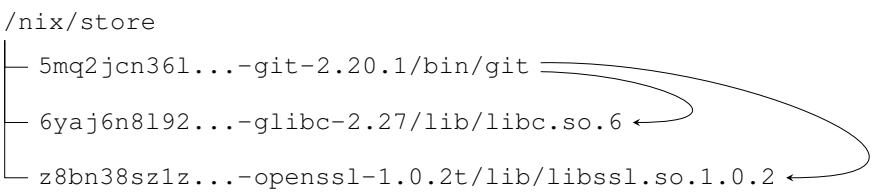
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2020
http://excel.fit.vutbr.cz

# Využití Nix/NixOps pro průběžnou integraci a nasazení software při vývoji

Tomáš Vlk*




**Abstrakt**

Tato práce se zabývá uplatněním funkcionálního balíčkovacího systému Nix a jeho ekosystému (NixOS, NixOps) pro CI/CD při agilním vývoji. Při použití těchto technologií jsou problémy způsobené odlišným prostředím prakticky eliminovány bez nutnosti kontejnerizace. Práce obsahuje popis možností a nedostatků Nix/NixOps a navrhuje obecný postup použití těchto technologií pro jednotlivé fáze agilního vývoje a CI/CD. Díky Nix/NixOps je implementace CI/CD velmi jednoduchá a celý proces je navíc reprodukovatelný. Výstupem práce je sada příkladů demonstrující použití Nix/NixOps v různých projektech, a která je dostupná jako open-source. Díky této sadě mohou vývojáři použít Nix rychle a jednoduše v jakémkoliv projektu, bez nutnosti studia velkého množství materiálů.

**Klíčová slova:** Continuous Integration — Continuous Deployment — Configuration Management — Infrastructure as Code — Nix — NixOS — NixOps

**Přiložené materiály:** nix-examples

*xvlkto00@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysokého učení technického v Brně*

## 1. Úvod

S agilním vývojem je úzce spjatý proces průběžné integrace a nasazení software (CI/CD). Pro spolehlivé otestování a nasazení aplikace je potřeba eliminovat problémy způsobené odlišným prostředím. V ideálním případě by prostředí, které používá vývojář, mělo být totožné s prostředím CI/CD serveru, na kterém je software testován a s produkčním serverem, na kterém je následně nasazen. Drobné odchylky mezi prostředími mohou způsobit, v lepším případě chybu při zpracovávání CI/CD serverem, ale v horším případě nefunkční nasazenou aplikaci na produkčním serveru.

Problémy spojené s nasazením se často řeší pomocí kontejnerizace aplikací. Software je na CI/CD serveru testován v kontejneru a na produkčním serveru je nasazen ve stejném kontejneru. Díky tomu jsou problémy s prostředím takřka odstraněny. Používání kontejnerů ale není ideální. Ve spoustě případů není potřeba aplikace v systému kompletně izolovat do samostatných kontejnerů. Musí se pak řešit orchestrace, monitorování, sdílení dat mezi kontejnery a v neposlední řadě může být i problém s rychlostí [1].

Ekosystém balíčkovacího systému Nix nabízí deklarativní zápis balíčku, celého systému nebo dokonce celé infrastruktury. To vše s jistotou reprodukovatelnosti. Díky tomu nenastávají problémy s prostředím a proces průběžné integrace a nasazení software je velmi spolehlivý a agilní vývoj rychlejší. Dokonce je i možné použít Nix velmi efektivně na testování nasazení aplikace, ještě před samotným nasazením.

Nevýhodou Nix je, že neexistuje kompletní a jednotný přehled použití pro jednotlivé aplikace různého

druhu. Například webové, mobilní, distribuované atd. Dále, kvůli nutnosti definování každého balíčku pomocí Nix, je nutné mít odlišné příklady pro jednotlivé programovací jazyky a nástroje pro sestavení, případně obecný návod, jak jakýkoliv projekt, používající oficiálně nepodporované technologie, zprovoznit pomocí Nix.

