

# Управляющие конструкции в коде Terraform

Елисей Ильин  
DevOps-инженер в Itransition



# Елисей Ильин

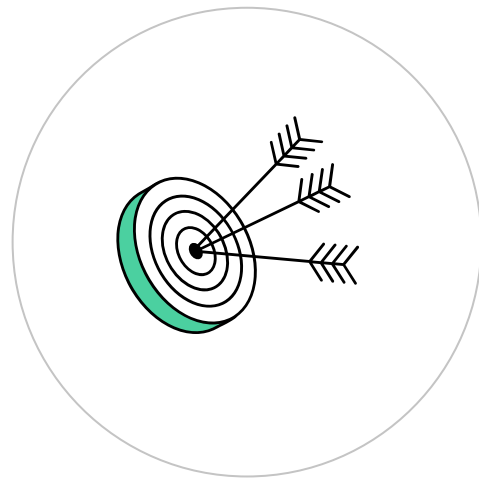
О спикере:

- DevOps-инженер
- опыт работы в IT 6 лет



# Цели занятия

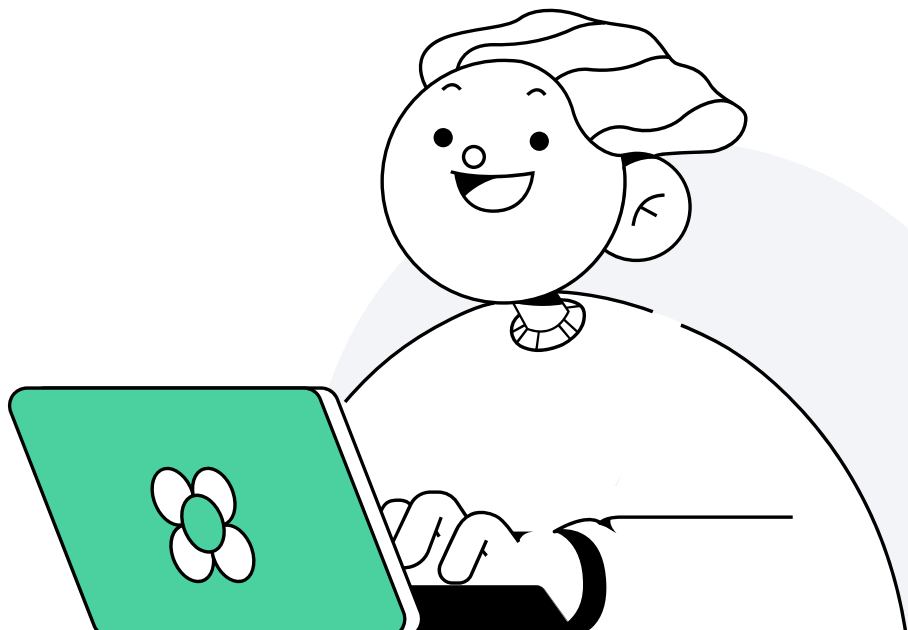
- Сделать код более динамичным, добавив в него логику
- Научиться дополнительно настраивать ресурсы, созданные Terraform



# План занятия

- 1 Метааргументы
- 2 Expressions
- 3 Provisioners
- 4 Итоги занятия
- 5 Домашнее задание

\*Нажми на нужный раздел для перехода



# Метааргументы



1



**Метааргументы — специальные аргументы, которые можно использовать в Terraform, чтобы настраивать поведение ресурсов и провайдеров**

# depends\_on

Terraform обычно строит правильную последовательность создания ресурсов, чтобы удовлетворить их зависимости друг от друга.

Однако в некоторых случаях возможны исключения и ошибки, которые могут привести к неправильному порядку создания ресурсов.

Аргумент **depends\_on** в блоках позволяет управлять порядком создания **ресурсов и модулей**.

