

Автоматизация разработки и эксплуатации программного обеспечения (осень 2022 года)



ИУ-5, бакалавриат, курс по выбору

Виды занятий

- Лекции:
 - 17 лекций, 34 часа.
 - ПОНЕДЕЛЬНИК, 10.15, 430 (ГЗ)
- Лабораторные работы
 - 8 лабораторных работ, 34 часа.
 - по расписанию
- Домашнее задание.
 - Проект по развертыванию программного обеспечения.
- Репозиторий курса:
 - <https://github.com/iu5git/DevOps>



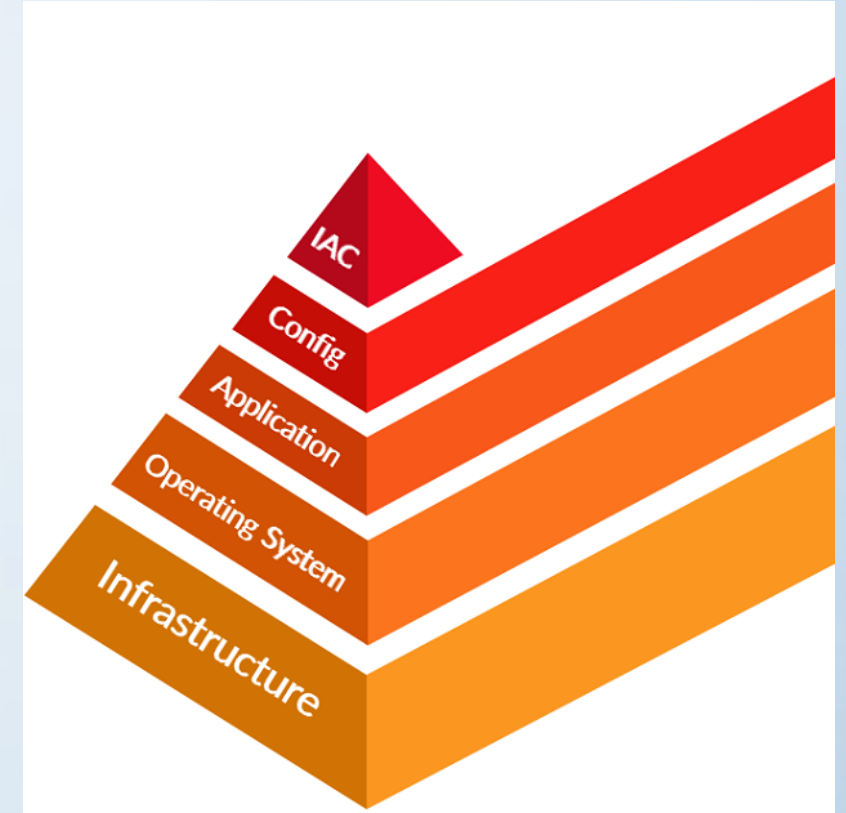
Infrastructure as Code. Terraform

БАЛАШОВ АНТОН

Infrastructure as Code

IAC, или **Infrastructure as Code**, представляет собой процесс подготовки (provisioning) и управления компьютерными центрами обработки данных с помощью машиночитаемых файлов определений, а не физической конфигурации оборудования или интерактивных инструментов конфигурации.

Хотя область IAC является относительно новой по сравнению с конвейером автоматизации DevOps, уже существует достаточно много IAC-инструментов, и новые технологии продолжают развиваться даже в этот самый момент.



Infrastructure as Code Benefits



Automated deployment

Repeatable process

Consistent environments

Reusable components

Documented architecture

Преимущества

- **Скорость и уменьшение затрат:** IaC позволяет быстрее конфигурировать инфраструктуру и направлен на обеспечение прозрачности, чтобы помочь другим командам со всего предприятия работать быстрее и эффективнее. Это освобождает дорогостоящие ресурсы для выполнения других важных задач.
- **Масштабируемость и стандартизация:** IaC предоставляет стабильные среды быстро и на должном уровне. Командам разработчиков не нужно прибегать к ручной настройке - они обеспечивают корректность, описывая с помощью кода требуемое состояние сред. Развертывания инфраструктуры с помощью IaC повторяемы и предотвращают проблемы во время выполнения, вызванных дрейфом конфигурации или отсутствием зависимостей. IaC полностью стандартизирует инфраструктуру, что снижает вероятность ошибок или отклонений.
- **Безопасность и документация:** Если за создание всех вычислительных, сетевых и служб хранения отвечает код, они каждый раз будут развертываться одинаково. Это означает, что стандарты безопасности можно легко и последовательно применять в разных компаниях. IaC также служит некой формой документации о правильном способе создания инфраструктуры и страховкой на случай, если сотрудники покинут компанию, обладая важной информацией. Поскольку код можно версионировать, IaC позволяет документировать, регистрировать и отслеживать каждое изменение конфигурации вашего сервера.
- **Восстановление в аварийных ситуациях:** IaC — чрезвычайно эффективный способ отслеживания инфраструктуры и повторного развертывания последнего работоспособного состояния после сбоя или катастрофы любого рода.

