Rego 102: Combining queries with AND/OR and custom messages

Written by Frank Van der Kogt Snyk Lab

November 9, 2021 · 15 mins read

This blog post series offers a gentle introduction to Rego, the policy language from the creators of the Open Policy Agent (OPA) project. If you're a beginner or want to get started with writing Rego policies as code, you're in the right place.

In this three-part series, we'll go over the following topics:

- Part 1: Rego for beginners: Introduction to Rego
- Part 2: Combining queries with AND/OR
- Part 3: Types of values and rules

As a reminder, Rego is a declarative query language from the makers of the Open Policy Agent (OPA) framework (Cloud Native Computing Foundation (CNCF) project). OPA can be encapsulated into a sidecar/proxy/agent (OPA and Envoy) and embedded from including where in 2021.

Rego is used to write policies as code, which applies programming practices such as version control and modular design to evaluate cloud and infrastructure as code (IaC) resources. OPA is the engine that evaluates policy as code written in Rego. And that uses the Rego language for creating rules.

Part recap

In Part 1 of this blog post series, we explained that a Rego rule is a conditional assignment. A rule queries the input to find a match to its conditions, and if a match is found, a value is assigned to a variable.

You can read a rule like this:

```
# this variable is set to true if
# input.user is "alice"
allow := true {
    input.user == "alice"
}
```

Here's the example we used, which represents a corporate policy that only lines, a network administrator, should have permission to create and delete virtual networks in the pod account:

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

# allow := true if
# input.user == "bob"
allow := true {
    input.user == "bob"
}
```

OPA evaluates a JSON or YAML input document against a rule to produce a policy judgment. The input document below represents the currently logged-in user:

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

If you use OPA to evaluate the input against the rule, it finds a match for its query `input.user == "alice"`. Therefore, the variable in the rule head, `allow`, is assigned the value in the rule head, `true`. Here's the output showing this:

```
{
    "allow": true
}
```

OPA has delivered the decision that Alice, the currently logged-in user, is allowed to create and delete virtual networks in the pod account. The logic is complicated with the rule.

and and or

So far, we've only shown rules with a single query. A rule can also contain **multiple queries**. If it does, the queries represent multiple conditions that must unfold what in order for a variable to be assigned. There's a logical **AND**. "The condition must be met **AND** the condition must be met."

For example, in the rule below, both `input.user == "alice"` **AND** `input.environment == "prod"` must be true in order for the variable `allow` to be assigned the value true:

```
# allow := true if
# input.user == "alice"
# input.environment == "prod"
allow := true {
    input.user == "alice"
    input.environment == "prod"
}
```

In order cases, **OR** might be more appropriate. You can represent **OR** by using the same head in multiple rules:

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

# allow := true if
# input.user == "bob"
allow := true {
    input.user == "bob"
}
```

The set of rules can read like this:

`allow` is true if user is "alice" **OR** user is "bob".

Technically, the set of rules forms a single rule because the head is the same for both. Because you're defining rules with multiple steps, it's called an **incremental rule**.

If you like, you can get rid of the second head and put the bodies together. It looks succinct way of writing the above as:

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

You might be wondering why we've used the **assignment operator** in rather than the **comparison operator** in. That's because variables are immutable in Rego. Even though rules with the same head are treated as a single incremental rule if you try to use the assignment operator, you're effectively "assigning" the same variable multiple times — and that will attempt to Rego. Instead, we use the unification operator in the rule head because it unifies multiple values with the same name.

If it's confusing to remember when to use which operator in the rule head, there's a simple way. Rego by default values and a lot of syntactic sugar.

Default values in rule heads

The default value given to a variable in the head of a rule is `true`. So, Rego offers some syntactic sugar here. When a rule assigns the value `true`, to the variable, you can omit the `:= true` from the rule head.