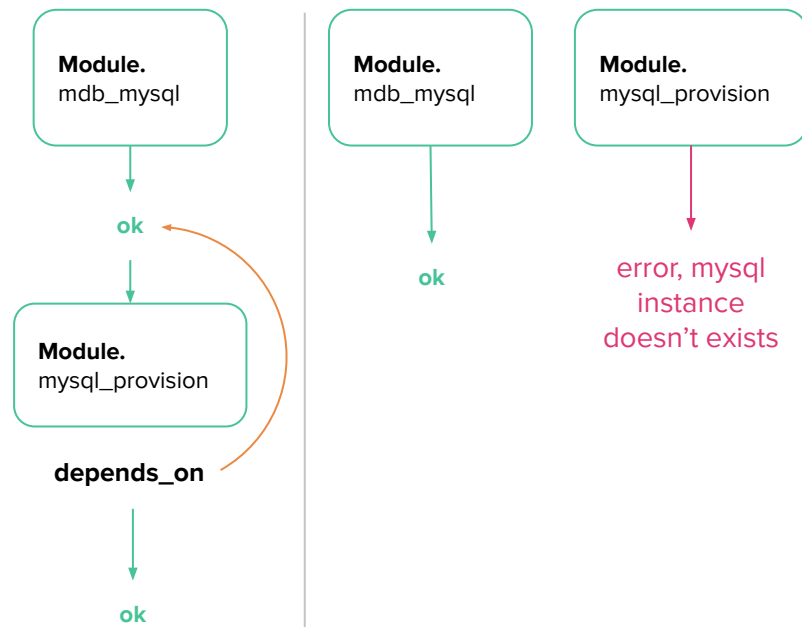
**depends\_on = [ resource.A , module.B, ...]**

# depends\_on

Обычно **depends\_on** используют **только** для определения порядка создания **child modules**. Работу с модулями мы изучим в следующей лекции:

```
#Объявление блока модуля
module "mdb_mysql" {
  instance_name = "test_instance"
}

module "mysql_provision" {
  depends_on = [module.mdb_mysql]
  database   = "test_db"
  user_name  = "test_user"
  instance_id = module.mdb_mysql.id
}
```

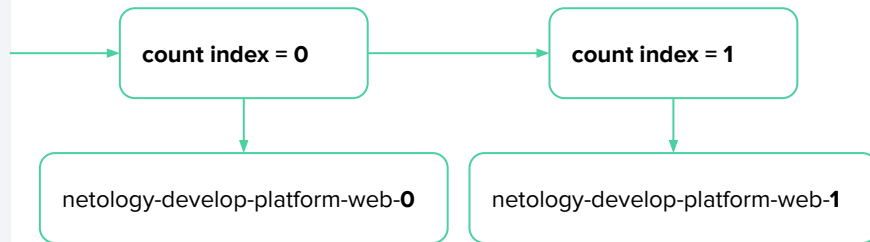




# count loop

- Позволяет указать количество экземпляров этого ресурса, которые необходимо создать
- Инициализирует итерируемую переменную **count.index**
- Подходит для создания **идентичных** ресурсов
- Если требуется создать отличающиеся ресурсы, стоит использовать метааргумент **for\_each**

```
resource "yandex_compute_instance" "web" {  
  count = 2  
  name =  
  "netology-develop-platform-web-${count.index}"  
  ...  
}
```



# for\_each loop with set

В отличие от count, в качестве указателя количества экземпляров принимает переменную типа **set** или **map**.