Недостатки

- **Логика и соглашения:** необходимо понимать IaC скрипты независимо от того, написаны ли они на языке конфигурации HashiCorp (HCL) или на обычном Python или Ruby
- **Обслуживаемость и возможность отслеживания:** хотя IaC предоставляет отличный способ отслеживания изменений в инфраструктуре и мониторинга, обслуживание сетапа IaC само по себе становится проблемой при достижении определенного масштаба. Когда IaC широко используется в организации с несколькими командами, отслеживание и управление версиями конфигураций не так просты, как может показаться на первый взгляд.
- **RBAC:** Основываясь на нем, управление доступом тоже становится сложной задачей. Установка ролей и разрешений в различных частях организации, которые внезапно получают доступ к скриптам для быстрого развертывания кластеров и сред, может оказаться довольно сложной задачей.
- **Запаздывание фич:** Инструменты IaC, не зависящие от поставщика (например, Terraform), часто запаздывают с фичами в сравнении с продуктами, привязанными к конкретному поставщику. Это связано с тем, что поставщикам инструментов необходимо обновлять провайдеров, чтобы полностью охватить новые облачные фичи, выпускаемые с постоянно растущими темпами.

Инструменты IAC

Имя	Описание	Синтаксис	Лицензия	Сайт	GitHub репозиторий
Terraform	Terraform — это программный IAC-инструмент, созданный HashiCorp. Он известен как декларативный provisioning-инструмент без агентов и без мастера.	.tf файл (похож на JSON)	MPL 2.0	terraform.io	github.com/hashicorp/terraform
Ansible	Поддерживаемый Red Hat, Ansible — это программный IAC-инструмент, вмещающий в себе provisioning, управление конфигурацией и развертывания приложений.	YAML	GPL 3.0	ansible.com	github.com/ansible/ansible
Chef	Chef автоматизирует процесс управления конфигурациями, гарантируя, что каждая система правильно настроена и согласована.	Ruby	Apache 2.0	chef.io/products/chef-infra	github.com/chef/chef
Puppet	Puppet — это инструмент управления конфигурацией программного обеспечения, который имеет собственный декларативный язык для описания конфигурации системы.	Язык Puppet (похож на JSON) или Ruby	Apache 2.0	puppet.com	github.com/puppetlabs/puppet
SaltStack	Поддерживаемый VMWare, SaltStack — это программное обеспечение с открытым исходным кодом на основе Python для управляемой событиями (event-driven) IT-автоматизации, удаленного выполнения задач и управления конфигурацией.	Python	Apache 2.0	repo.saltproject.io	github.com/saltstack/salt
Pulumi	IAC SDK с открытым исходным кодом Pulumi позволяет создавать, развертывать и управлять инфраструктурой в любом облаке, используя ваши любимые языки.	Различные языки программирования	Apache 2.0	pulumi.com	github.com/pulumi/pulumi

Agent vs Agentless

IAC-инструмент может требовать запуска агента на каждом сервере, который вы хотите настроить.

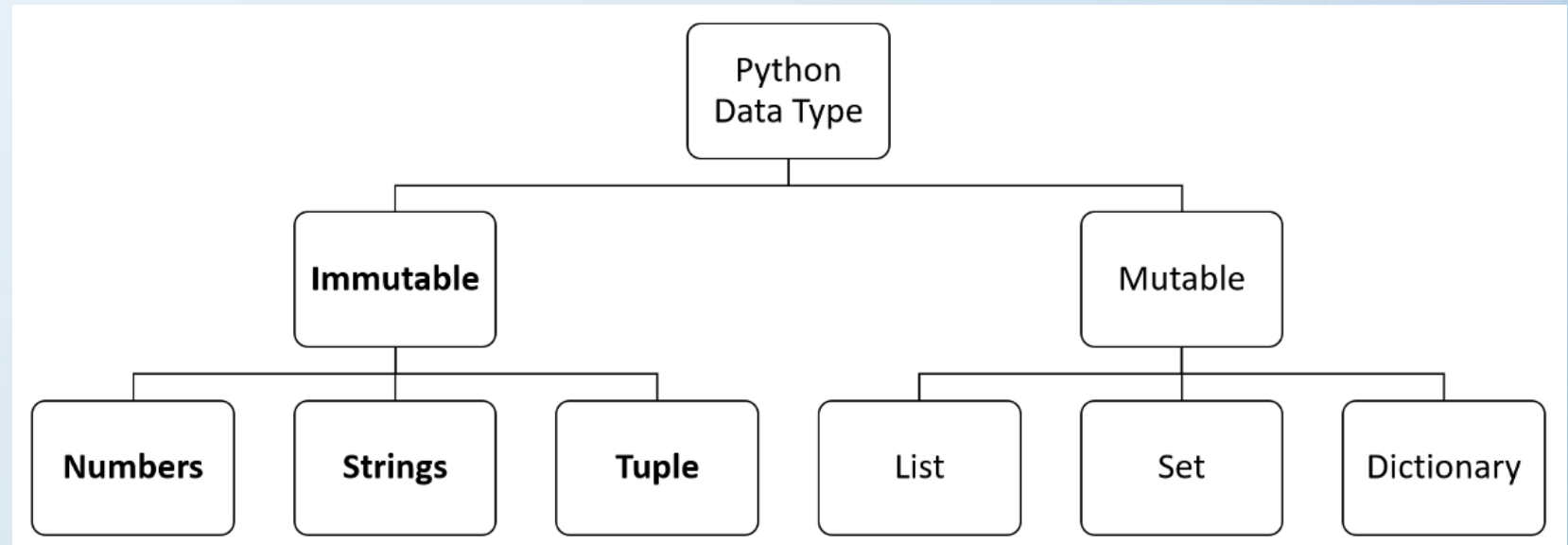
Если нет, то такой инструмент называется «agentless».



