluster.

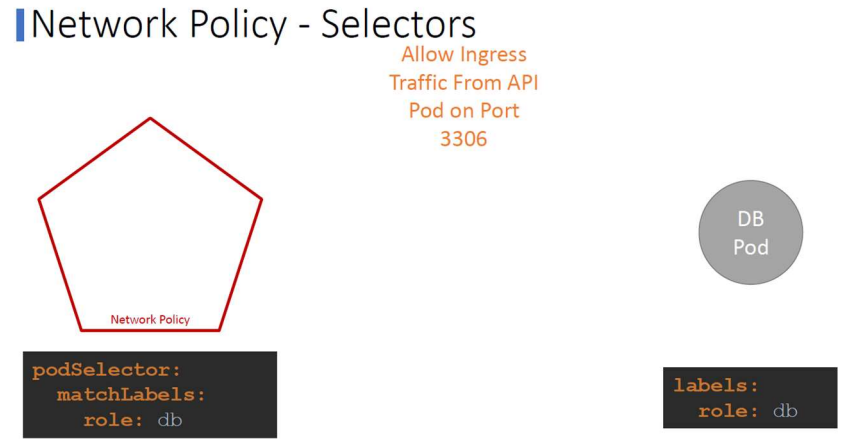


Let us now bring back our earlier discussion and see how it fits into Kubernetes. For each component in the application, we deploy a pod. One for the front end web server, for the API server and one for the database. We create services to enable communication between the as well as to the end user. Based on what we discussed in the previous slide, by default, all the three pods can communicate with each other within the Kubernetes cluster. What if we do not want the front end web server to be able to communicate with the database server directly? Say, for example, your security teams and audits require you to prevent that from happening. That is where you would implement a network policy to allow traffic to the DB server only from the API server.

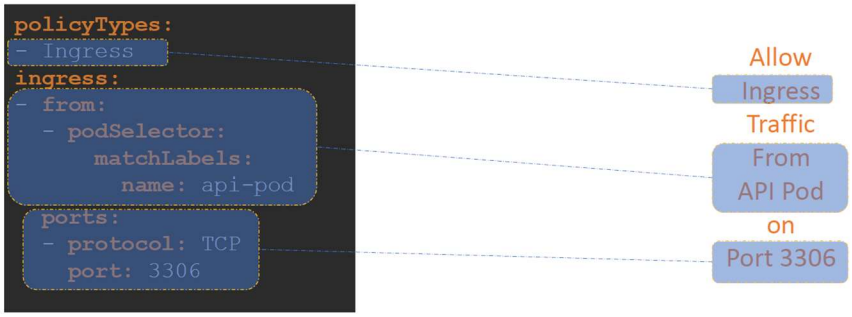
And network policy is another object in the Kubernetes namespace just like pods, replica sets or services, you link a network policy to one or more pods. You can define rules within the network policy. In this case, I would say only allow ingress traffic from the API pod on port 3306. Once this policy is created, it blocks all other traffic to the pod and only allows traffic that matches the specified rule. Again, this is only applicable to the pod on which the network policy is applied.



So how do you apply or link a network policy to a pod? We use the same technique that was used before to link replica sets or services to pod, labels and selectors. We label the pod and use the same labels on the port selector field in the network policy and then we build our rule.



Under policy types, specify whether the rule is to allow ingress or egress traffic or both. In our case, we only want to allow ingress traffic to the DB pod, so we add ingress. Next, we specify the ingress rule. That allows traffic from the API pod and you specify the API pod again using labels and selectors. And finally, the port to allow traffic on, which is 3306.