Доступны атрибуты:

- **each.key**
- **each.value**

**В случае set** each.key==each.value

```
resource "yandex_compute_instance" "web" {  
  for_each = toset([ 0, 1 ])  
  name =  
  "netology-develop-platform-web-${each.key}"  
}
```



```
"netology-develop-platform-web-0"  
"netology-develop-platform-web-1"
```

# for\_each loop with map

```
resource "yandex_compute_instance" "web" {  
  for_each = {  
    0 = "first"  
    1 = "second"  
  }  
  name = "netology-develop-platform-web-${each.key}"  
  tags = {  
    Name = "${each.value}"  
  }  
}
```



```
"netology-develop-platform-web-0" tags=["first"]  
"netology-develop-platform-web-1" tags=["second"]
```

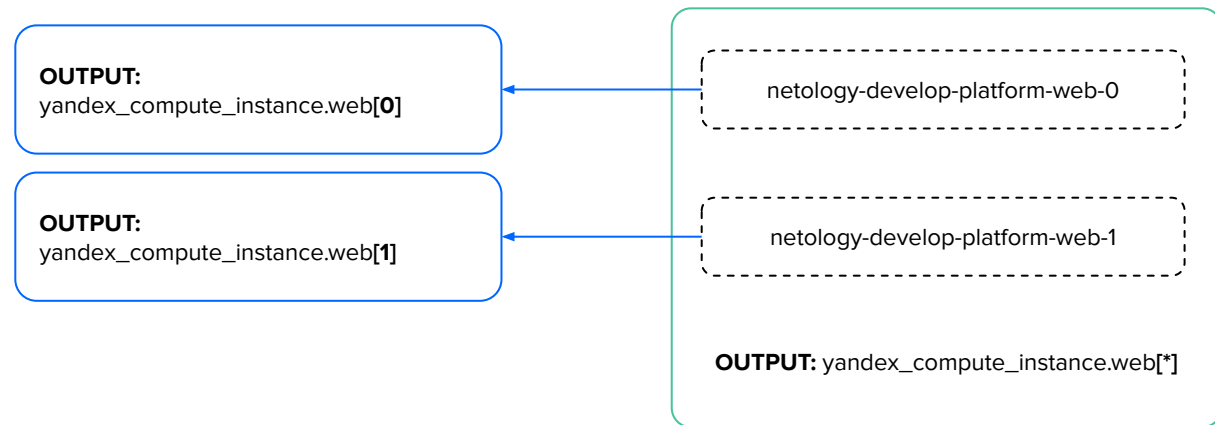
# Обращение к ресурсам при использовании count или for\_each

**Для конкретного ресурса используется индекс:**

yandex\_compute\_instance.web[0]

**Для всех:** yandex\_compute\_instance.web[\*] **или**

yandex\_compute\_instance.web.\*



# lifecycle

Метааргумент lifecycle позволяет изменить поведение при внесении изменений в ресурсы:

- **create\_before\_destroy** = true
- default = false
- **prevent\_destroy** = true, защита от случайного удаления
- **ignore\_changes** = [ tags ], игнорировать изменения, например, тегов. Полезно, если тегами управляет стороннее ПО

```
resource "yandex_compute_instance" "platform" {  
  name          = "netology-develop-platform-web"  
  ...  
  lifecycle {  
    prevent_destroy = true  
  }  
}
```

# provider

Позволяет **переопределить настройки** провайдера для выбранного ресурса.

**Обычно** этот аргумент имеет смысл использовать только в мультирегиональных облаках, таких как **aws, gcp, azure, digital\_ocean, selectel**. В них создаётся ресурс, чтобы выбрать регион.

В **Yandex Cloud** на сейчас только один регион

```
provider "aws" {  
  alias = "eu-west-1"  
  region = "eu-west-1"  
  access_key = "var.aws_ak_west"  
  secret_key = "var.aws_sk_west"  
}  
  
provider "aws" {  
  alias = "eu-central-1"  
  region = "eu-central-1"  
  access_key = "var.aws_ak_eu"  
  secret_key = "var.aws_sk_eu"  
}  
  
resource "aws_instance" "ec2_eu_central1" {  
  provider = aws.eu-central-1  
  ami = "ami-0ff338189efb7ed37"  
  instance_type = "t2.micro"  
  count = 1  
}
```

