

# Основи в NixOS. Модулната система на Nix.

## Универсални конфигурации с Nix

Павел Атанасов    Камен Младенов

29.04.2025

- Създавахме много пакети за програми, написани върху разнообразни езици

# Раздел 1

## Операционна система, базирана на Nix

# Операционна система, базирана на Nix

- За сега използваме Nix само като допълнителна програма към вече съществуваща система
- Nix е пакетен мениджър, не можем ли да премахнем вградените и да оставим само Nix?
- Можем! Това наричаме NixOS. Обаче...

## Раздел 2

### Проблеми и решения

# Проблеми и решения

Трябва ни глобално състояние!

- Твърде е неудобно за всяко нещо да викаме `nix-shell`
- Нужна е допълнителна възпроизводимост: между включвания.

Не може да направим локални състояния, да рестартираме и да ги загубим.

- Трябва да инсталираме и конфигурираме драйвъри по специални начини
- Имаме да стартираме и менажираме много системни сървиси (daemons)

# Системата като една деривация

- Можем да дефинираме цялата система като една деривация.
- Нейният **builder** ще извършва всички нужни действия. Но ще трябва да му дадем и допълнителни права.
  - ▶ За всяко нещо се правят символични връзки от локации в хард-диска и `/nix/store`
  - ▶ Изключение правим при bootloader-a, където просто добавяме нова опция

# Получаваме някои интересни ефекти

- Можем да имаме няколко версии на системата, и да избираме при включване коя искаме
- Лесно можем да разпространим конфигурацията
- Лесно можем да направим виртуална машина за дадена система



## И някои важни проблеми

- Една система ще има **много** nix код в себе си, твърде много да очакваме потребителя сам да го поддържа
- Трудно е да включваме/изключваме/презаписваме стойности, голямото количество код прави промените сложни

## Раздел 3

### Концептуални основи за модули

# Концептуални основи за модули

- Решаваме проблемите чрез нов метод за разделяне и менажиране на много Nix код - модулната система
- Това е библиотека в `nixpkgs`, т.е. всичко е имплементирано чрез Nix
- С един модул дефинираме *конфигурационни опции*
- И *какво се случва*, когато подадем стойност на опция

# Псевдо-обяснение

- За да опростим обясненията си, в началото ще използваме лек *псевдокод*
- Модул е **функция**, която приема редица стойности и връща
  - ▶ какви стойности са позволени (име, валидност)
  - ▶ какво се случва със системата при подадена стойност

## Псевдокод

```
{...}@values: {  
  allowedValues = { ... };  
  systemConfig = { ... };  
}
```

- Примерно, можем спрямо подадената опция да добавяме глобални пакети

## Псевдокод

```
{ includeCurl ? false, ... }: {  
  allowedValues = {  
    includeCurl = {  
      type = types.bool;  
    };  
  };  
  systemConfig = {  
    globalPackages = if includeCurl then [ curl ] else [];  
  };  
}
```

- Или пък да избираме драйвъри

## Псевдокод

```
{ GPU ? "", ... }: {  
  allowedValues = {  
    GPU = types.string;  
  };  
  systemConfig = {  
    enabledDrivers = if GPU == "nvidia" then [ nvidiaDriver ]  
                     else if GPU == "amd" then [ amdDriver ]  
                     else [];  
  };  
}
```

- Трябва да са се породили няколко въпроса
  - ❶ Nix е динамично типизиран, как ограничаваме типове?
  - ❷ Нима всеки модул презаписва системни стойности? Ако искаме два модула да добавят неща в `globalPackages`?
  - ❸ Как хем получаваме конкретни стойности, хем връщаме техните ограничения (без да правим по-подробни проверки)?

# 1. Nix е динамично типизиран, как ограничаваме типове?

- Модулната система е библиотека
- Тя експлицитно прави проверки чрез `builtins` функции, като `isBool` или `isString`, преди изобщо да извика нашия модул (и да му подаде стойностите)



## 2. Нима всеки модул презаписва системни стойности? Ако искаме два модула да добавят неща в `globalPackages`?

- Системни конфигурации се композират/обединяват по специален (рекурсивен) начин
- Примитивни стойности като низове се презаписват, но контейнери като списъци и атрибутни множества се обединяват
- `{ systemPackages = [ curl ]; }` и `{ systemPackages = [ wget ]; }` ще се обединят в `{ systemPackages = [ curl wget ]; }`

### 3. Как хем получаваме конкретни стойности, хем връщаме техните ограничения (без да правим по-подробни проверки)?

- Магията идва от **fix** функцията.  
Тя връща *константна* стойност, която е *същата* като подадената.
- Тук е нужно по-задълбочено разбиране на функционално програмиране. Цялата схема идва от факта, че Nix е **лениво оценен**.
- Атрибути и елементи на списъци не са оценени (тяхната стойност не е потърсена и записана в паметта), докато не е нужно

- Представете си следния израз:

```
{ a = "foo"; b = "bar"; c = a + b; }
```

- Искаме да премахнем всички променливи. Коя стойност, само с константи, е еквивалентна на горната?
- { a = "foo"; b = "bar"; c = "foobar"; }

- Как го направихме?