Let us put all that together. We start with a blank object definition file, and as usual, we have API version, kind, metadata and spec. The API version is networking.k8s.io/v1. The kind is network policy, we will name the policy DB-policy and then under the spec section, we will first move the pod selector to apply this policy to the DB pod. Then we will move the rule we created in the previous slide under it, and that's it. We have our first network policy ready.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
   podSelector:
     matchLabels:
       role: db
   policyTypes:
   - Ingress
   ingress:
   - from:
     - podSelector:
         matchLabels:
           name: api-pod
     ports:
     - protocol: TCP
       port: 3306
```

Now note that ingress or egress isolation only comes into effect if you have ingress or egress in the policy types. In this example, there's only ingress in the policy type which means only ingress traffic is isolated and all egress traffic is unaffected. Meaning the pod is able to make any egress calls and they're not blocked. So for an egress or ingress isolation to take place, note that you have to add them under the policy types as seen here. Otherwise, there is no isolation. Run the Kube control create command to create the policy.

Remember that network policies are enforced by the network solution implemented on Kubernetes cluster and not all network solutions support network policies. A few of them that are

supported are Cube Router, Calico, Romana, and WaveNet. If you used Flannel as the networking solution, it does not support network policies as of this recording. Always referred to the Network Solutions Documentation to see support for network policies.

Solutions that Support Network Policies:
- Kube-router
- Calico
- Romana
- Weave-net

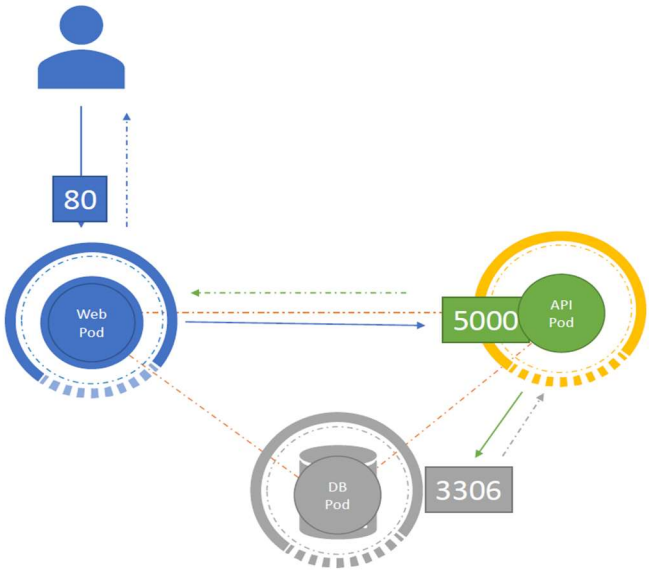
Solutions that DO NOT Support Network Pol
- Flannel

Also, remember, even in a cluster configured with a solution that does not support network policies, you can still create the policies but they will just not be enforced. You will not get an error message saying the network solution does not support network policies.

Developing network policy

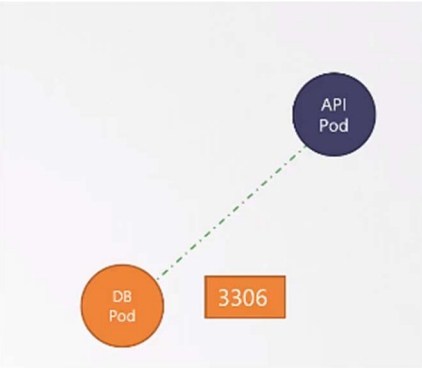
In this lecture, we will take a look at network policies in more detail. So here we have the same web API and database pods that we discussed about in the previous lecture. So first, let's be very clear with our requirements. Our goal is to protect the database pod so that it does not allow access from any other pod except the API pod and only on port 3306.

So let's assume that we are not concerned about the web pod or the API pod. For those pods, we are okay for all traffic to go in and out from anywhere. However, we want to protect the database pod and only allow traffic from the API pod.



So let's get the other things out of our way so we can focus exactly on the required tasks. So we don't need to worry about the web pod or its port, as we don't want to allow any traffic from any

other sources other than the API pod. So let's get rid of that. We can also forget about the port on the API pod to which the web server connects, as we don't care about that either.



As we discussed, by default, Kubernetes allows all traffic from all pods to all destinations. So as the first step, we want to block out everything going in and out of the database pod.

So we create a network policy, we will call it DB policy, and the first step is to associate this network policy with the pod that we want to protect. And we do that using labels and selectors. So we do that by adding a pod selector field with the match labels option and by specifying the label on the DB pod, which happens to be said to role DB, and that associates the network policy with the database pod. Now, it still doesn't block out traffic because we haven't specified policy types yet.