#В примере показан выбор провайдера по alias

# Expressions

Выражения



2

# Simple expressions

Выражения — **значения**, вычисляемые в процессе выполнения кода. Они позволяют сделать код более гибким:

- любой тип данных terraform
- индексы и атрибуты
- ссылка на именованные значения
- арифметические и логические операторы
- интерполяция строк
- строки а here document (HereDoc)

- `list[5], map["key"]`
- `var.<NAME>, <RESOURCE TYPE>.<NAME>`
- `5+5, a!=b,a==b,a >=5, &&, ||`
- `${...}`

```
block {  
  value = <<-EOL  
  hello  
    world  
  EOL  
}
```

**Результат:**  
hello  
world



# Functions

Язык HCL не позволяет добавлять пользовательские функции, но предоставляет множество встроенных.

## Вызов функции:

```
<FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>...<ARGUMENT N>)
```

## Примеры:

```
>join( ",", ["Hello ", "world ", "!" ] )
```

```
>split( "_", "A_B_C_D" )
```

```
>concat( [ 1,2,3 ], [ 4,5,6 ] )
```

```
>merge( { "1": "A " }, { "2": "B" } )
```

**Результат:** "Hello world!"

**Результат:** [ "A", "B", "C", "D", ]

**Результат:** [ 1, 2, 3, 4, 5, 6, ]

**Результат:** { "1" = "A" , "2" = "B" }

# Functions

## Виды функций:

- числовые
- строковые
- коллекции
- дата и время
- хеш и шифр
- сеть IP
- файловая система
- кодирование
- преобразование типов данных

# Условные выражения

Используют для логического выбора между двумя значениями.

## Синтаксис:

```
условие ? истинное значение : ложное значение
```

## Пример:

```
mysql_hosts_count = var.env_name == "production" ? 3 : 1
```

Кроме того, их используют в метааргументе **count**, чтобы создать ресурсы по условию.

## Пример:

```
count = var.bastion_instance == true ? 1 : 0
```



**При работе с данными, например со списком чисел [1, 2, 3, 4, 5], вам может понадобиться выполнить одну и ту же операцию для каждого элемента списка.**

Допустим, вам нужно вывести **каждое** число на экран, при этом оно должно быть умножено на 2.

Для этого в программировании используют **итерацию в цикле** — процесс, когда вы **последовательно** повторяете одну и ту же операцию для каждого элемента списка, начиная с его первого элемента и заканчивая последним



**for loop — цикл, который позволяет  
итерироваться по содержимому list и map,  
применяя к нему функции, условные  
выражения, преобразование данных**

# Итератор

Это объект, который по очереди принимает значение каждого элемента списка, чтобы выполнить с ним запрограммированное действие. В примере итератором является **num**:

```
locals {  
  numbers = [ 1, 2, 3, 4, 5 ]  
}  
  
output "doubled_numbers" {  
  value = [ for num in local.numbers: num * 2 ]  
}
```

**Результат:**  
**doubled\_numbers** = [  
 2,  
 4,  
 6,  
 8,  
 10,  
]

# for loop

**Для list:** [ for <ITEM> in <LIST> : <OUTPUT\_VALUE> ]

**Пример:** test\_list = ["develop", "staging", "production"]

```
[for env in local.test_list : upper(env) if env != "develop" ]  
[  
    "STAGING",  
    "PRODUCTION",  
]
```

**Для map:** { for <KEY>, <VALUE> in <MAP> : <OUTPUT\_KEY> <OUTPUT\_VALUE> }

**Пример:** test\_map = { John = "admin", Alex = "user" }

```
> [for k,v in local.test_map : "${k} is ${v}" ]  
[  
    "Alex is user",  
    "John is admin", ]
```

# Directives

Конструкция `%{ ... }` позволяет итерироваться по `list`, `map` или `set`.

Поддерживает функции, выражения и условную логику.