This creates the **shortcut**:

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

# allow := true if
# input.user == "bob"
allow {
    input.user == "bob"
}
```

...is the same as this **AND** one:

```
# allow := true if
# input.user == "alice"
# input.environment == "prod"
allow {
    input.user == "alice"
    input.environment == "prod"
}
```

and shortcut, this **OR** one:

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

# allow := true if
# input.user == "bob"
allow {
    input.user == "bob"
}
```

...is the same as this **OR** rule:

```
# allow := true if
# input.user == "alice"
# input.environment == "prod"
allow {
    input.user == "alice"
    input.environment == "prod"
}
```

"based" instead

default keyword

As we discussed in Part 1, if there are no matches in the input for a rule query, the variable in the rule head is not assigned the value in the head.

To demonstrate this, let's return to our example rule, which says that only Alice, a network administrator, should have permission to create and delete virtual networks in the pod account:

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

And we'll say no human or input document where the currently loggedin user is Bob:

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

Since `input.user` is not `"alice"`, when we evaluate the rule against the input, OPA does not find a match in the input. Therefore, `allow` is not assigned the value `true`, and the result of the evaluation is an empty set:

```
{}
```

You might be wondering why we've used the **unification operator** in rather than the **assignment operator** :=. That's because variables are immutable in Rego. Even though rules with the same head are treated as a single incremental rule if you try to use the assignment operator, you're effectively "assigning" the same variable multiple times — and that will attempt to Rego. Instead, we use the unification operator in the rule head because it unifies multiple values with the same name.

If it's confusing to remember when to use which operator in the rule head, there's a simple way. thanks to default values and a lot of syntactic sugar.

Default values in rule heads

The default value given to a variable in the head of a rule is `true`. So, Rego offers some syntactic sugar here. When a rule assigns the value `true`, to the variable, you can omit the `:= true` from the rule head.

This creates the **shortcut**:

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

# allow := true if
# input.user == "bob"
allow {
    input.user == "bob"
}
```

...is the same as this **AND** one:

```
# allow := true if
# input.user == "alice"
# input.environment == "prod"
allow {
    input.user == "alice"
    input.environment == "prod"
}
```

and shortcut, this **OR** one:

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

# allow := true if
# input.user == "bob"
allow {
    input.user == "bob"
}
```

...is the same as this **OR** rule:

```
# allow := true if
# input.user == "alice"
# input.environment == "prod"
allow {
    input.user == "alice"
    input.environment == "prod"
}
```

"based" instead

default keyword

As we discussed in Part 1, if there are no matches in the input for a rule query, the variable in the rule head is not assigned the value in the head.

To demonstrate this, let's return to our example rule, which says that only Alice, a network administrator, should have permission to create and delete virtual networks in the pod account:

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

And we'll say no human or input document where the currently loggedin user is Bob:

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

Since `input.user` is not `"alice"`, when we evaluate the rule against the input, OPA does not find a match in the input. Therefore, `allow` is not assigned the value `true`, and the result of the evaluation is an empty set:

```
{}
```

Now, if OPA determines that `input.user` is not `"alice"`, `allow` does not evaluate to an empty set. Instead, it takes on the default value, which we've declared in `false`:

```
# default allow := false
default allow := false
```

Note that when you specify the `default` keyword, you use the unification operator := instead of the assignment operator :=. It's a bit like rule where you define the default value and the rule body, which uses the unification operator, assigns. Again, that's because variables are immutable. If you try to use the assignment operator, you're "assigning" the same variable multiple times, which Rego doesn't allow. You can the unification operator instead:

```
# default allow := false
default allow := false
```

You can, of course, take advantage of the syntactic sugar we described earlier and leave out the `:= true` in the rule with the conditional assignment. This is perfectly acceptable and perhaps easier to see because you don't need to remember which operator to use in the conditional assignment:

```
# default allow := false
default allow := false
```

Custom messages

Sometimes, you want to return a series of messages rather than a single query or `false`. `true` or `false` conditional statements are not sufficient for your use case. Rego's `contains` keyword allows you to return a set of messages. Plus the rule's structure is the same as before, and it looks like a rule. In the example, the policy "Deny a network access" is added to the rule `deny`, and such that more about rules in Part 1:

```
# deny[msg] {
# input.user == "bob"
# msg := "Deny: No network access"
deny[msg] {
    input.user == "bob"
    msg := "Deny: No network access"
}
```