Mutable vs Immutable

Некоторые инструменты, такие как Terraform, занимаются не изменением уже подготовленной (provisioned) инфраструктуры, а развертывают новый сервер, что означает, что они следуют парадигме неизменяемой (immutable) инфраструктуры.

Другие инструменты, такие как Ansible, Chef, SaltStack и Puppet, могут изменять существующие ресурсы, а это означает, что эти инструменты следуют парадигме изменяемой (mutable) инфраструктуры.




Procedural vs Declarative

Процедурные языки, такие как Ansible и Chef, позволяют описывать с помощью кода поэтапное выполнение.

Декларативные языки, такие как Terraform, SaltStack и Puppet, позволяют просто указать желаемое состояние.

DECLARATIVE VS. PROCEDURAL

Declarative	"Make me a cake."
Procedural	Mix 3 cups of flour, 2 ¼ cups of sugar, 2 tsp. of baking powder, 1 tsp. salt, 3 sticks of butter, ½ tsp. of vanilla extract, 1 cup milk, and 8 large egg whites. Bake at 350 for 40 minutes.



A hand holding a black pen is pointing at the 'Procedural' row of the table.

Master vs Masterless

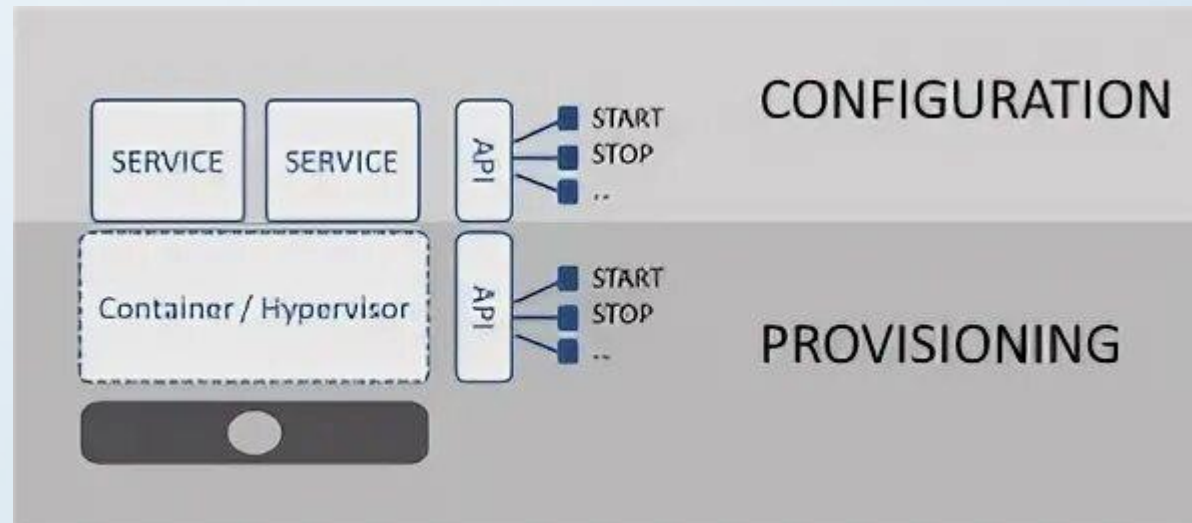
Языки, такие как Chef, требуют, чтобы вы запускали отдельный главный сервер (master), чтобы обеспечить дополнительное управление и постоянные состояния.

Другие языки, такие как Ansible и Terraform, не нуждаются в определении мастера.