## Синтаксис:

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor}
```

В строковой директиве можно указать маркер `~`, чтобы удалить все пробелы и переносы строки:

- в начале `%{~ .. }`
- в конце `%{ .. ~}`
- в начале и в конце `%{~ .. ~}`



# Directives

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor}
```

Пример:

```
locals{  
  test_list = ["develop", "staging", "production"]  
}
```

```
> "%{ for env in local.test_list}**${upper(env)}_!%{endfor}"
```

```
"**DEVELOP_!**STAGING_!**PRODUCTION_!"
```



**Dynamic blocks используют для динамической генерации многократно повторяющихся, вложенных блоков**

# Dynamic blocks

Рассмотрим пример создания группы безопасности в YC (firewall для ресурсов). Внимание: **сервис находится на стадии Preview**.

Нужно создать **отдельный блок ingress** (входящее правило) или **egress** (исходящее правило) для **каждой** записи.

## Недостатки:

- 1 Правил может быть **огромное** количество
- 2 Со временем количество правил может изменяться, это потребует править код
- 3 Для каждого окружения (dev, prod) придётся хардкодить свой код

На следующих слайдах рассмотрим преимущества dynamic block

```
#Хардкод способ без dynamic block
resource "yandex_vpc_security_group" "all_to_all" {
  name      = "web-server"
  .....
  ingress {
    protocol      = "TCP"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 22
  }
  ingress {
    protocol      = "TCP"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 80
  }
  ingress {
    protocol      = "TCP"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 443
  }
  egress {
    protocol      = "TCP"
    v4_cidr_blocks = ["0.0.0.0/0"]
    from_port     = 0
    to_port       = 65365
  }
  .....A long-long story in a git far-far away.....
}
```

### # Описание переменной в файле security.tf

```
variable "security_group_ingress" {
  type = list(object(
    {
      protocol      = string
      description   = string
      v4_cidr_blocks = list(string)
      port          = optional(number)
      from_port     = optional(number)
      to_port       = optional(number)
    })
  default = []
}
```

```
variable "security_group_egress" {
  type = list(object(
    {
      protocol      = string
      description   = string
      v4_cidr_blocks = list(string)
      port          = optional(number)
      from_port     = optional(number)
      to_port       = optional(number)
    })
  default = []
}
```

### #Загружаем переменные из файла security.auto.tfvars

```
security_group_ingress = [
  {
    protocol      = "TCP"
    description   = "разрешить входящий ssh"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 22
  },
  {
    protocol      = "TCP"
    description   = "разрешить входящий http"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 80
  },
  {
    protocol      = "TCP"
    description   = "разрешить входящий https"
    v4_cidr_blocks = ["0.0.0.0/0"]
    port          = 443
  },
]

security_group_egress = [
  {
    protocol      = "TCP"
    description   = "разрешить весь исходящий трафик"
    v4_cidr_blocks = ["0.0.0.0/0"]
    from_port     = 0
    to_port       = 65365
  },
]
```