Je možné nalézt několik málo projektů, které demonstrují použití Nix. Například projekt nixtodo[1] nebo TodoMVC[2]. Oba tyto příklady jsou ale pro začátečníka velmi komplikované, protože obsahují pro jeden projekt více než 15 `.nix` souborů. Navíc autoři těchto příkladů mají s Nix velké zkušenosti a mají tendenci používat Nix i pro jiné účely, než pro které byl původně určen.

Cílem této práce je vytvoření obsáhlé sady příkladů demonstrující komplexní použití Nix. Přitom je ale kladen důraz na jednoduchost a jednotnost příkladů. Každý příklad obsahuje stejné `.nix` soubory se stejným významem a je snaha o minimalizaci rozdílů mezi různými technologiemi. Je ale potřeba přistupovat odlišně ke správě závislostí. U některých sestavovacích nástrojů mohou být závislosti projektu dodány jako samostatné balíčky, ale u jiných musí být dodány všechny jako jeden balíček. Díky jednotnému rozhraní je pak možné spustit proces průběžné integrace a nasazení jedním příkazem a pro integraci s klasickými CI/CD řešeními stačí už jenom Nix nainstalovat.

Čtenář se v této práci dozví, jak Nix funguje, jaké má vlastnosti a díky příkladům, které jsou výstupem této práce, bude moci Nix ihned použít ve svých projektech. Příklady pokrývají valnou většinu typických projektů od desktopových aplikací až po aplikace pro mobilní či vestavěné platformy. Díky funkcionálnímu jazyku a nástrojům vytvořeným okolo Nix, není pak problém tyto aplikace škálovat, či komponovat do větších celků.

# 2. Nix a jeho ekosystém

Povědomí o Nix je zatím spíše raritou, a proto jej a technologie, které s ním bezprostředně souvisí, tato kapitola popisuje. Protože je ale vývoj Nix velmi rychlý a jeho možnosti jsou obrovské, obsahuje tato kapitola hlavně zásadní principy a vlastnosti jednotlivých technologií. Bez tohoto úvodu do Nix by nemusely být následující kapitoly plně srozumitelné, pokud ale čtenář Nix zná, může tuto kapitolu přeskočit.

## 2.1 Standardní balíčkovací systémy

Standardní správce balíčků, jako je APT pro distribuce založené na Debianu, nebo DNF pro distribuce používající RPM formát balíčků, jsou *stavové*. Pokud se balíčky aktualizují, tak se přepisují soubory nainstalovaných balíčků a mění se stav systému. Změna se nemusí povést a špatně se pak řeší rollback, jestliže nebyly staré soubory zazálohovány. Společně s tím nastává i problém atomičnosti, některé balíčky mohou být v určitý čas už aktualizované a některé ještě ne. [2]

Dalším problémem standardních balíčkovacích systémů je nemožnost mít několik verzí stejného programu/knihovny. Může nastat situace, kdy budou dva balíčky závislé na jiné verzi knihovny, přičemž jenom jedna verze této knihovny může být v systému aktivní. Takový problém je ve standardních balíčkovacích systémech neřešitelný a nazývá se jako *dependency hell*. Dokonce samotná verze nemusí stačit a může být vyžadována přímo nějaká varianta (například přeložená s jinou konfigurací). Varianty ale standardní správci balíčků vůbec neidentifikují, rozlišují pouze mezi verzemi. Navíc není jednoduché zreprodukovat sestavení již sestaveného balíčku[3], takže každé sestavení může mít stejné důsledky jako jiná varianta balíčku.

Problém několika verzí a s ním spjatý dependecy hell se snaží řešit správci balíčků Snappy a Flatpak. Fungují tak, že v instalovaném balíčku jsou zabaleny současně i všechny jeho závislosti a instalovaný balíček pak používá jenom svoje závislosti, ne ty nainstalované globálně v systému. Problémem tohoto přístupu je ale velká velikost balíčku. Balíčky nainstalované pomocí Snappy a Flatpak mezi sebou nesdílí závislosti nebo jenom minimálně. Dalším problémem je bezpečnost, když je nalezena bezpečností díra v nějaké závislosti. Pokud závislost není sdílená mezi balíčky, je obtížné opravit nebo aktualizovat tuto závislost u všech balíčků. V neposlední řadě nemusí vždy fungovat správně vizuální integrace se systémem, jako jsou vlastní témata nebo fonty.

