SkillFactory

ПРОФЕССИЯ

DevOps-инженер

КОНСПЕКТ

МОДУЛЬ B5. УПРАВЛЕНИЕ ИНФРАСТРУКТУРОЙ

О чем этот модуль

Terraform позволяет в декларативном формате описывать инфраструктуру (и не только). А главное — ***Terraform*** позволит управлять множеством различных провайдеров и сервисов через единую точку входа.

B5.1 Подход *IaC. Immutable infrastructure*

Infrastructure-as-Code (***IaC***) — это в большинстве своем декларативный (описываем конечный состояние, которое хотим получить, а не изменения, которые нужно внести) подход, описывающий инфраструктуру, в противовес ручному редактированию конфигураций или интерактивному взаимодействию.

B5.2 Знакомство с *Terraform*

Провайдеры

Почти любой компонент инфраструктуры можно представить в качестве ресурса. Мост для связи между инфраструктурными ресурсами и облачным провайдером осуществляется посредством ***API***.

Состояние (*Terraform State*)

Файлы состояния (***state***) хранят информацию об инфраструктуре, с которой ***Terraform*** работает.

State-файл сопоставляет реальные ресурсы провайдера с вашей описанной инфраструктурой, т. е. это некая связка уже созданных ресурсов с описанными ресурсами в коде. С помощью этого ***Terraform*** понимает, нужно ли создать новый ресурс, обновить, удалить или ничего не делать.

Terraform «из коробки» поддерживает ***code style*** конвенцию по форматированию кода.

Для того чтобы применить форматирование для вашего кода, достаточно ввести `terraform fmt`.

B5.3 Взаимозависимость ресурсов. *Data Sources*

`terraform_remote_state` — это возможность использовать удаленный ***state***-файл от другой части конфигурации или из другого ***Terraform***-проекта.

Таким образом, мы можем получать данные из различных проектов, передавать данные в различные ресурсы, которые не находятся в одном проекте.

B5.4 *Remote State. Remote Backends*

Зачем хранить ***state***-файл на удаленном хранилище?

- Чтобы не помешать работе коллег в ***Terraform*** предусмотрен **механизм блокировки**.
  Механизм блокировки работает со ***state***-файлом. На этапе построения плана и зависимостей ***Terraform*** проверяет наличие блокировки, если блокировка есть, то построение плана прекращается, это означает, что кто-то уже применяет изменения в ***Terraform***, и нужно подождать, пока блокировка не будет снята. Блокировка снимается после окончания работы со ***state***-файлом.

  Что может пойти не так, если не использовать эту полезную особенность?
  Если у вас ***state***-файл хранится в гите, то о изменениях друг друга вы узнаете только после коммита в гит, и, скорее всего, это будет неожиданностью и не совсем то, что вы ожидали увидеть.

- Иметь актуальный **бэкап *state***-файла.
  Если ***Terraform*** потеряет ***state***-файл, то он ничего не будет знать о существующей инфраструктуре и будет думать, что у вас ничего нет.

- Иметь доступ к данным из другого удаленного ***state-***файла через ***Terraform_remote_state.***
  Это может быть очень удобно, когда у нас конфигурации разделены логически, по частоте использования или еще по какой-либо логике на несколько проектов, но компоненты между тем являются связанными и используют данные друг друга в связанных ресурсах.

B5.5 Модули. Продвинутая концепция использования Terraform

Причины использования модульной структуры:

1. Понимание кода и навигация по коду сильно затруднена.

2. Ваш код сильно связан, обновление в одной части кода может привести к неожиданным последствиям в другой части.

3. Принцип ***DRY*** (***Don't repeat yourself***) не используется в коде. Множество частей кода повторяется. При обновлении одной повторяющейся части кода нужно менять код и во всех других связанных местах.

4. Вы хотите поделиться частью инфраструктурного кода с другой командой, переиспользовать в другом проекте.

Список рекомендаций, который необходим для разработки хороших модулей:

1. Стоит использовать версионирование для модулей. Этот момент можно легко реализовать во всех способах хранения модулей. Это поможет вам точно знать, какую версию кода вы собираетесь применить на продакшене.