### #Способ с использованием dynamic block

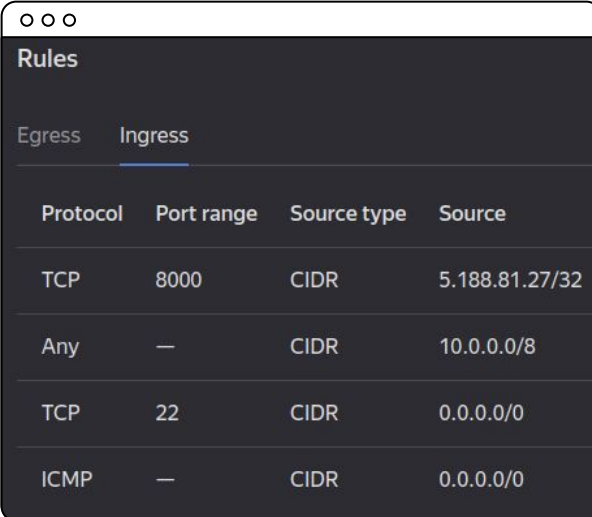
```
resource "yandex_vpc_security_group" "example" {
  name          = "example_dynamic"
  network_id    = yandex_vpc_network.develop.id
  folder_id     = var.folder_id

  dynamic "ingress" {
    for_each = var.security_group_ingress
    content {
      protocol      = lookup(ingress.value, "protocol", null)
      description   = lookup(ingress.value, "description", null)
      port          = lookup(ingress.value, "port", null)
      from_port     = lookup(ingress.value, "from_port", null)
      to_port       = lookup(ingress.value, "to_port", null)
      v4_cidr_blocks = lookup(ingress.value, "v4_cidr_blocks", null)
    }
  }

  dynamic "egress" {
    for_each = var.security_group_egress
    content {
      protocol      = lookup(egress.value, "protocol", null)
      description   = lookup(egress.value, "description", null)
      port          = lookup(egress.value, "port", null)
      from_port     = lookup(egress.value, "from_port", null)
      to_port       = lookup(egress.value, "to_port", null)
      v4_cidr_blocks = lookup(egress.value, "v4_cidr_blocks", null)
    }
  }
}
```



Name	ID	Network	Description
example_dynamic	enp07gee2nrr2ni8fk10	netology-develop	—



Ingress			
Protocol	Port range	Source type	Source
TCP	8000	CIDR	5.188.81.27/32
Any	—	CIDR	10.0.0.0/8
TCP	22	CIDR	0.0.0.0/0
ICMP	—	CIDR	0.0.0.0/0

# Шаблонизация

Функция `templatefile` ("путь к файлу-шаблону", { переменные })

Рекомендуемые расширение файлов-шаблонов — **\*.tftpl** или **\*.tpl**

Чтобы наполнить файлы шаблонов, используют **for loop** и **directives**

# Пример шаблона для Ansible inventory

содержимое файла `ansible.tftpl`:

```
[servers]
```

```
%{~ for k,v in webservers ~}  
${k}    ansible_host = ${v}  
%{~ endfor ~}
```

```
> templatefile("./ansible.tftpl",  
{ webservers = { server1="1.1.1.1", server2="2.2.2.2" } })
```

**Результат:**

```
[servers]
```

```
server1    ansible_host = 1.1.1.1  
server2    ansible_host = 2.2.2.2
```

# Provisioners

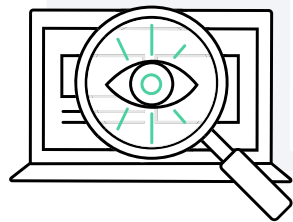


3



# Демонстрация работы

Terraform console ([ссылка](#) на демокод на GitHub). Dynamic blocks





**Блок provisioner используют, чтобы выполнять определённые действия на целевом ресурсе после его создания.**

Этот блок может содержать команды для выполнения на локальном или удалённом сервере, что позволяет настроить целевой ресурс

# Блок provisioner

Объявляются внутри блока **resource {}** и **выполняются только один раз, сразу после создания этого ресурса.**

Если поместить **provisioner** в специальный **null\_resource**, его можно запускать по условию с помощью блока **trigger {}**.

## Типы provisioners:

- file
- local-exec
- remote-exec

Вместо provisioners разработчики Terraform рекомендуют использовать подготовленные **образы Packer или cloud-init**, входящий во все Linux ОС

```
resource "yandex_compute_instance" "web" {  
  ...  
    provisioner "<provisioner type>" {  
      command 1  
      command 2  
    }  
  
    provisioner "<provisioner type>" {  
      command 1  
      command 2  
    }  
  ...  
}
```

# file provisioner

Копирует файлы или директории с локального Terraform-сервера на удалённую VM.

Требует блок **connection** { .. } для настройки авторизации.