Velká výhoda výše zmíněných balíčkovacích systémů je jejich rychlost. Balíčky jsou distribuované už zkompilované a takové správce balíčku lze pak označit jako *binary based*. Naproti tomu existují *source code based* balíčkovací systémy. V takových balíčkovacích systémech probíhá sestavení balíčku u uživatele a až teprve se instaluje do systému. Někdy se balíčky mohou distribuovat primárně v binárním formátu, ale uživatel může zvolit i lokální kompilaci a následnou instalaci. Takový balíčkovací systém lze pak označit jako *hybridní*.

[1] https://github.com/basvandijk/nixtodo

[2] https://github.com/nix-community/todomvc-nix

[3] https://reproducible-builds.org

## 2.2 Nix: čistě funkcionální správce balíčků

Správce balíčků Nix je hybridní balíčkovací systém. Pro každý balíček existuje speciální předpis, jak se má sestavit a Nix garantuje reprodukovatelnost sestavení. Součástí sestavení bývá často i kompilace a ta může trvat velmi dlouho. Proto jsou již sestavené balíčky uloženy v cache (na uživatelům dostupných serverech). Z pohledu uživatele se tak Nix jeví jako binary based balíčkovací systém.

Na rozdíl od standardních balíčkovacích systémů nepoužívá Nix standardní uložiště programů `/bin` nebo `/usr/bin`. Všechny balíčky jsou instalovány do *Nix store*, což je jednoduše jenom adresář, ve výchozím nastavení `/nix/store`. V uložišti ukládá tři typy objektů: derivace, výstupy derivací (derivát) a zdrojové soubory. *Derivace* je speciální objekt, který popisuje prostředí, zdrojové soubory a příkazy potřebné k sestavení nějakého balíčku. Vykonáním derivace získáme *výstup derivace*, tedy samotný balíček. Pro sestavení jsou potřeba již zmíněné *zdrojové soubory*, které jsou v `/nix/store` uloženy samostatně. Pojem balíček má v Nix obecnější význam, nemusí se jednat jen o spustitelný program. Balíček je výstup derivace, který může obsahovat cokoliv.

Každý objekt je uložen v `/nix/store` pod unikátním identifikátorem, skládajícího se z hashe a názvu samotného objektu. Používá se SHA-256 hash, který je oříznut na 160 bitů a zakódován pomocí notace Base32 [3]. Pokud se jedná o výstup derivace, je hash získán ze všech vstupů, které byly použity k sestavení (hash derivace). U zdrojových souborů a derivací je hash vytvořen jednoduše z obsahu souboru.

Díky hashi v názvu objektu, který je takřka bezkolizní a tedy unikátní, je možné, aby byl balíček uložen v `/nix/store` v různých verzích a variantách zároveň. Při instalaci nebo aktualizaci se nepřepisují soubory a nemění se závislosti již nainstalovaných balíčků, jako je tomu například u APT nebo RPM, ale jen se přidávají nové. Na rozdíl od klasických správců balíčků, tak Nix nemění stav systému a je tedy *bezstavový*.

Na obrázku 1 je příklad objektů v `/nix/store`. Každý objekt je soubor nebo adresář umístěný přímo v adresáři `/nix/store`. Šipky v obrázku znázorňují závislosti mezi objekty. Například objekt `rr3y0c6zyk...-hello-2.10` je závislý na objektu `6yaj6n8l92...-glibc-2.27`. Konkrétně obsahuje tento binární soubor řetězec `/nix/store/6yaj6n8l92...-glibc-2.27/lib/libc.so.6`. Závislosti jsou v binárních souborech uvedeny explicitně. Dynamický linker pak nevyhledává knihovny například v adresáři `/usr/lib`, ale rovnou je načte [2].