2. Прописывайте максимально возможное количество значений по умолчанию, желательно указывать средневзвешенные оптимальные значения.
   Это имеет несколько преимуществ:
   - Вы будете знать примерно, какое значение ожидается и сможете переопределить его при желании в корневом модуле, т. е. это своего рода документация.
   - Если вам не нужно переопределять значение по умолчанию, то ваш код в корневом модуле становится человекочитаемым.

3. Указывайте версии зависимостей в модуле. Разные версии модулей могут по-разному работать с разными версиями провайдеров, а этого мы стараемся избегать с помощью ***IaC*** и хороших практик.

4. Добавляйте описание модулей в ***Readme***. Это поможет быстро понять, что этот модуль делает, какие переменные использует и т. д.

5. Используя переменные, заполняйте необязательные поля, описание переменной и тип. Это удобно использовать для автодокументации (более

подробно в юните о лучших практиках), а также ваш код будет более прозрачным.

6. Придерживайтесь официальных рекомендаций ***Terraform*** для модулей.

B5.6 Лучшие практики организации кода в Terraform

Почему важно версионировать модули и другие компоненты?

Потому что инфраструктурные ресурсы не имеют версии, а код имеет, поэтому с помощью рекомендаций ниже мы сможем версионировать нашу инфраструктуру.

Несколько причин, по которым рекомендуется разделять ваш ***Terraform***-код:

- План проекта строится более 10 мин, по умолчанию таймаут на ожидание ресурса у ***Terraform*** — 5 минут.

  Суть вот в чем: когда ***Terraform*** анализирует ***state***, он обрабатывает весь ***state*** полностью. И когда у вас в коде описано множество ресурсов, то ***Terraform*** должен опросить все ресурсы, сравнить это со своим ***state***-файлом и выдать разницу. В больших проектах это может занять 20 и более минут. Этот фактор будет влиять на скорость изменения в продакшен, будет влиять на скорость разработки (т.к. на демо-стендах этот процесс будет проходить с той же скоростью) и тестирования.

- Тяжело поддерживать.

  В репозитории уже каша, в которой очень тяжело разобраться. Таким образом, на минимальном уровне будут две метрики: читаемость кода и управляемость кода.

- Ваш проект очень сложен.

  В нем большое количество связанных ресурсов. Большое количество ***Terraform***-провайдеров. Стоит стремиться к упрощению кода, а не к усложнению.

- Одна часть вашей инфраструктуры изменяется очень редко, а другая — постоянно и достаточно часто.

Вы можете разделить эти части.

- Когда конфигурацию меняют несколько команд, при этом одна команда ответственна только за одну часть кода.

  Разделив ***Terraform*** на репозитории по этому признаку, вы дадите больше гибкости и понимания кода для этих команд.

- В одном репозитории описано множество различных платформ.

  Если разделить по этому признаку, то мы можем обращать больше внимания на специфику каждой платформы, что снова нам даст больше гибкости.

- В одном репозитории множество окружений.
  Можно разделить по принципу географического местоположения ресурсов, по принадлежности к окружению.

Как сгруппировать репозитории

1. **По частоте изменения**

   Например:
   - ***Core*** (скорее всего, эту часть мы не часто используем, только при первоначальной инициализации или при мажорных обновлениях).
   - ***Apps*** (наш базовый сервис).

2. **По целевому окружению (или набору окружений)**

   - ***Production***,
   - ***Staging***,
   - ***Development/Test/Demo.***

3. **По уровню абстракции**

   Например, команда, которая вносит изменения только в верхний уровень абстракции описания окружения (например, увеличивает количество нод на ресурсе), не обязана (или не хочет) разбираться в тонкостях работы ***Terraform*** и поддерживать ***Terraform***-модули.

4. **По уровню доступа**

Вероятно, не все участники проекта могут изменить инфраструктуру на сверхприватных/секретных/важных окружениях.

