**5.3 Enforcing Access-Control Policies at the API Gateway with Open Policy Agent**

In this section, we look at controlling access to the Order Processing microservice by using *Open Policy Agent* (*OPA*) at the API gateway. The API gateway here is acting as a policy enforcement point. OPA is a lightweight general-purpose policy engine that has no dependency on microservices. You can use OPA to define fine-grained access-control policies and enforce those policies at different places in a microservices deployment.

In [chapter 2](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/chapter-2-17.xhtml#ch02), we looked at using OAuth 2.0 scopes to control access to microservices. There we enforced OAuth 2.0 scope-based access control at the service level by modifying the service code, which is not a good practice. Access-control policies evolve as business requirements change--so every time we have to change our access-control policies, changing the microservice code is not a good practice.

OPA helps you externalize access-control policies and enforce them at any point in the request or response path. [Figure 5.13](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/section-53-40.xhtml#fig5-13) illustrates the sequence of events that happens when an API gateway intercepts client requests to apply authorization policies using OPA. The OPA engine runs as a different process, outside the API gateway, and the API gateway connects to the OPA engine over HTTP. Please check [appendix F](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-F-152.xhtml#app-f) for more details on OPA.

Figure 5.13: Sequence of events that happens when an API gateway intercepts client requests to apply authorization policies using OPA

Larger ViewDiagram

Description automatically generated

**5.3.1 Running OPA as a Docker Container**

Starting OPA as a Docker container is the most straightforward and easiest way to get started. As in the Prometheus example, you need Docker installed and running on your machine to try this out, along with the other prerequisites mentioned in [chapter 2](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/chapter-2-17.xhtml#ch02) for running samples in this book in general.

First check out the samples for this section from chapter05/sample03 at <https://github.com/microservices-security-in-action/samples>. As the first step of executing this sample, we need to start the OPA Docker container (if you are new to Docker, refer to [appendix E](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-E-132.xhtml#app-e)). You can do this by using your command-line client to execute the following command from within the chapter05/sample03 directory. This script starts the OPA server on port 8181 and binds it to port 8181 of your local machine. So, make sure that no other process is running on port 8181. If you want to change the port, you can do so by editing the run\_opa.sh file:

\> sh run\_opa.sh

If your OPA container starts successfully, you should see a message in your terminal window as follows. Note that if this is the first time you are running this command, it might take a few minutes for the OPA Docker image to be downloaded from the Docker registry. Subsequent attempts will be much faster than the initial attempt:

{

"addrs":[

":8181"

],

"insecure\_addr":"",

"level":"info",

"msg":"Initializing server.",

"time":"2019-11-04T01:03:09Z"

}

**5.3.2 Feeding the OPA Engine with Data**

Now that our OPA engine is up and running and can be accessed on port 8181, it is time to register the data required for executing policies. As you can see, listing 5.4 is a declaration of a collection of resource paths. These resources represent one or more resources corresponding to the Order Processing microservice. Each resource has an id, path, method, and a collection of scopes that are associated with the resource. The OPA's REST APIs allow registering these types of data sets on it. You can find the content of listing 5.4 in the file chapter05/sample03/order\_policy.json. This data set is a collection of service paths (resources), where each resource declares the scope required for accessing it.

**Listing 5.4: A set of resources in Order Processing microservice, defined as OPA data**

[

{

"id": "r1", ❶

"path": "orders", ❷

"method": "POST", ❸

"scopes": ["create\_order"] ❹

},

{

"id": "r2",

"path": "orders",

"method": "GET",

"scopes": ["retrieve\_orders"]

},

{

"id": "r3",

"path": "orders/{order\_id}",

"method": "PUT",

"scopes": ["update\_order"]

]

*❶ An identifier for the resource path*

*❷ The resource path*

*❸ The HTTP method*

*❹ To do an HTTP POST to the orders resource, you must have this scope.*

You can register this data on the OPA server by running the following curl command from the chapter05/sample03 directory:

\> curl -v -H "Content-Type: application/json" \

-X PUT --data-binary @order\_policy.json \

http://localhost:8181/v1/data/order\_policy

You should see a response with status code 204 if the request was successful. Note that the order\_policy element in the OPA endpoint, after the data element, is important. OPA uses order\_policy to derive the package name for the data you pushed. In this case, the data you pushed to the OPA server is registered under the data.order\_policy package name. You can find more details on this in [appendix F](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-F-152.xhtml#app-f).

To verify that your data has been successfully registered on the OPA engine, you can execute the following curl command. You should see the content of your resource definitions if the request is successful:

\> curl http://localhost:8181/v1/data/order\_policy

Once the OPA engine has been initialized with the dataset required for our policies, the next step is to implement and deploy access-control policies on OPA.