- ▶ Всичко, което има константи го оставихме, без да го мислим
- ▶ Всичко, което има променливи
  - ❶ заменихме променливите с техните стойности
  - ❷ изпълнихме всички операции, докато стигнем до константа

- Можем да накараме Nix да извърши същата процедура

```
lib.fix (obj: { a = "foo"; b = "bar"; c = obj.a + obj.b; })  
/* Връща */  
{ a = "foo"; b = "bar"; c = "foobar"; }
```

- *fix* *допуска*, че *obj* и `{ a = "foo"; b = "bar"; c = obj.a + obj.b; }` трябва да имат еднакви стойности
- Тоест, успява да свърже *obj.a* с *a = "foo"* и *obj.b* с *b = "bar"*

- Къде идва ленивото оценяване? Представете си, че строим върната стойност атрибут по атрибут.
  - ▶ Създаваме празно атрибутно множество, итерираме през всички атрибути на върнатата стойност
  - ▶ Виждаме `a`, стойността е константа, директно добавяме `a = "foo"`
  - ▶ Виждаме `b`, стойността е константа, директно добавяме `b = "bar"`. До момента изобщо не сме погледнали подадената стойност.
  - ▶ Виждаме `c`, вътре има две променливи. Сега поглеждаме аргумента `obj.fix` ни казва, че `obj` е същото като върнатата.  
Търсим стойностите в досега построеното атрибутно множество. Намираме ги, заместваме, конкатенираме, имаме константа, записваме `c = "foobar"`.

- Тази схема си има очевидните лимитации
- Това хвърля грешка

```
lib.fix (obj: { a = obj.a; })
```

- Както и това

```
lib.fix (obj: { a = obj.b; b = obj.a; })
```

## Раздел 4

### Истинска дефиниция на модули



# Истинска дефиниция на модули

- Модул е **функция**, приемаща атрибутно множество и връщаща атрибутно множество

```
{ config, pkgs, ... }: {  
  imports = [ ... ];  
  options = { ... };  
  config = { ... };  
}
```

- Приетото има 2 *нужни* атрибута
  - ▶ `config` - **всички константни** стойности на атрибути в системата
  - ▶ `pkgs` - пакетите в системата
- Върнатото има 3 задължителни атрибута
  - ▶ `imports` - списък с пътища към други модули
  - ▶ `options` - имена и типове на позволените стойности (`allowedValues` в нашия псевдокод)
  - ▶ `config` - атрибути и стойности в системата (`systemConfig` в нашия псевдокод)

- Реалистичен пример:

```
{ config, pkgs, ... }: {  
  options = { includeCurl = ...; };  
  config = {  
    environment.systemPackages = if config.includeCurl  
      then [ pkgs.curl ] else [];  
  };  
}
```

- Някой модул трябва да е направил

```
{ config, pkgs, ... }: {  
  ...  
  config = {  
    includeCurl = true;  
    ...  
  }
```

- Опции се декларират чрез функцията `lib.mkOption`. Подава ѝ се атрибутно множество.
- Главните атрибути са `type` и `description`, името се определя в модула
- Някои от по-често използваните типове са:  
`bool`, `int`, `str`, `package` (деривация), `listOf TYPE`

```
someOption = lib.mkOption {  
  type = lib.types.int;  
  description = "Expects some number";  
};
```

- Подробен примерен модул

```
{ config, pkgs, ... }: {  
  options = {  
    includeCurl = pkgs.lib.mkOption {  
      type = pkgs.lib.types.bool;  
      description = "Should curl be added as a global package";  
    };  
  };  
  config = {  
    environment.systemPackages = if config.includeCurl  
      then [ pkgs.curl ]  
      else [];  
  };  
}
```

# Къде идва истинската логика?

- Модулите само ни позволяват да дефинираме и запишем стойности в системния `config` атрибут
- Обаче, къде тези опции се използват за нещо полезно? Къде нещо поглежда `environment.systemPackages` и инсталира изредените програми?
- За тази цел има “вградени” модули
- Те са “примитивните” модули, които дефинират опции чрез които да се обработва целия компютър
- Всичко останало стои над тях

## Раздел 5

Как се дефинира NixOS система?

# Как се дефинира NixOS система?

- Започва се от `/etc/nixos/configuration.nix` файла
- Това е първоначалния модул, който Nix гледа в NixOS
- В него използваме `imports`, за да разделим конфигурацията на по-малки модули



- Най-често дори не дефинираме опции, просто записваме (константни) стойности на атрибути
- Тогава (**и само тогава**) можем да изпуснем `config` атрибута и да пишем всички негови атрибути като част от върната стойност

- Това

```
{ config, pkgs, ... }: {  
  imports = [ ... ];  
  config = {  
    a = 10;  
    b = 20;  
  };  
}
```

е СЪЩОТО КАТО

```
{ config, pkgs, ... }: {  
  imports = [ ... ];  
  a = 10;  
  b = 20;  
}
```

## Раздел 6

Въпроси?