Структура кода

Давайте рассмотрим рекомендации, которые помогут нам создать хорошо структурированный ***Terraform***-код.

Не помещайте всю ***Terraform***-конфигурацию в один файл.

Не стоит в реальном проекте помещать весь ***Terraform***-код в один файл, потому что он будет большим и неудобно читаемым (обычно рекомендуют задуматься о разбиении кода, если количество строк кода более 100), его будет сложно поддерживать, в нем будет сложно разобраться, и, конечно, это плохая практика.

Стоит сгруппировать контент по различным файлам и назвать файлы значимо, чтобы можно было понять, что находится внутри.

Пример конфигурации ***Terraform*** со сгруппированным контентом по различным файлам:

```
.
├── README.md
├── main.tf
├── eu-region.tf
├── usa-region.tf
├── variables.tf
├── outputs.tf
├── locals.tf
```

Старайтесь использовать все те же принципы, что и в программировании для организации структуры кода:

1. ***DRY*** — ***Don't repeat yourself***, в переводе с англ. — «Не повторяйся».
   Этот принцип легко соблюсти, используя:
   - модули,
   - шаблоны.

2. Мы шаблонизируем часто повторяющий код, передавая только часто изменяющиеся переменные.
3. ***KISS*** — ***Keep It Simple, Stupid***, в переводе с англ. — «Не усложняй».
    - Модули,
    - Разделение на репозитории.
4. ***SOLID*** — ***Single-responsibility principle***, в переводе с англ. — «Принцип единственной ответственности».
    - Модули,
    - Разделение на репозитории.
5. Другие принципы и подходы из программирования, которые вы сочтете наиболее удобными у вас в проекте.

Использование ***locals*** для интерполяции и операций с данными.

Это достаточно удобно, когда вам, например, нужно получить данные из одного источника, обработать их и передать в далее.

Рассмотрим пример работы с ***locals***:

```
variable "owner" {
  type        = string
  description = "Base data of the cluster"
  default     = "ops-team"
}

locals {
  # Common tags to be assigned to all resources
  service_name = "service-${var.owner}"
  common_tags = {
    service = local.service_name
    owner   = var.owner
  }
}

resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Здесь мы воспользовались интерполяцией для создания составного имени, в переменной `local common_tags` мы сгруппировали несколько значений в объект и ниже использовали в другом ресурсе.

Начинайте новый проект вместе с хранением ***Terraform state*** в удаленном хранилище (если это не исключение, например, обучающий проект или проект с особыми требованиями).

Это избавит вас от дальнейших проблем с переносом и обновлением ***state***.

Ведите версионирование в хранилище, где у вас лежит ***Terraform state.***

Это поможет вам вернуться к предыдущей версии ***Terraform state*** в случае, если вы случайно примените неправильную/проблемную конфигурацию. Вы просто можете запросить в бакете (хранилище ***state***) предыдущую версию ***state***-файла и применить его.

Используйте модули по максимуму, где только это возможно.

Это поможет эффективно использовать большинство важных принципов разработки.

Не стоит присваивать постоянное значение переменной (харкодить) вместо того, чтобы передавать значение динамически.

При большом количестве однотипных ресурсов, вам придется менять значение в каждом из них, вместо того, чтобы поменять значение в переменной один раз, для всех однотипных ресурсов.

Пишите понятное описание для переменных, а также удобочитаемое имя.

Прозрачность кода увеличится. Из описания переменных (имя, тип, описание, значение по умолчанию) удобно создавать автоматическую документацию.

Например, с помощью инструмента под названием ***Terraform-docs***. Его работу мы увидим в примерах ниже.

Конвенция и соглашения

Конвенция — это что-то вроде официального договора между участниками команды. Конвенция необходима, чтобы код был единообразным, выполнял свод соглашений при изменении участниками команды. Например, может быть конвенция об именовании переменных.

Линтеры и тестирование

Линтеры — это наборы программных средств, призванные привести к единому стилю ваш код, а также провести проверку на соответствии вашего кода определенному набору правил.