So if there are no policy types specified in the network policy definition, the protection is not enforced. Since we'd like to restrict both ingress and egress traffic, we will add ingress and egress to the policy types, and that should block all ingress and egress traffic on the pod.

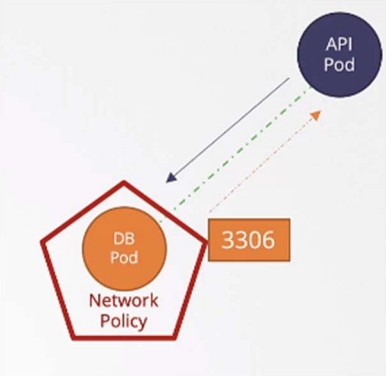
However, we need the API pod to be able to query the database on port 3306. So that's what we are going to configure next. First, we need to figure out what type of policies should be defined on this network policy object to meet our requirements.

So there are two types of policies that we discussed in the previous lecture. We have ingress and egress. So do we need ingress or egress here, or both? So you always look at this from the DB pods perspective. From the DB pods perspective, we want to allow incoming traffic from the API pod. So that is incoming, so that is ingress. The API pod makes database queries and the database pods returns the results.

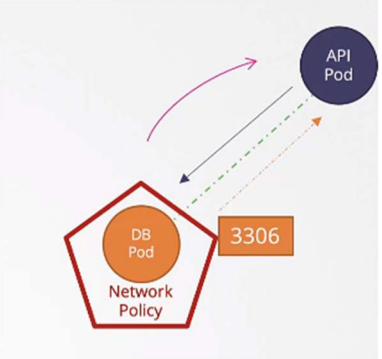
So what about the results? Do you need a separate rule for the results to go back to the API pod? No, because once you allow incoming traffic, the response or reply to that traffic is allowed back automatically. We don't need a separate rule for that.

So in this case, all we need is an ingress rule to allow traffic from the API pod to the database pod, and that would allow the API pod to connect to the database and run queries and also retrieve the result of the queries.

So when deciding on what type of rule is to be created, you only need to be concerned about the direction in which the request originates, which is denoted by the straight line here, and you don't need to worry about the response, which is denoted by the dotted line.



However, this rule does not mean that the database pod will be able to connect to the API pod or make calls to the API. Say, for example, the database pod tries to make an API call to the API pod, then that would not be allowed because that is now an egress traffic originating from the database pod and would require a specific egress rule to be defined.



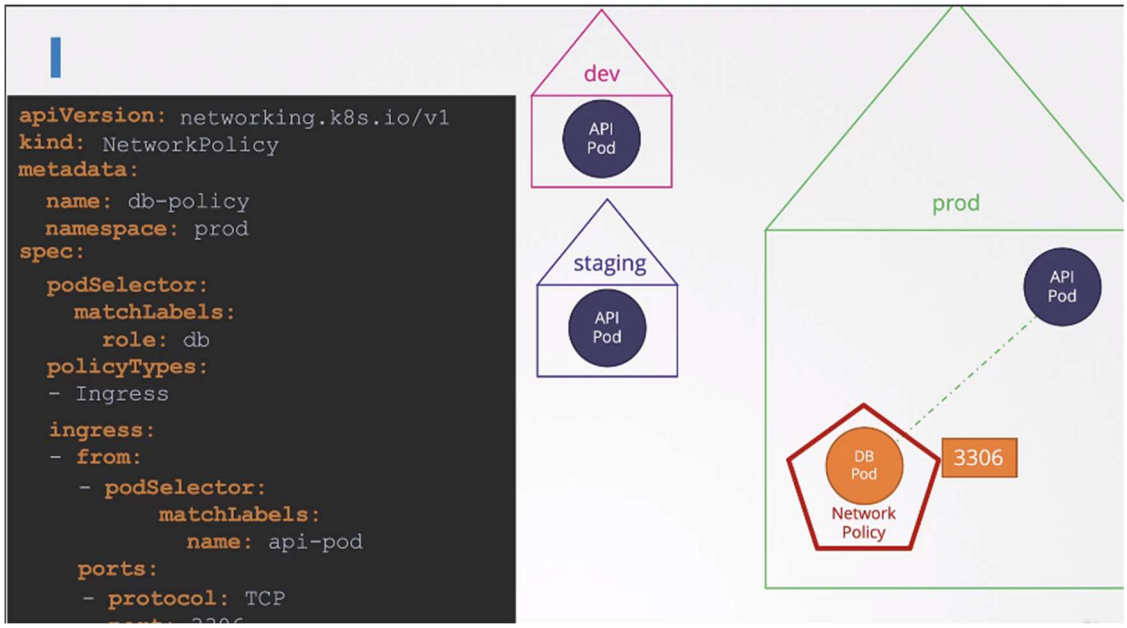
So I hope you get the difference between the two and are clear about ingress and egress rules.I just wanted to make sure that you're clear on what type of policy is to be selected for the requirement that you have in hand. So a single network policy can have an ingress type of rule, an egress type of rule, or both in cases where a pod wants to allow incoming connections, as well as wants to make external calls. So for now, our use case only requires ingress policy types.

Now that we have decided on the type of policy, the next step is to define the specifics of that policy. If it's ingress, we create a section called ingress within which we can specify multiple rules. Each rule as a from and ports fields. The from field defines the source of traffic that is allowed to pass through to the database pod. And here we would use a pod selector and provide the labels of the API pod. The ports field defines what port on the database pod is the traffic allowed to go to. In this case, it's 3306 with the TCP protocol. And that's it. This would create a policy that would block all traffic to the DB pod except for traffic from the API pod.

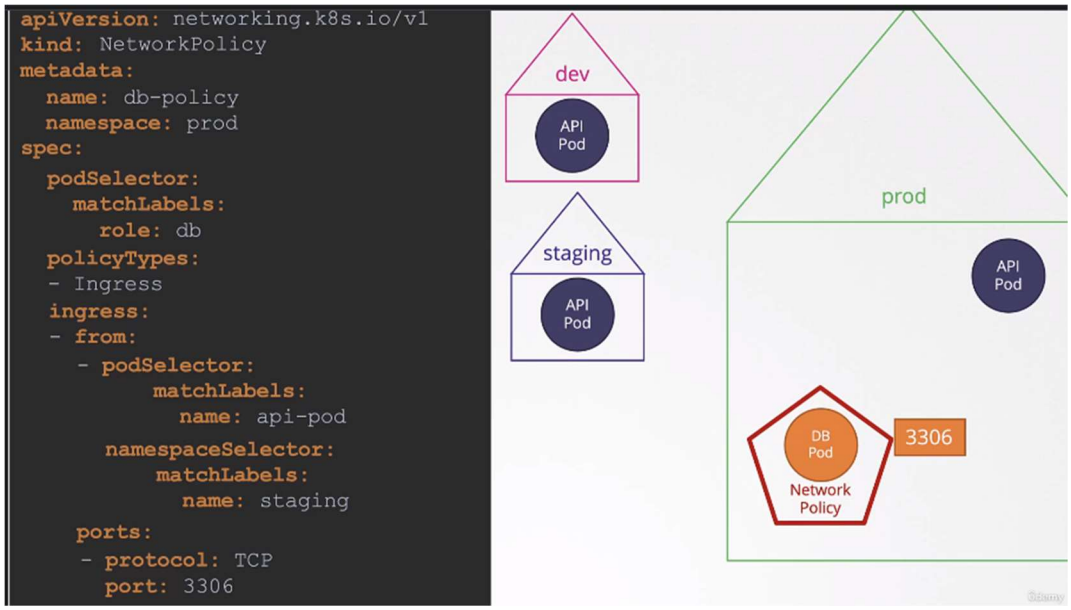
```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db