Configuration vs Provisioning

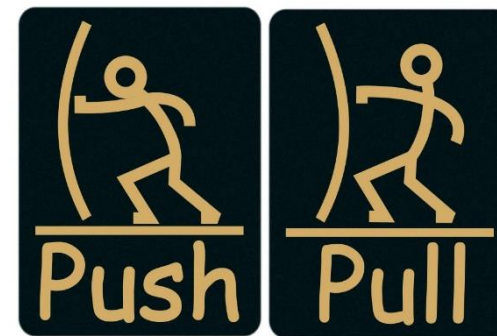
- Ansible, Chef, SaltStack и Puppet известны как инструменты управления конфигурацией, что означает, что их основная цель — настроить ресурсы. Другие инструменты, такие как Terraform и Pulumi, являются инструментами обеспечения (provisioning), а это означает, что их основная цель — предоставлять ресурсы. Однако по мере того, как инструменты развиваются, их функционал может начать пересекаться.



Pull & Push

В основе подхода pull лежит тот факт, что все изменения применяются изнутри кластера. Внутри кластера есть оператор, который регулярно проверяет связанные репозитории Git и Docker Registry. Если в них происходят какие-либо изменения, состояние кластера обновляется изнутри. Обычно считается, что подобный процесс весьма безопасен, поскольку ни у одного внешнего клиента нет доступа к правам администратора кластера.

В push-подходе внешняя система (преимущественно CD-пайплайны) запускает развертывания в кластер после коммита в Git-репозиторий или в случае успешного выполнения предыдущего CI-пайплайна. В этом подходе система обладает доступом в кластер.



Chef

В отличие от Terraform или Ansible, Chef требует установки мастер-сервера и запустили агентов на серверах, с которыми вы хотите работать. Наличие мастера и агентов может иметь несколько преимуществ.

Мастер-сервер может быть центральным местом, из которого можно управлять и контролировать инфраструктуру.

Наличие агентов может гарантировать правильную работу обновлений. Однако, агенты и мастер-сервер также предполагают дополнительное обслуживание большую уязвимость. Когда вы выбираете IAC для начала, вы должны убедиться, какой инструмент лучше всего подходит для вашей ситуации.

Файл конфигурации Chef основан на языке программирования Ruby, который может дать вам некоторую гибкость, если вы хотите попробовать дополнительную логику управления.



Puppet



- **Puppet** — это инструмент управления конфигурацией программного обеспечения с декларативным синтаксисом, требующий наличия мастер-сервера и агентов. Puppet — это инструмент с открытым исходным кодом, имеющий лицензию **Apache 2.0**. Язык программирования Puppet — это декларативный язык, который описывает состояние компьютерной системы в терминах «ресурсов», которые представляют собой базовые конструкции сети и операционной системы. Пользователь собирает ресурсы в манифесты, которые описывают желаемое состояние системы.

SaltStack



SaltStack, также известный как **Salt**, это event-driven ПО с открытым исходным кодом на основе Python для автоматизации IT, удаленного выполнения задач и управления конфигурацией. Основой Salt является механизм удаленного выполнения, который создает высокоскоростную, безопасную и двунаправленную сеть связи для групп систем. Помимо этой системы связи, Salt предоставляет чрезвычайно быструю, гибкую и простую в использовании систему управления конфигурациями, называемую Salt States. Вы также должны познакомиться с такими терминами Salt, как **grains**, **pillars** и **mine**.

Pulumi



Pulumi — это IAC-инструмент с открытым исходным кодом для создания, развертывания и управления облачной инфраструктурой. Хотя Pulumi — самый молодой инструмент из этого списка, в последнее время он набирает огромную популярность. Самым уникальным аспектом этого инструмента является то, что вы можете написать код конфигурации на любом из следующих языков программирования: Python, Typescript, Javascript, C# или Go.

Terraform



Infrastructure automation tool

Open-source and vendor agnostic

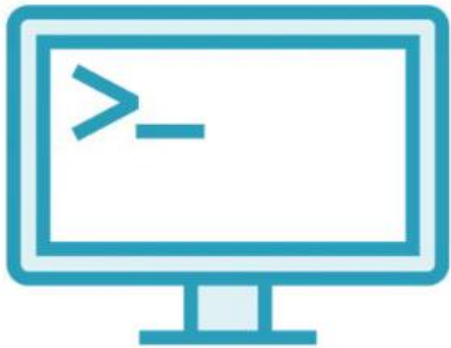
Single binary compiled from Go

Declarative syntax

HashiCorp Configuration Language or JSON

Push based deployment

Core Components



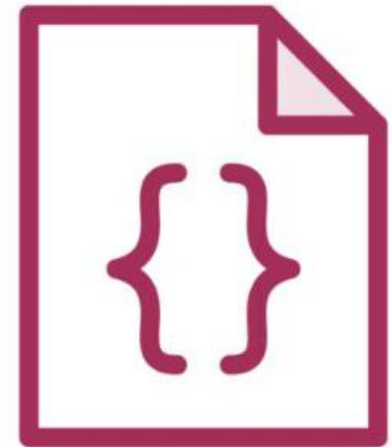
Executable



**Configuration
files**



Provider plugins



State data



Terraform executable

SANCTIONS

Скачать с зеркала:

<https://hashicorp-releases.yandexcloud.net/terraform/>

```
export PATH=$PATH:/path/to/terraform
```



VK Cloud

Yandex  Cloud





Hashicorp Configuration Language

- Для автоматизации работы с инфраструктурой Terraform использует собственный язык написания конфигурационных файлов Hashicorp Configuration Language (HCL). По сути, этот язык описывает желаемое состояние инфраструктуры в конфигурационном файле.
- Для описания того или иного создаваемого элемента необходимо подготовить блок, содержащий заключенные в фигурных скобках названия переменных, и их значения, передаваемые функциям.
- Как и в большинстве языков программирования, в HCL используются аргументы для присвоения значений переменным. В Terraform эти переменные являются атрибутами, связанными с определенным типом блока. Таким образом, весь код HCL состоит из подобных блоков.