When we evaluate the rule against the input devices using OPA's Rego Playground in the `input.user == "bob"` section, we get the following output: `{"deny": ["Deny: No network access"]}` command (for instructions, see in Part 1). Using jq tool, the result looks like this:

```
{
    "deny": [
        "Deny: No network access"
    ]
}
```

What's happening here? We're actually creating a **set** rule, a concept we'll return to in our next blog post in this series. Sets are collections of unique values. The `deny` rule's head contains a variable, `msg`, which is assigned to the variable `msg`. Rego `msg` is and are accumulated into of unique elements, such as integers `{1, 2, 3}` or strings `{"alice", "bob", "carlotta"}` or even other sets `{1, {2, 3}, 4}`. We can make a set out of **non-assorted Rego type**, or even mix and match types within a single set.

In the instance of our single rule with element in the `deny` set is a string containing a message. There's only one element in the set this particular input document, but there can be more, and we'll show you an example later in this blog post.

What if we want to return additional information in the message? We can use the **sprintf** built-in function to display the value of the `input.user` field that caused a deny result:

```
# deny[msg] {
# input.user == "bob"
# msg := sprintf("Deny: No network access", [input.user])
deny[msg] {
    input.user == "bob"
    msg := sprintf("Deny: No network access", [input.user])
}
```

The `sprintf` function takes two arguments — a string and an array of values. In this case, the only element in the array is a string representing the `input.user`. We use `%s` as a placeholder in the first argument, and the value in the array takes its place when the rule is evaluated.

Now, if we evaluate the rule using the following input:

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

...we see this result:

```
{
    "deny": [
        "Deny: Bob is denied access"
    ]
}
```

The some keyword

Let's supplement with the example above to demonstrate the `some` keyword so that it means the opposite. Most of the time, you'll want to use the `in` to query to specify the absence of a property from the input. As, for example, this query:

```
# input.tags.environment
```

...means "The input document has a `tags.environment` property and this value is not `"Prod"`, and this query:

```
# not input.tags.environment
```

...means "The input document does not have a `tags.environment` property." Rego's `some` keyword is used in the `some` "There's no overlap or middle ground" — an expression and its inverse are mutually exclusive.

Here's an example rule that assigns `true` to `deny` if the input does not have a `tags.owner` key:

```
# deny := true if
# input.tags.owner
deny := true {
    input.tags.owner
}
```

Let's use this input:

```
{
    "tags": {
        "environment": "staging"
    }
}
```

If we were to evaluate this input against the rule above, we'd see that `deny` returns `true` because it is missing the required `tags.owner` property:

```
{
    "deny": true
}
```

Evaluating an example rule with OPA

Let's supplement with the concepts we've discussed in this blog post by evaluating an example rule. In Part 1, we will focus on two ways of interacting with OPA:

- Using the Rego Playground
- Using OPA's command line tool

For instructions on using these interfaces, see Part 1.

This time, we're using more of a real-world example involving a Kubernetes pod. Here's the JSON manifest we will use as input:

```
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "name": "myapp-pod",
        "labels": {
            "app": "myapp"
        }
    },
    "spec": {
        "containers": [
            {
                "name": "myapp",
                "image": "busybox:1.28.4",
                "command": [
                    "sleep",
                    "1000000"
                ]
            }
        ]
    }
}
```

And here's the rule we'll be evaluating it against, which we've written to enforce the company policy "Kubernetes pods must be labeled with release and environment":

```
# deny[msg] {
# input.kind == "Pod"
# not input.metadata.labels.release
# msg := sprintf("Deny: Missing release label", [input.metadata.name])
# deny[msg] {
# input.kind == "Pod"
# not input.metadata.labels.environment
# msg := sprintf("Deny: Missing environment label", [input.metadata.name])
deny[msg] {
    input.kind == "Pod"
    not input.metadata.labels.release
    msg := sprintf("Deny: Missing release label", [input.metadata.name])
}
deny[msg] {
    input.kind == "Pod"
    not input.metadata.labels.environment
    msg := sprintf("Deny: Missing environment label", [input.metadata.name])
}
```

This rule demonstrates some concepts we've discussed in this blog post:

- `deny[msg]` is shorn a set of custom messages instead of a `true` or `false` conditional
- Both **AND** and **OR** rule structures:
  - Deny if the Kubernetes object is a pod **AND** it is missing the release label **OR**
  - Deny if the Kubernetes object is a pod **AND** it is missing the environment label
- The `not` keyword is shorn for the absence of a property
- The `sprintf` function is shorn a message that lists the name of the misconstrued pod

For your convenience, we've created a playground with this content already:

https://play.openpolicyagent.org/p/WRXRfNbJ7

If you evaluate the rule by selecting the **Evaluate** button in the playground or by executing a command such as `opa eval -i input.json -d check.rego 'data.play.deny[msg]'` (Linux/macOS), if running OPA locally, you'll see the output:

```
{
    "deny": [
        "Deny: Missing release label",
        "Deny: Missing environment label"
    ]
}
```

Or, if you evaluate the rule against the input devices, you'll see the following output:

```
{
    "deny": [
        "Deny: Missing release label",
        "Deny: Missing environment label"
    ]
}
```

Finally, let's add in a `labels.release` and `labels.environment` property to the input, so it looks like this:

```
{
    "labels": {
        "release": "myapp",
        "environment": "prod"
    }
}
```

What happens if we evaluate the rule again? We see that the `deny` set is empty:

```
{
    "deny": {}
}
```

This means our pod is compliant because OPA did not add any messages to the `deny` set. Hooray!

What's next?

Be sure to return to our blog to read Rego for Beginners Part 3, where we'll explore set rules, object rules, functions, and iteration.

In the meantime, here are some resources:

- OPA website
- Rego policy reference
- Rego Playground
- Open Rego as a generic policy language
- Building Policy as Code with OPA and Rego
- Five tips for using the 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 ground-to-sky and compliance-mapped objects, our custom rules enable you to set unprecedented security controls across your IaCs!

IaC gives you a single view and controls for your configuration issues from code to cloud with an issues UI, dashboard, and policy engine spanning AD, SCM, CLI, CI/CD, Terraform Cloud, and deployed cloud environments such as AWS, Azure, and Google Cloud.

Snyk Top 10: Vulnerabilities you should know

Find and remediate top-of-mind vulnerabilities that pose a threat to open projects based on exploitability and severity. Watch now.

Watch the Webinar

Snyk Top 10: Vulnerabilities you should know

Find and remediate top-of-mind vulnerabilities that pose a threat to open projects based on exploitability and severity. Watch now.

Watch the Webinar

Posted in: IaC Security, Cloud Security

Snyk is a developer security platform. Integrating directly into development tools, workflows, and automation pipelines, Snyk makes it easy for teams to find, prioritize, and fix security vulnerabilities in code, dependencies, containers, and infrastructure as code. Supported by industry-leading application and security intelligence, Snyk puts security expertise in any developer's toolkit.

Start free

Book a live demo

Product

What is Snyk?
Snyk Code (SAST)
Snyk Open Source (SCA)
Snyk Container
Snyk Infrastructure as Code
Snyk AppRisk (ASPM)
Developer Security Platform
Application security
Software supply chain security
Secure AI-generated code
DeepCode AI
Pricing
Deployment options
Integrations
IDE plugins
Git Security
CI/CD pipelines security
Snyk CLI
Snyk Learn
Snyk for JavaScript

Resources

Documentation
Snyk API Docs
API status
Disclosed vulnerabilities
Support portal & FAQ's
Blog
Security fundamentals
Resources for security leaders
Resources for ethical hackers
Vulnerability Database
Snyk OSS Advisor
Snyk Top 10
Videos
Customer resources

Company

About
Customers
Careers
Events
Snyk for government
Press kit
Security & trust
Legal terms
Privacy
For California residents: Do not sell my personal information
Website Terms of Use

Connect

Book a live demo
Contact us
Support
Report a new vuln

Security

Application Security
Container Security
Supply Chain Security
JavaScript Security
Open Source Security
AWS Security
Secure SDLC
Security posture
Secure coding
Ethical Hacking
AI in cybersecurity
Code Checker
Python
Enterprise Cybersecurity
JavaScript
Snyk With GitHub
Snyk vs Veracode
Snyk vs Checkmarx

© 2024 Snyk Limited
Registered in England and Wales