  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-pod
    ports:
    - protocol: TCP
      port: 3306
```

Now, what if there are multiple API pods in the cluster with the same labels but in different namespaces? Here we have different namespaces for dev, staging, and prod environments, and we have the API pod with the same labels in each of these environment.

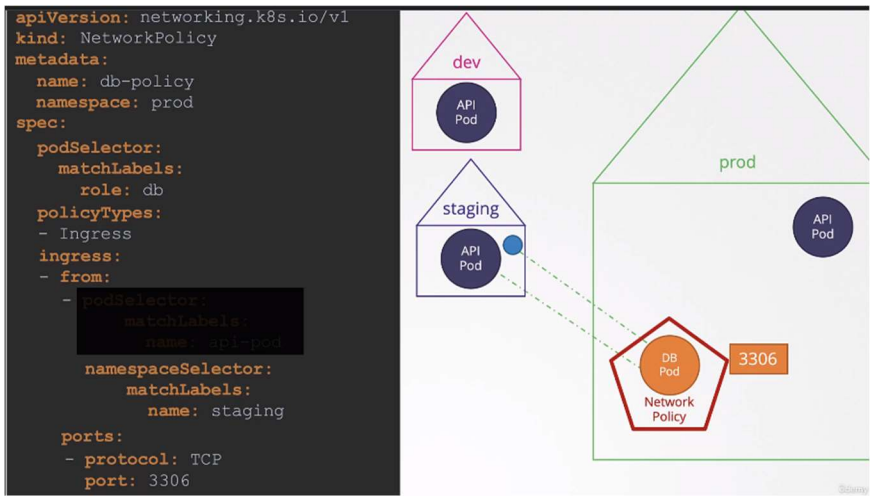
And first, if you are creating a network policy for the prod namespace, which is a namespace other than the default namespace, then it should have the namespace defined as prod on the network policy object definition. Now by default, this policy will only allow the pod in the prod namespace, which is the same namespace as that of the network policy, but matching labels to reach the database pod.



But what if for some reason we want the pod in the staging namespace, which is another namespace, to reach the database in the production namespace? For this, we add a namespace selector property, as well along with the pod selector property. Under this, we use match labels again to provide a label set on the namespace. Remember, you must have this label set on the namespace first for this to work. So just note that. So that's what the namespace selector does. It helps in defining from what namespace traffic is allowed to reach to the database pod.



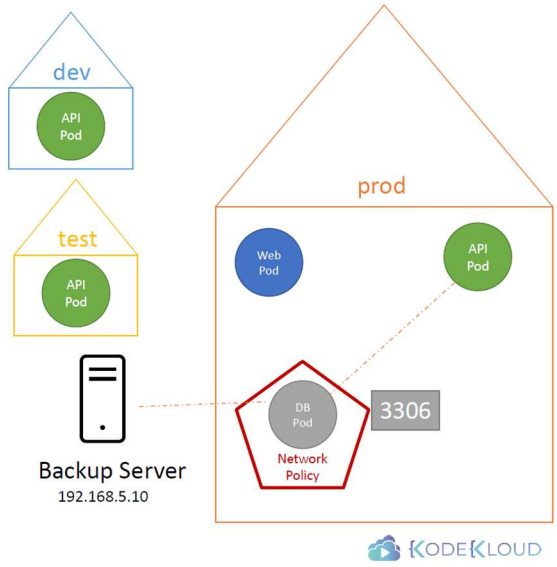
Now what if you only have the namespace selector and not the pod selector like this? In that case, all pods within the specified namespace will be allowed to reach the database pod, such as the web pod in the other, in the staging namespace. But pods from outside this namespace won't be allowed to go through.



Let's look at another use case. Say we have a backup server somewhere outside of the Kubernetes cluster and we want to allow this server to connect to the database pod. Now since this backup

server is not deployed in our Kubernetes cluster, the pod selector and namespace selector fields that we use to define traffic from won't work because it's not a pod in the cluster.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-pod
      namespaceSelector:
        matchLabels:
          name: prod
    ports:
    - protocol: TCP
      port: 3306
```