Hashicorp Configuration Language

Хранение файлов

Все конфигурационные файлы Terraform должны размещаться в одном каталоге. Так, для того примера. По умолчанию Terraform предполагает, что все файлы с *.tf расширениями в данном каталоге являются частью конфигурации, независимо от имен файлов.

Для грамотной организации больших конфигурации потребуются три файла:

- variables.tf – для всех объявленных входных переменных.
- provider.tf – для объявленных поставщиков, которых вы используете.
- main.tf – для объявления фактических ресурсов, которые будут созданы.

Однако, такая структура не является обязательной. Можно поместить весь код в один файл и все будет корректно работать.

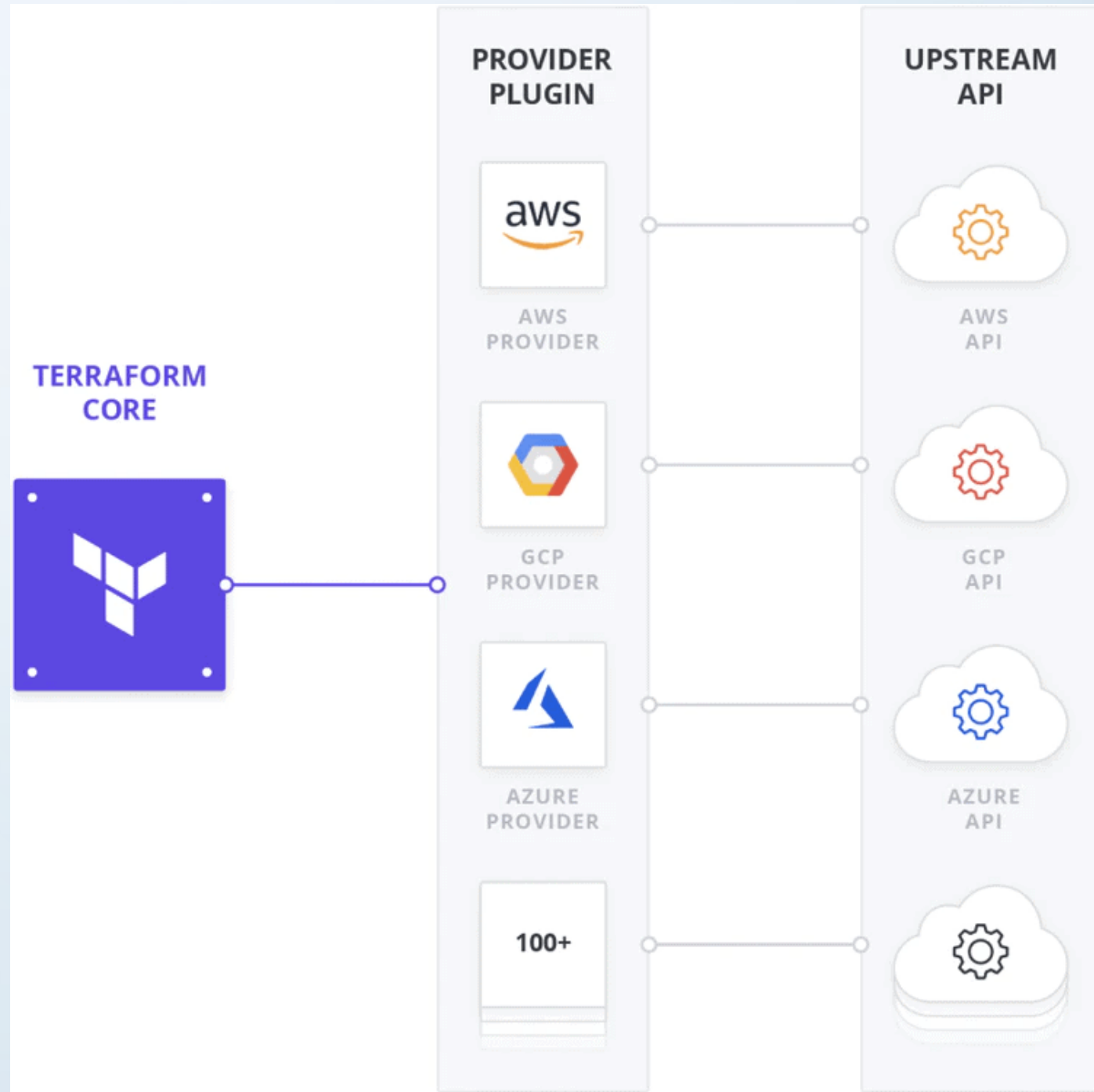
Пример

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

- В первой строке, открываем блок terraform, который хотя и не является обязательным с точки зрения синтаксиса системы, но рекомендуется его указывать.
- Вложенный блок required providers, определяет требуемых провайдеров. Нам потребуется поставщик aws с указанными параметрами source и version.