**Obrázek 1.** Objekty v `/nix/store` a závislosti mezi nimi. Pro lepší přehlednost jsou hashe v názvech objektů zkrácené a některé soubory jsou vynechané.

Aby bylo zaručeno, že objekt není závislý na jiných objektech mimo `/nix/store`, používá se k sestavení derivací čisté prostředí a při spuštění programu se používá upravený dynamický linker [4]. Protože názvy závislostí obsahují hash, jsou závislosti vždy explicitně a přesně určeny. Pokud vývojář zapomene specifikovat nějakou závislost, tak s velkou pravděpodobností program nepůjde sestavit nebo nebude fungovat. Nestane se ale to, že by program fungoval kvůli tomu, že tato závislost je v systému ve standardní cestě. To je dobře, protože díky tomu není tato závislost opomenuta. Závislosti jsou textově zapsány ve výstupech derivací a díky hashi se dají v souborech vyhledat. Použití Nix tak velmi pomáhá ke kompletní specifikaci všech závislostí balíčku [3].

### Deklarativní zápis balíčků

Každý standardní balíčkovací systém používá nějaký formát pro popis metadat a závislostí balíčků. Při deklarování závislostí je potřeba přesně specifikovat v jaké verzi a variantě tato závislost má být. Většinou tento formát umožňuje pouze deklarování verze závislosti. Jak bylo zmíněno výše, Nix používá kryptografické hashe, které jednoznačně určují verzi, variantu i způsob sestavení balíčku. Při definování nového balíčku by tedy stačilo deklarovat jeho závislosti ve formě hashů. Tyto hashe ale slouží k identifikaci výstupu, a pokud by nějaký tento hash nebyl v `/nix/store` a nebyl ani v cache, musela by se tato závislost znovu sestavit. Jenže chybí zpětná informace, jaká derivace produkuje daný hash a není tedy možné tuto derivaci sestavit a uložit výstup v `/nix/store`.

V Nix se proto v popisu balíčku závislosti nedeklarují, ale definují se. To znamená, že v případě po-

třeby mohou být vždy znovu sestaveny. Z toho důvodu musejí být součástí definice i tranzitivně definice všech závislostí. K tomuto účelu byl vytvořen stejnojmenný speciální programovací jazyk Nix, který umožňuje právě deklarativně popsat, jak se má balíček sestavit, jaké má závislosti a navíc, umožňuje i vytvářet kompozice balíčků.

Jazyk Nix je silně dynamicky typovaný funkcionální jazyk s podporou *lazy evaluation*. Je velmi jednoduchý a nenabízí tolik možností jako jiné funkcionální jazyky. Jedná se o *DSL (Domain Specific Language)* primárně určený pro deklarativní zápis balíčků. Důvodem pro jeho vytvoření byla hlavně možnost snadno tvořit varianty balíčků a vytvářet grafy závislostí mezi derivacemi.

Jednoduchý příklad definice balíčku je ve výpisu 1. Celý balíček je zadefinován jako funkce. Díky tomu může být při změně parametru balíček z funkce vrácen v jiné variantě. Funkce je *čistá*, nemá přístup k ničemu globálnímu a nemá žádné vedlejší účinky. Právě proto je Nix čistě funkcionální správce balíčků. Pokud je balíček na něčem závislý, musí obsahovat parametr, který tuto závislost zprostředkuje. To umožňuje snadno sestavit balíček s jinou verzí nebo variantou dané závislosti. Oficiální repozitář, obsahující takto definované balíčky dostupné pro Nix, je *Nixpkgs*[4].

```
{ stdenv, fetchurl, someDependency }:

stdenv.mkDerivation rec {
  pname = "example";
  version = "1.0";

  src = fetchurl {
    url =
  "https://example.org/${pname}-${version}";
    sha256 = "0ssi1wpafc...7c9lng89nd";
  };

  buildInputs = [ someDependency ];

  buildPhase = ''
    gcc example.c -o example
  '';
}
```

