Rego 101: Introduction to Rego

Written by Derek Lee

November 1, 2021 · 10 mins read

This blog post explores a different paradigm in policy. We're particularly focused on the contents of the Open Policy Agent (OPA) engine. If you're a beginner and want to get started with writing Rego policy as code, you're in the right place.

In this three-part series, we'll get you up to speed on the following:

- Part 1 (this post): Basic Rego concepts and how to get started with OPA
- Part 2: Intermediate Rego syntax
- Part 3: Types of values and sets

What are Rego and OPA?

Rego is a declarative query language from the makers of the Open Policy Agent (OPA) Framework. The Cloud Native Computing Foundation (CNCF) recent OPA as an incubation-level hosted project in April 2019, and the project from including release in 2021.

Rego is used to write policy as code, which applies programming practices such as version control and modular design to the evaluation of cloud and infrastructure as code (IaC) resources.

OPA is the engine that evaluates policy as code written in Rego. It separates decision-making from policy enforcement, which means it delivers a decision on whether a resource is compliant with policy or you don't need to hard-code those checks into your application code.

By separating policy from your software and offloading policy checks to OPA, you add speed and flexibility to the development lifecycle. You can update policies at any time without having to recompile your application or redeploy your service. This enables you to build at scale, increase visibility into your compliance posture, and programmatically enforce policies. Introducing policy as code also has the benefit of shifting cloud and IaC development left, which means introducing it earlier in the lifecycle.

Here at Snyk, we're the creators of Kubernetes Kyverno and OPA Gatekeeper, and behind the scenes, Snyk uses the OPA engine to evaluate Rego policies and return decisions. In fact, Snyk has run more than one billion security rule evaluations using OPA!

How does Rego work?

As a declarative query language, the Rego, you describe the data you want to retrieve, and the program searches a data source / input for a match. It's a match. This is different from traditional imperative languages, where you describe the steps that need to happen to produce a result. Perhaps you are already familiar with some declarative query languages - SQL is probably the most widely used one.

With Rego, you describe the conditions for passing or failing policy, and the OPA searches a JSON (or YAML) input document for data that matches those pass/fail conditions.

Sounds great in the abstract, but what does a rule look like? Let's take a look at how to enforce a corporate policy that only Alice, a network administrator, should have permission to create and delete virtual networks in the production environment. Here's an example rule we could write:

```
allow := true {
  input.user == "alice"
}
```

Don't worry about the syntax for Rego, you describe the data you want to retrieve, and the program searches a data source / input for a match. It's a match. This is different from traditional imperative languages, where you describe the steps that need to happen to produce a result. Perhaps you are already familiar with some declarative query languages - SQL is probably the most widely used one.

With Rego, you describe the conditions for passing or failing policy, and the OPA searches a JSON (or YAML) input document for data that matches those pass/fail conditions.

Sounds great in the abstract, but what does a rule look like? Let's take a look at how to enforce a corporate policy that only Alice, a network administrator, should have permission to create and delete virtual networks in the production environment. Here's an example rule we could write:

```
allow := true {
  input.user == "alice"
}
```

Don't return to this rule momentarily as we can explain what it does and how it works. For now, just marvel at its elegant simplicity!

Input

OPA can process any JSON or YAML document as input. Did you notice we used `input` in the example rule above? `input` is treated as a special JSON document that can be accessed globally, meaning you can refer to it from anywhere within the Rego policy.

Here's an example input document to go with our example rule from the previous section:

```
{
  "user": "alice"
}
```

Don't say that this is a document representing the currently logged-in user. In the real world, this input document might be a Kubernetes manifest or the output of a Terraform plan. We'll show you an example in an upcoming blog post.

Rules

Now that we've shown you what input looks like let's dive into the concept of rules. In the Rego language, a rule is a conditional assignment. Each rule has two parts:

```
<rule_name>
<expression>
```

- The head consists of a variable and a value that may be assigned to it.
- The body consists of one or more queries that will OPA will attempt (successfully) to find a match for the value to be assigned to the variable.

You can read a rule like this:

```
THE VARIABLE <rule_name> HAS THE VALUE <true>
IF THERE CONDITIONS ARE MET
```

In such, a rule queries the input to find a match for a condition, and if a match is found, a value is assigned to a variable.

Here's the example rule we used earlier:

```
allow := true {
  input.user == "alice"
}
```

In the example above, the head (variable and value) is `allow := true` and the body (query) is `input.user == "alice"`. Put the head and body together, and you get a complete rule, which can be read as:

The variable `allow` has the value `true` IF `input` is equal to `"alice"`

Note that in Rego, `:=` is the assignment operator, sometimes known as the walrus operator. It simply assigns a value to a variable, much like the equals sign does in other languages.

Queries

Let's dig into the concept of queries. As we stated earlier, a query represents a condition to check – it's essentially the first half of an if statement.

For example, this is the query from the example rule in the previous section:

```
input.user == "alice"
```

This tells OPA (Rego) to query the `input` document to find out if the `user` is equal to `"alice"`.

There's some nuance here – Rego is declarative, so a query is technically just making the statement "This is true if `x` and OPA finds all the values in the input that make that statement true. As in the example, the query says, The `user` property is set to `"alice"`. The OPA will examine the `user` property and find all the users in the input that match the statement `"alice"`. If this was true, it's looking for the value `"alice"`.