Может пригодится для копирования:

- скриптов
- сертификатов
- конфиг-файлов

```
resource "yandex_compute_instance" "web" {  
  ...  
  provisioner "<provisioner type>" {  
    command 1  
    command 2  
  }  
  
  provisioner "<provisioner type>" {  
    command 1  
    command 2  
  }  
  ...  
}
```



**local-exec provisioner позволяет  
выполнить shell-команду на локальном  
сервере (там, где запускается Terraform)**

# local-exec provisioner

Часто используется в **null\_resource** для запуска ansible-playbook с сервера, где запускается Terraform. Команды внутри этого ресурса **выполняются по порядку**.

Триггером для запуска provisioner «local-exec» в этом примере служит **изменение значения текущего времени** (т. е. запускается всегда).

Таким образом Terraform сначала создаёт ВМ, а после запускает ansible для её настройки

```
resource "null_resource" "web_hosts_provision" {
  depends_on = [yandex_compute_instance.web]

  #Добавление ssh ключа в ssh-agent
  provisioner "local-exec" {
    command = "echo '${var.private_key}' | ssh-add -"
  }

  #Создание inventory из файла шаблона
  provisioner "local-exec" {
    command = <<-EOA
    echo "${templatefile("ansible_inventory.yml.tftpl",
    { hosts = yandex_compute_instance.web[*] })}" >
    hosts.yml
    EOA
  }

  #Запуск ansible-playbook
  provisioner "local-exec" {
    command = "ansible-playbook -i hosts.yml
    provision.yml"
    interpreter = ["bash"]
    environment = { ANSIBLE_HOST_KEY_CHECKING = "False"
  }

  triggers = { always_run = "${timestamp()}" }
}
```

# remote-exec provisioner

Выполняет команды на VM удалённо.

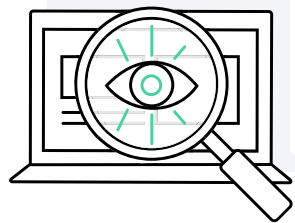
Требует блок **connection** { } для настройки авторизации.

Чтобы указать адрес подключения, используют специальную переменную **self**

```
resource "yandex_compute_instance" "web" {  
  ...  
  connection {  
    type      = "ssh"  
    user      = "root"  
    private_key = var.id_rsa  
    host      = self.public_ip  
  }  
  
  provisioner "remote-exec" {  
    command = " ..."  
  }  
  ...  
}
```

# Демонстрация работы

templatefile: ansible inventory. local-exec ansible





# Итоги занятия

- Узнали, для чего нужны метааргументы
- Разобрали виды выражений
- Познакомились с provisioners, совместным использованием terraform и ansible

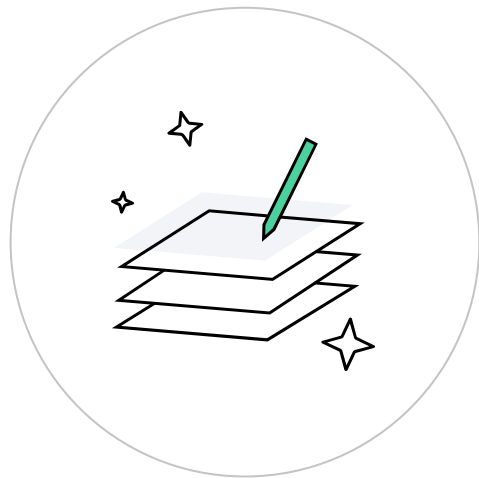


# Домашнее задание

Давайте посмотрим ваше домашнее задание

**Цель домашнего задания** — научиться работать с условными выражениями и шаблонизатором.

- 1 Вопросы о домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



# Дополнительные материалы

- [Документация](#) expressions
- [Документация](#) provisioners



# Задавайте вопросы и пишите отзыв о лекции

Елисей Ильин  
DevOps-инженер в Itransition