Providers





State

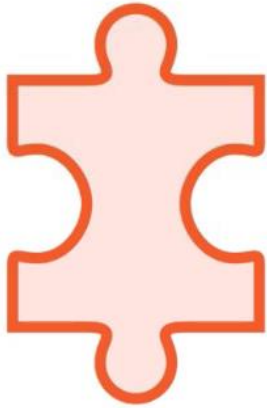
Состояние Terraform - это файл, в котором хранятся все сведения о ресурсах, которые были созданы в контексте данного проекта.

Например, если объявить ресурс `azure_resourcegroup` и запустить Terraform, в файле состояния будет сохранен его идентификатор.

Основная цель файла состояния - предоставить информацию об уже существующих ресурсах, поэтому, когда изменяются файлы конфигурации ресурсов, Terraform может определить, что ему нужно делать.

Важным моментом в отношении файлов состояния является то, что они могут содержать конфиденциальную информацию. Примеры включают начальные пароли, используемые для создания базы данных, закрытые ключи и так далее. Terraform использует концепцию серверной части для хранения и извлечения файлов состояния. Серверной частью по умолчанию является локальная серверная часть, которая использует файл в корневой папке проекта в качестве места хранения

Terraform Object Types



Providers



Resources



Data sources

Providers

Provider работает как драйвер устройства операционной системы. Он предоставляет набор типов ресурсов, используя общую абстракцию, таким образом маскируя детали того, как создавать, изменять и уничтожать ресурс.

Terraform автоматически загружает поставщиков из своего публичного реестра по мере необходимости, исходя из ресурсов данного проекта. Он также может использовать пользовательские плагины, которые должны быть установлены пользователем вручную.

За некоторыми исключениями, использование провайдера требует настройки его с некоторыми параметрами

Хотя это и не является строго необходимым, считается хорошей практикой явно указывать, какого поставщика мы будем использовать в проекте Terraform, и указывать его версию:

```
provider "kubernetes" {  
  version = "~> 1.10"  
}
```


Resources

В Terraform ресурс - это все, что может быть целью для операций CRUD в контексте данного поставщика. Некоторыми примерами являются экземпляр EC2, Azure MariaDB или запись DNS.

```
resource "aws_instance" "web" {  
    ami = "some-ami-id" instance_type = "t2.micro"  
}
```

Сначала всегда есть ключевое слово `resource`, с которого начинается определение. Далее тип ресурса, который обычно соответствует типу провайдера. В приведенном выше примере `aws_instance` - это тип ресурса, определенный поставщиком AWS. После этого появляется определяемое пользователем имя ресурса, которое должно быть уникальным для этого типа ресурса в том же модуле - подробнее о модулях позже.

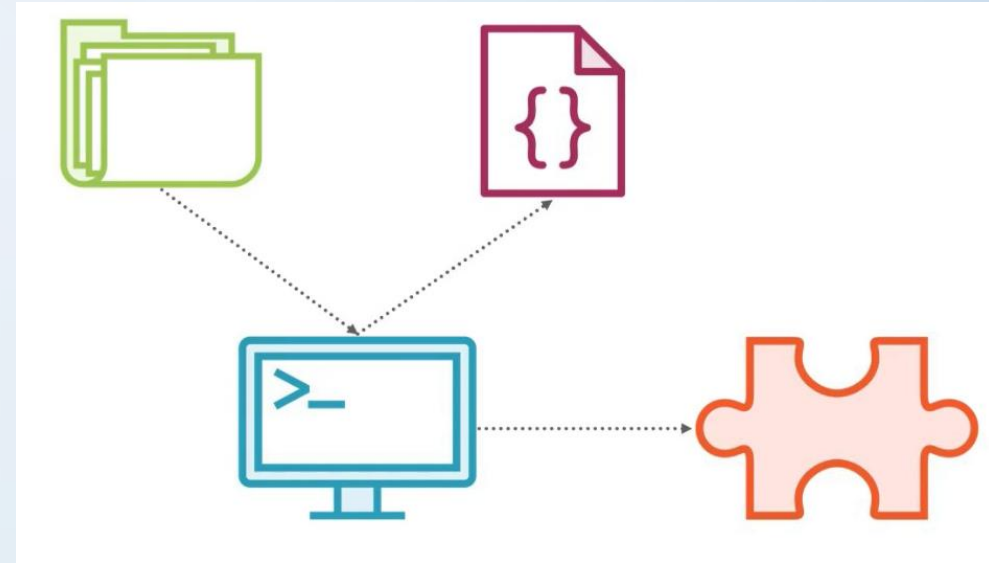
Data sources

```
data "aws_ami" "ubuntu" {  
  most_recent = true  
  
  filter {  
    name = "name"  
  
    values = ["ubuntu/images/hvm-ssd/ubuntu-trusty-14.04-amd64-server-*"]  
  }  
  
  filter {  
    name = "virtualization-type"  
  
    values = ["hvm"]  
  }  
  
  owners = ["099720109477"] # Canonical  
}
```