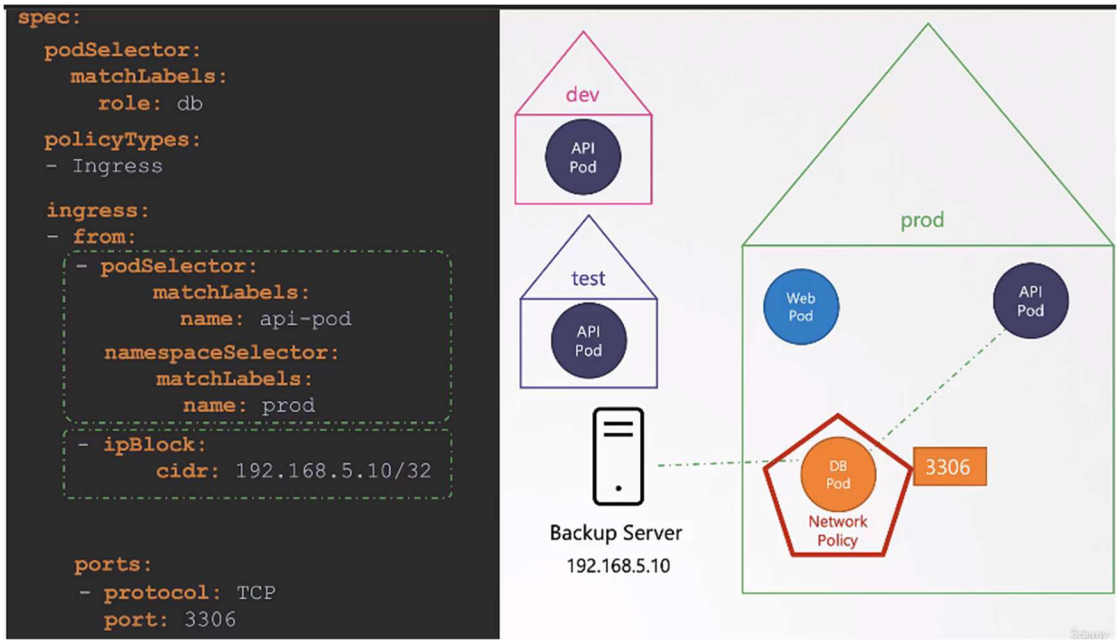


However, we know the IP address of the backup server, and that happens to be 192.168.5.10. We could configure a network policy to allow traffic originating from certain IP addresses. For this, we add a new type of from definition known as the IP block definition. IP block allows you to specify a range of IP addresses from which you could allow traffic to hit the database pod.

