

# Основы Terraform



Андрей  
Борю



## Андрей Борю

Principal DevOps Engineer, Snapcart





# План занятия

1. [Состояние проекта](#)
2. [Создание проекта](#)
3. [Пространства имен](#)
4. [Жизненный цикл](#)
5. [Provisioners \(провайдеры\)](#)
6. [Нулевой ресурс](#)
7. [Итоги](#)
8. [Домашнее задание](#)



# Состояние проекта

---

# State

Основная цель состояния Terraform - хранить связь между объектами в удаленной системе и экземплярами ресурсов, объявленными в конфигурации.

- По-умолчанию сохраняется в файле **terraform.tfstate**.
- В том числе хранит в себе метаданные, которые невозможно получить из облачных провайдеров.

---

# Работа со стейтом

**terraform state <subcommand> [options] [args]**

Поддерживаемые команды:

- list
- mv
- pull
- push
- rm
- show
- replace-provider

---

## Бэкэнды (backends)

Определяют место расположения стейтов и как следствие влияют на выполнения операций задействующие стейты. Бэкэнды использовать не обязательно.

Преимущества использования:

- Работа в команде, возможность блокировки, чтобы несколько людей одновременно не пытались применить изменения.
- Хранить конфиденциальную информации не на локальном диске.
- Удаленные операции, выполняющиеся не на вашем локальном компьютере.

## Типы бэкендов

- Standard – поддерживают сохранение стейтов и лок операций.
- Enhanced – стандарт + удаленные операции.

Все доступные бэкенды:

[terraform.io/docs/backends/types/index.html](https://terraform.io/docs/backends/types/index.html)



---

# S3 (aws storage) backend

Пример:

```
terraform {  
  backend "s3" {  
    bucket          = "terraform-states"  
    encrypt          = true  
    key              = "main-infra/terraform.tfstate"  
    region           = "ap-southeast-1"  
    dynamodb_table = "terraform-locks"  
  }  
}
```

---

## Workspaces (окружения)

- Зачастую одну и ту же конфигурацию с небольшими отличиями необходимо воссоздать несколько раз.
- Например несколько окружений: стейдж и продакшн.
- Каждому воркспейсу будет соответствовать отдельный стейт.
- Воркспейс **default** создается по-умолчанию.

---

# Работа с воркспейсами

Основные команды:

**terraform workspace [new, list, show, select and delete]**

- **new** – создать новый
- **list** – посмотреть список (проверяются стейт файлы)
- **select** – выбрать с которым будем работать
- **show** – показать название текущего
- **delete** – удалить



# Создадим первый проект

---

# main.tf

```
provider "aws" {  
  region = "us-east-1"  
}
```

---

# versions.tf

```
terraform {  
  required_providers {  
    aws = {  
      source  = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
  }  
}
```

# Инициализируем проект

```
$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Finding hashicorp/aws versions matching "~> 3.0"...
- Installing hashicorp/aws v3.8.0...
- Installed hashicorp/aws v3.8.0 (signed by HashiCorp)

```
Terraform has been successfully initialized!
```

---

## Создаем воркспейсы (не обязательно)

```
$ terraform workspace new stage
```

```
Created and switched to workspace "stage"!
```

You're now on a new, empty workspace. Workspaces isolate their state,

so if you run "terraform plan" Terraform will not see any existing state

for this configuration.

```
$ terraform workspace new prod
```

```
Created and switched to workspace "prod"!
```



---

# Создаем ec2 инстанс

Добавляем ресурс в main.tf

```
resource "aws_instance" "web" {  
  ami = "ami-00514a528eadbc95b" // Amazon Linux  
  instance_type = "t3.micro"  
  
  tags = {  
    Name = "HelloWorld"  
  }  
}
```

---

# Минимальный набор параметров

- **ami.** Образ Amazon Machine Image (AMI), который будет запущен на сервере EC2. В [AWS Marketplace](#) можно найти платные и бесплатные образы. Также можно создать собственный экземпляр AMI, применяя такие инструменты, как Packer.
- **instance\_type.** [Тип сервера EC2](#), который нужно запустить. У каждого типа есть свой объем ресурсов процессора, памяти, дискового пространства и сети.

# Планируем изменения

## Выполним `terraform plan`

Terraform will perform the following actions:

```
# aws_instance.web will be created
+ resource "aws_instance" "web" {
  + ami                        = "ami-0c55b159cbfaffe1f0"
  + arn                       = (known after apply)
  ...
+ root_block_device {
  + delete_on_termination = (known after apply)
  ...
  + volume_size           = (known after apply)
  + volume_type            = (known after apply)
}
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

# Ищем название ami автоматически

Воспользуемся блоком **data**.

```
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name = "name"
    values = ["amzn-ami-hvm-*-x86_64-gp2"]
  }
  filter {
    name = "owner-alias"
    values = ["amazon"]
  }
}

resource "aws_instance" "web" {
  ami = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"
}
```