Источники данных работают в значительной степени как ресурсы, доступные только для чтения, в том смысле, что мы можем получать информацию о существующих, но не можем создавать или изменять их. Обычно они используются для извлечения параметров, необходимых для создания других ресурсов.

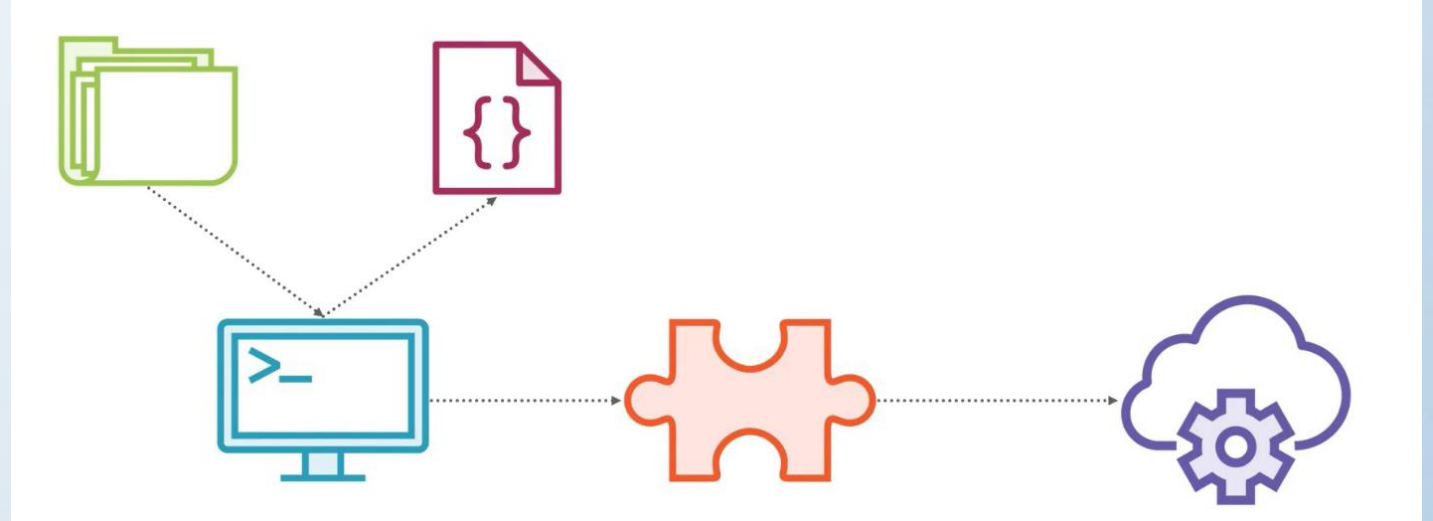
Workflow

- Terraform init
 - Ищет конфигурационные файлы
 - Смотрит, нужно ли скачивать плагины и скачивает если нужно
 - Сохраняет состояние