**Výpis 1.** Jednoduchý příklad definice balíčku. Hash zdrojových souborů byl pro přehlednost zkrácen.

Sestavení derivace probíhá v *sandboxu* – v čistém a reprodukovatelném prostředí. Nejdříve se vytvoří dočasný pracovní adresář. Dále jsou vymazány všechny proměnné prostředí a některé jsou znovu nastaveny na základě popisu derivace (`$out`, `$PATH`, ...). Poté je vykonán sestavovací skript, a nakonec jsou všechny výstupy derivace uloženy v `/nix/store`. U všech souborů uložených v `/nix/store` jsou dokonce znovu nastavena práva přístupu a nastaveny časové značky na hodnotu 1 (00:00:01 1/1/1970 UTC). Samotný sestavovací skript nemá přístup k souborům v systému. Má přístup pouze k `/nix/store`, svému dočasnému pracovnímu adresáři a k upraveným variantám systémových souborů, jako jsou soubory v adresáři `/proc` nebo `/dev`. Spuštěný proces je oddělen od systému a ostatních procesů pomocí *namespaces*. Má například vlastní namespace pro číslo procesu, souborový systém nebo síťová zařízení. Kvůli determinismu nemá skript vůbec žádný přístup k internetu. Navíc je každé sestavení derivace spuštěné pod jiným speciálním uživatelem, aby se více spuštěných sestavení nemohlo ovlivňovat. [3]

Pokud derivace specifikuje hash svého výstupu ve speciálním atributu `outputHash`, jedná se o *fixed output derivation (FOD)*. Sestavení se provádí stejně jako u normální derivace, jen je hash výstupu již předem znám. Pokud se hash výstupu derivace neshoduje s uvedeným hashem, skončí sestavení neúspěšně. FOD derivace má dokonce i se zapnutým sandboxem přístup k internetu. Reprodukovatelnost není narušena díky referenční transparentnosti celé derivace – je jedno co derivace provádí, její výstup musí odpovídat specifikovanému hashi. Například funkce `fetchurl` použitá ve výpisu 1 je FOD.

### 2.3 NixOS: čistě funkcionální linuxová distribuce

V klasických unix-like operačních systémech většinou není možné mít nainstalovaný program ve více verzích nebo variantách. Jedním z problémů je globální uložiště nainstalovaných programů v adresáři `/bin`. Tento problém se často řeší přidáním čísla verze k názvu programu a vytvořením symbolického odkazu na aktivní verzi. Jenže každá verze programu může vyžadovat jiné verze konfiguračních souborů. Může se jednat o jiný formát nebo jiné možnosti nastavení. Tudíž nastává stejný problém, akorát s globálním uložištěm konfiguračních souborů v adresáři `/etc`.

Hlavním problémem klasických unix-like systémů, je použití stavového správce balíčků. Při použití stavového správce balíčků jsou balíčky a konfigurace instalovány a upravovány pomocí imperativních kroků, které modifikují globální stav systému [5]. To ztěžuje sledování změn, reprodukovatelnost a možnosti provedení rollbacku. Navíc aplikování několika imperativních transformací není atomické.

Existují komplexní řešení, která automatizují tyto imperativní kroky a umožňují spravovat několik systémů zároveň. Patří do kategorie nástrojů pro správu

[4] https://github.com/NixOS/nixpkgs

konfigurace (CMS) a jsou to například nástroje Ansible a Puppet. Tyto nástroje ale neznají přesný stav sytému a jenom provádí zadané příkazy. Například v konfiguraci Ansible může být příkaz pro instalaci nějakého balíčku. Ansible tento příkaz vykoná a v systému bude tento balíček nainstalován. Později může být v konfiguračním souboru Ansible instalace tohoto balíčku odebrána. Po opětovném spuštění Ansible ale není provedena odinstalace na cílovém systému. Stav systému se tak imperativně mění a může se postupem času lišit od požadovaného stavu. Jedná se o tzv. *konvergentní model* [6].