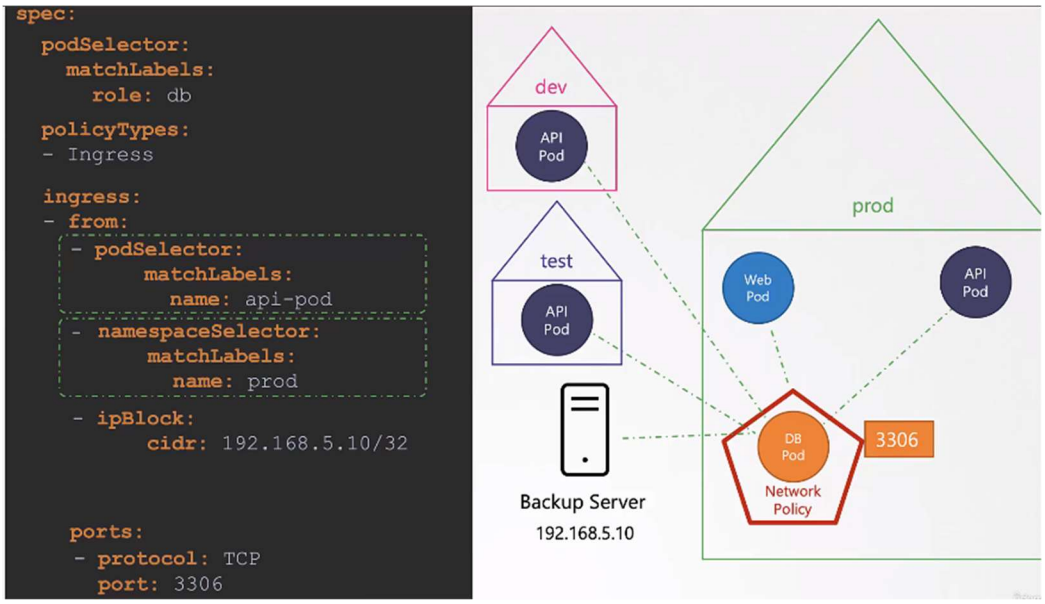
So those are three supported selectors under the **from** section and ingress. And these are also applicable to the **to** section in egress, and we'll see that in a few minutes. We have pod selector to select pods by labels, we have namespace selector to select namespaces by labels, and we have the IP block selector to select IP address ranges.

These can be passed in separately as individual rules or together as part of a single rule. In this example, under the **from** section, we have two elements. So these are two rules. The first rule has the pod selector and the namespace selector together, and the second rule has the IP block selector.

So this works like an OR operation. Traffic from sources meeting either of these criteria are allowed to pass through. However, within the first rule, we have two selectors part of it. That would mean traffic from sources must meet both of these criteria to pass through. So they have to be resonating from pods with matching labels of API pod, and those pods must be in the prod namespace. So it works like an AND operation.



Now what if we were to separate them by adding a dash before the namespace selector like this? Now, they are two separate rules. and traffic matching the second rule is allowed, which is from any pod within the prod namespace, that is either from the pod web, and of course, along with the backup server as we have the IP block specification as well. So now we have three separate rules, and almost traffic from anywhere is allowed to the DB pod. So a small change like that can have a big impact. So it's important to understand how you could put together these rules based on your requirements.



So now let's get rid of all of that and go back to a basic set of rules, and we'll now look at egress. So say for example, instead of the backup server initiating a backup, say we have an agent on the

DB pod that pushes backup to the backup server. In that case, the traffic is originating from the database pod to an external backup server.

For this, we need to have egress rule defined. So we first add egress to the policy types and then we add a new egress section to define the specifics of the policy. So instead of from, we now have to under egress. So that's the only difference. Under to, we could use any of the selectors, such as a pod, a namespace, or an IP block selector. And in this case, since the database server is external, we use IP block selector and provide the the CIDR block for the server. The port to which the requests are to be sent to is AT, so we specify AT as the port. So this rule allows traffic originating from the database pod to an external backup server at the specified address.

```yaml
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
          matchLabels:
            name: api-pod
    ports:
    - protocol: TCP
      port: 3306

  egress:
  - to:
    - ipBlock:
          cidr: 192.168.5.10/32
    ports:
    - protocol: TCP
      port: 80
```