**5.3.3 Feeding the OPA Engine with Access-Control Policies**

Let's see how to deploy authorization policies into the OPA server; these policies check whether a user/system that's accessing a resource with an access token bears the scopes required for accessing that resource. We'll use OPA policies to make authorization checks on the Order Processing microservice.

OPA policies are written using a declarative language called *Rego*. It has rich support for traversing nested documents and transforming data using syntax similar to Python and JSONPath. Each policy written in Rego is a collection of rules that need to be applied on your microservice.

Let's take a look at the policy defined in the following listing, which checks whether a token being used to access the Order Processing microservice bears the scopes required by the microservice.

**Listing 5.5: OPA policy written in Rego**

package authz.orders ❶

import data.order\_policy as policies ❷

default allow = false ❸

allow { ❹

policy = policies[\_] ❺

policy.method = input.method ❻

policy.path = input.path

}

*❶ The package name of the policy*

*❷ Declares the set of statically registered data identified by order\_policy, as in listing 5.4*

*❸ All the requests by default are disallowed. If this is not set and no allowed rules are matched, OPA will return an undefined decision.*

*❹ Declares the conditions to allow access to the resource*

*❺ Iterates over values in the policies array*

*❻ For an element in the policies array, checks whether the value of the method parameter in the input matches with the method element of the policy*

Here, the package declaration is an identifier for the policy. If you want to evaluate this policy against certain input data, you need to make an HTTP POST request to the http://localhost:8181/v1/data/authz/orders endpoint, having the input data as a JSON payload. Here we refer to the policy in the URL by /authz/orders. This is exactly the same as the package declaration of the policy, with the period character (.) being replaced by forward slash (/).

You can find the policy we define in listing 5.5 in the sample03/orders.rego file. We can register this policy in OPA by executing the following command from within the chapter05/sample03 directory:

\> curl -v -X PUT --data-binary @orders.rego \

http://localhost:8181/v1/policies/orders

We can execute the following command to verify that our policy has been registered successfully. If it's successful, you should get the response with the content of your policy:

\> curl http://localhost:8181/v1/policies/orders

**5.3.4 Evaluating OPA Policies**

Once we have the OPA policy engine running with data and policies, we can use its REST API to check whether a given entity is authorized to perform a certain action. To send a request to the OPA policy engine, first we need to create an OPA input document. An input document will let OPA know details of the resource being accessed and details of the user who is accessing it.

Such inputs are provided to the policy so that the policy can compare that with the set of statically defined data to make its decision. These inputs are provided in JSON format from the microservice (or the API gateway) to the OPA engine at the time of serving a business API request. The following listing shows an example of an input document that contains information of a particular request that is being served by the Order Processing microservice.

**Listing 5.6: OPA input document**

{

"input":{

"path":"orders",

"method":"GET",

"scopes":["retrieve\_orders"]

}

This input document tells OPA that the microservice is serving a request on the path orders for an HTTP GET method. And there's a scope named retrieve\_orders that's associated with the user (or the token) accessing the Order Processing microservice. OPA will use this input data and the statically declared data to evaluate the rules declared in its policies.

Let's query the OPA engine by using its REST API to check whether a particular input results in a true or false evaluation. We first evaluate a true case by using the input defined in sample03/input\_true.json. You can evaluate this by executing the following command from the chapter05/sample03 directory:

\> curl -X POST --data-binary @input\_true.json \

http://localhost:8181/v1/data/**authz/orders** -v

This should give you an HTTP 200 OK response with the following response body. This means that the details we used in the input\_true.json file match one of the rules in the policy registered on OPA. Note that, as we discussed before, the OPA endpoint is derived from the package name of the policy we want to evaluate, which is authz.orders (see listing 5.5):

{"result":{"allow":**true**}}

If you execute the same command using the input\_false.json file, you would see a 200 OK response with the following content. This means that you do not have rights to access the given resource with the given scope:

{"result":{"allow":false}}

**5.3.5 Next Steps in Using OPA**

Let's discuss some of the limitations and next steps with respect to the OPA use case we've discussed. You can learn how to address these limitations in [appendix F](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-F-152.xhtml#app-f):

* The connection to the OPA server for evaluating policies is not properly secured. There are multiple options to secure OPA endpoints, which we discuss in [appendix F](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-F-152.xhtml#app-f).
* The OPA server runs as a Docker container, and all the policies and data pushed to the OPA server using APIs will be gone when you restart the server. Once again, in [appendix F](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/appendix-F-152.xhtml#app-f) we discuss how to overcome that.
* In our example, we use only the curl client to evaluate OPA policies against a given request (or an input document). If you would like to engage OPA with the Zuul API gateway, you need to write a Zuul filter, which is similar to the ThrottlingFilter we used in listing 5.1. This filter has to intercept the requests, create an input document, and then talk to the OPA endpoint to see whether the request is authorized.