# Добавляем зависимость от воркспейса

Для этого воспользуемся локальной переменной.

```
locals {  
  web_instance_type_map = {  
    stage = "t3.micro"  
    prod  = "t3.large"  
  }  
}  
  
resource "aws_instance" "web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = local.web_instance_type_map[terraform.workspace]  
}
```

# Создаем несколько ресурсов

Параметр count

```
locals {  
  web_instance_count_map = {  
    stage = 0  
    prod  = 1  
  }  
}  
  
resource "aws_instance" "web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t3.micro"  
  count = local.web_instance_count_map[terraform.workspace]  
}
```

## Еще один цикл

Параметр `for_each`

```
locals {  
  instances = {  
    "t3.micro" = data.aws_ami.amazon_linux.id  
    "t3.large" = data.aws_ami.amazon_linux.id  
  }  
}  
  
resource "aws_instance" "web" {  
  for_each = local.instances  
  
  ami = each.value  
  instance_type = each.key  
}
```

# Жизненный цикл

Меняем стандартное поведение ресурса.

- **create\_before\_destroy** – создать новый ресурс, перед удалением старого, если нет возможности обновить ресурс без пересоздания.
- **prevent\_destroy** – запретить удалять ресурс.
- **ignore\_changes** – не обращать внимания при планирование изменений на указанные свойства ресурсов.



# Жизненный цикл

Меняем стандартное поведение ресурса.

```
resource "aws_instance" "web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t3.micro"  
  tags = {"project": "main"}  
  
  lifecycle {  
    create_before_destroy = true  
    prevent_destroy = true  
    ignore_changes = ["tags"]  
  }  
}
```

# Таймауты

Иногда создание ресурса может занять очень много времени

```
resource "aws_instance" "web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t3.micro"  
  
  timeouts {  
    create = "60m"  
    delete = "2h"  
  }  
}
```



# Provisioners (провайдеры)



## Provisioners (провайдеры)

Это дополнительные блоки позволяющие расширить функционал ресурсов.

Но их рекомендуется использовать только в крайнем случае, если точно нет более подходящих средств.



## File provision

Используется для копирования файлов или каталогов с компьютера, на котором выполняется Terraform, во вновь созданный ресурс.

# Передача файлов

```
resource "aws_instance" "web" {  
  # ...  
  # Копируем файла myapp.conf в /etc/myapp.conf  
  provisioner "file" {  
    source      = "conf/myapp.conf"  
    destination = "/etc/myapp.conf"  
  }  
  
  # Создаем файл содержащий строку /tmp/file.log  
  provisioner "file" {  
    content      = "ami used: ${self.ami}"  
    destination = "/tmp/file.log"  
  }  
  
  # Копируем каталог configs.d в /etc/configs.d  
  provisioner "file" {  
    source      = "conf/configs.d"  
    destination = "/etc"  
  }  
}
```

---

# Настройки соединения

```
resource "aws_instance" "web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t3.micro"  
  
  provisioner "file" {  
    source = "conf/myapp.conf"  
    destination = "/etc/myapp.conf"  
  
    connection {  
      type = "ssh"  
      user = "root"  
      password = "${var.password}"  
      host = "${self.public_ip}"  
    }  
  }  
}
```

# local-exec

Вызывает локальный исполняемый файл после создания ресурса.

```
resource "aws_instance" "web" {  
  # ...  
  
  provisioner "local-exec" {  
    command = "echo ${self.private_ip} >> private_ips.txt"  
  }  
}
```

- **command** – команда
- **working\_dir** – рабочая директория для исполнения
- **interpreter** – интерпретатор (perl, python, php ...)
- **environment** – переменные окружения



## remote-exec

Вызывает скрипт на удаленном ресурсе.

```
resource "aws_instance" "web" {  
  # ...  
  provisioner "remote-exec" {  
    inline = [  
      "puppet apply",  
      "consul join ${aws_instance.web.private_ip}",  
    ]  
  }  
}
```

- **inline** – скрипты,
- **script** – путь к локальному скрипту, который будет скопирован и исполнен удаленно,
- **scripts** – список скриптов.

---

## Нулевой ресурс

Если необходимо запустить действия, которые напрямую не связаны с конкретным ресурсом, то можно воспользоваться **null\_resource**, который, по-умолчанию, ничего не делает.

Но можно указать:

- зависимости,
- локальные или удаленные скрипты,
- триггеры,
- другие аргументы.



# Итоги

---

## Итоги

- Узнали как хранится состояние проекта.
- Разобрались с бэкэндами.
- Создали отдельные воркспейсы.
- Познакомились с особенностями настройки ресурсов.



# Домашнее задание



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Андрей Борю**



[andreyborue](https://t.me/andreyborue)



[andreyborue](https://netology.ru/andreyborue)



[andreyborue](https://t.me/andreyborue)