Referring to input in a query

When you craft a Rego query, you can dot-notation to drill down to the property you're looking for, meaning each nested level of the input document is separated by a period. First, you start with `input`, and then add a dot and the name of the property at the top level in the JSON. `input.user`

To refer to a nested property, you add a period for each level of the nested JSON. For example, if `user` is nested, and the top level property (`input.user.property.foo`), then keep adding dots and property names until you reach the nested property you want to query. If the property you want to query is an array, all right – we'll address that momentarily.

Let's say the input document looks like this, instead:

```
{
  "admin": {
    "user": "alice"
  }
}
```

Since `user` is nested under `admin`, which is nested under `input`, we'd refer to it like so:

```
input.admin.user
```

If the input property you're referring to is an `array`(s), on the other hand, you will use the wildcard operator – an underscore `_` to specify the property. For example, let's say the input includes an `user` array:

```
{
  "users": [{"name": "alice", "age": 30}, {"name": "bob", "age": 25}]
}
```

In this case, if you wanted to find out if `alice` is in the `user` array, you'd use this syntax:

```
allow := true {
  input.users[_] == "alice"
}
```

Above, the wildcard operator OPA OPA is looking through the array and see if any of the elements is equal to `alice`. We'll dive into iteration in a future blog post.

Likewise, let's suppose you want to check whether the value of the `admin` property is `true` or any of the elements in the `users` array below (even though there's only one element above):

```
{
  "admin": {
    "name": "alice",
    "admin": true
  }
}
```

```
allow := true {
  input.admin.admin == true
}
```

Of course, `if` conditions aren't very useful unless there's a conditional action to go with it. That's where it's useful to understand how assignment works in Rego – the second half of the `if` statement.

Rule evaluation

The conditional action in a rule is assigning a variable. In our example, we are looking in the input to find out if the variable `allow` should be assigned the value `true`. You can read it like this:

```
allow := true
IF THERE IS A CONDITION IS MET
```

What are variables?

A variable in a reference is a specific value. Here, the variable `i` is assigned the value `1`.

```
a = 1
```

You can use the variable in place of the value after that, and for all the same to Rego.

```
a = 1
b = 2
c = a + b
```

Connect to our example, the variable `allow`, and the value to be assigned is `true`.

```
allow := true
```

Combined with the condition (query) we discussed earlier, you get the entire rule:

```
allow := true {
  input.user == "alice"
}
```

To review, you'd read this like this:

The rule `allow` has the value `true` IF `input.user` is equal to `"alice"`.

In practice, this kind of rule is very common, so Rego allows you to abbreviate it to the following:

```
allow := true {
  input.user == "alice"
}
```

Testing non-compliant

Let's say we want to see what happens when a non-compliant input is used. You can find instructions for installing the `k8-OPA-documentation`. Once you install it, you'll need two things:

- A `.rego` policy file containing your rule, along with a package declaration such as `package kube.check_user`
- A `.yaml` file containing the input you'd like to test against.

Create a `.yaml` file containing the input, and as mentioned we issue the `test` command.

Once you have those two things, you can use the `opa eval` command to evaluate your policy as code:

```
$ opa eval -i input.yaml -d check_user.rego "data.kube.check_user"
```

You can name your files whatever you want – just make sure the command follows this structure:

```
$ opa eval -i <input file> -d <rule file> "data.<package name>"
```

If `allow` evaluates to `true`, as it does in our example, you'll see the same output you saw in the Rego Playground:

```
{
  "kube": true
}
```

Again, OPA is indicating that the input is consistent with – compliant with – our company policy, which means the currently logged-user is allowed to create and delete virtual networks in the production environment.

Testing non-compliant input

What does it look like if an input document is non-compliant? In the playground or in your local `.rego` file, change the input to the following:

```
{
  "user": "bob"
}
```

Now, when you evaluate the rule (by clicking Evaluate in the Rego Playground, or running the aforementioned `opa eval` command), you see the following output:

```
{
}
```

What does it mean? OPA returns an "undefined" result (i.e. an empty set) in this example because it does not find a value in the input that matches the condition `input.user == "alice"`. The rule is stating because that one is results, the `allow` rule does not evaluate to `true`, and the input document is not compliant with company policy. Sorry, Bob!

What's next?

Be sure to return to our blog for the rest of our Rego for Beginners content series, where we'll explore intermediate Rego rule syntax, including AND and OR structures, custom messages, special keywords, and more.

In the meantime, here are some resources:

- OPA's website
- Rego policy reference
- Rego Playground
- Testing Rego as a security policy language
- Enabling Policy as Code with OPA and Rego
- This Rego Fun series Rego Rego language for OPA

If you're interested in using Rego to write custom rules for Snyk IaC, check out our documentation here. In addition to Snyk's out-of-the-box security & compliance-focused rulesets, our custom rules enable you to set customized security controls across your IaaS.

With Snyk you can scan single-item and controls for your configuration issues from code to cloud with an issues UI, ruleset, and policy engine spanning IDE, SCM, CI, CI/CD, Terraform Cloud, and deployed cloud environments such as AWS, Azure, and Google Cloud.

IaC security designed for devs

Scan and secure your infrastructure as code from IDE to runtime in the cloud with a unified policy as code engine so your teams can develop, deploy and scale.

Book a live demo

Posted in: Cloud Security, IaC Security

Snyk Top 10: Vulnerabilities you should know

Find out which types of vulnerabilities you most likely to encounter, based on Snyk's vulnerability database and security research.

See the report