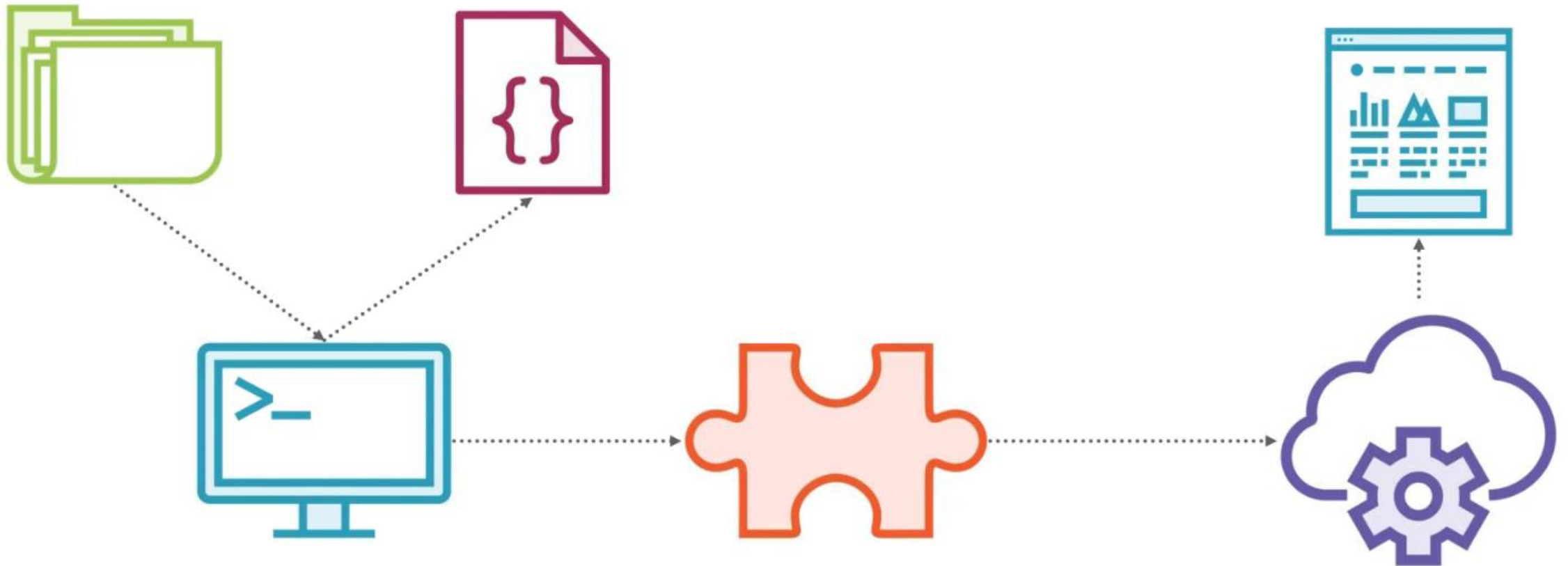
Workflow

- Terraform plan
 - Сравнивает конфигурацию и состояние и строит план преобразования



Workflow

- Применяет изменения к инфраструктуре



Provider: local

main.tf

```
provider "local" {  
  version = "~> 2.2.3"  
}  
  
resource "local_file" "hello" {  
  content = "Hello, Terraform"  
  filename = "hello.txt"  
}
```


Provider: local

➤ terraform init

Initializing the backend...

Initializing provider plugins...

- Finding hashicorp/local versions matching "~> 2.2.3"...
- Installing hashicorp/local v2.2.3...
- Installed hashicorp/local v2.2.3 (unauthenticated)

Provider: local

> terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

local_file.hello will be created

+ resource "local_file" "hello" {

+ content = "Hello, Terraform"

+ directory_permission = "0777"

+ file_permission = "0777"

+ filename = "hello.txt"

+ id = (known after apply)

}

Provider: local

> terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

local_file.hello will be created

+ resource "local_file" "hello" {

+ content = "Hello, Terraform"

+ directory_permission = "0777"

+ file_permission = "0777"

+ filename = "hello.txt"

+ id = (known after apply)

}

Provider: local

> terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

local_file.hello will be created

+ resource "local_file" "hello" {

+ content = "Hello, Terraform"

+ directory_permission = "0777"

+ file_permission = "0777"

+ filename = "hello.txt"

+ id = (known after apply)

}

Provider: local

> terraform plan

local_file.hello: Refreshing state...

[id=392b5481eae4ab2178340f62b752297f72695d57]

No changes. Your infrastructure matches the configuration.

Provider: local

```
> cat .\hello.txt
```

Hello, Terraform

```
> echo foo > hello.txt
```

```
> terraform plan
```

local_file.hello: Refreshing state... [id=392b5481eae4ab2178340f62b752297f72695d57]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

local_file.hello will be created

Terraform destroy

- Terraform destroy
 - Освобождение ресурсов – удаляет инфраструктуру
 - Пользоваться надо с осторожностью

Minikube + Terraform



- `curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb`
- `sudo dpkg -i minikube_latest_amd64.deb`

Стартуем

- `minikube start --vm-driver=docker --base-image='kicbase/stable:v0.0.32'`

Проект

`minikube/main.tf`

Yandex cloud

- <https://cloud.yandex.ru/docs/iam/concepts/authorization/oauth-token>
- `yc config set token <OAuth-token>`
- `export YC_TOKEN=$(yc iam create-token)`
- `export YC_CLOUD_ID=$(yc config get cloud-id)`
- `export YC_FOLDER_ID=$(yc config get folder-id)`

Yandex cloud

```
nano ~/.terraformrc
```

```
provider_installation {  
  network_mirror {  
    url = "https://terraform-mirror.yandexcloud.net/" include = ["registry.terraform.io/*/*"]  
  }  
  direct {  
    {  
      exclude = ["registry.terraform.io/*/*"]  
    }  
  }  
}
```



ВОПРОС ?

ОТВЕТ !

Список литературы

- <https://habr.com/ru/company/otus/blog/570926/>
- <https://www.baeldung.com/ops/terraform-intro>
- <https://cloud.yandex.ru/docs/tutorials/infrastructure-management/terraform-quickstart>
- <https://www.freecodecamp.org/news/terraform-syntax-for-beginners/>