Jak už název napovídá, *NixOS* používá bezstavový správce balíčků Nix. Celý systém je deklarativně popsán pomocí jednoho Nix výrazů a je možné ho reprodukovatelně sestavit. Při změně popisu systému se systém znovu sestaví a je ve stavu čisté instalace. Z pohledu správy konfigurace se tak jedná o *kongruentní model* [6].

V NixOS jsou všechny komponenty systému (včetně jádra, balíčků a konfiguračních souborů) sestaveny pomocí Nix [7]. Celý systém je sestaven na základě vyhodnocení výrazu v hlavním konfiguračním souboru systému `/etc/nixos/configuration.nix`. Příklad takovéhoto souboru je ve výpisu 2. Možnosti konfigurace celého systému jsou definovány v *modulech*. Ty jsou stejně jako hlavní konfigurační soubor napsané v jazyku Nix a jsou importovány (viz řádek 3 ve výpisu 2).

Konfigurační soubory programů jsou po sestavení uloženy v `/nix/store` stejně jako balíčky. Soubory v `/nix/store` jsou neměnné, takže konfigurační soubory nelze poté upravovat. Pokud je potřeba změnit nějakou konfiguraci, musí se upravit konfigurace v jazyku Nix a znovu je nechat sestavit. Díky tomu, má tento systém podobné vlastnosti jako správce balíčků Nix. Je reprodukovatelný, podporuje atomické změny a je možné jednoduše provést rollback. Kvůli použití balíčkovacího systému Nix, avšak NixOS nevyhovuje standardu Filesystem Hierarchy Standard. [5]

## 2.4 NixOps: infrastruktura jako kód založená na Nix

Při přístupu DevOps je potřeba průběžně upravovat infrastrukturu a automatizovaně konfigurovat jednotlivé stroje. Infrastruktura může být provozována ve vlastní síti nebo u cloudových poskytovatelů. V předchozí podsekci byl představen deklarativní a reprodukovatelný systém NixOS. Lze na něj nahlížet nejen jako na linuxovou distribuci, ale i jako na nástroj pro CMS. Pokud je provozovaných NixOS systémů více, které jsou mezi sebou propojené, tak je potřeba je konfigurovat společně.

```
{ config, pkgs, ... }: {

  imports = [ ./hello.nix ];

  fileSystems."/mnt" = {
    fsType = "ext4";
    device = "/dev/sda1";
  };

  services.openssh.enable = true;
  environment.systemPackages = with pkgs; [
    wget vim
  ];
}
```

**Výpis 2.** Příklad konfigurace systému NixOS. V systému je připojené zařízení `/dev/sda1`, povolena služba `openssh` a nainstalovány balíčky `wget` a `vim`.

Další technologií postavenou nad Nix je *NixOps*, která slouží jako nástroj pro IaC (Infrastructure as Code), neboli automatické nasazení infrastruktury. Tento nástroj je úzce spjatý s NixOS a umožňuje deklarativně popsat infrastrukturu jednotlivých strojů a jejich konfigurace. NixOps poté vykoná nutné kroky nebo akce k tomu, aby tento popis byl realizován. Uchovává si stav v jakém stroje jsou a při znovu nasazení provádí jenom ty akce, které jsou potřeba [8].

NixOps podporuje několik cílových prostředí pro nasazení:

- Amazon EC2
- Digital Ocean
- Google Compute Engine
- Hetzner
- Libvirtd (Qemu)
- Microsoft Azure
- NixOS
- VirtualBox VM

Podobně jako u deklarativního zápisu konfigurace NixOS, může být zápis infrastruktury strukturován do menších celků. K zápisu se používá jazyk Nix, stejně jako u NixOS a popisu balíčků. Příklad deklarativního zápisu infrastruktury je ve výpisu 3.

I když v předchozím textu převažují pozitivní vlastnosti, má Nix i několik nevýhod. Některé z nich se mohou v budoucnu vyřešit, ale některé už z principu návrhu Nix řešitelné nejsou:

- Pro vytvoření balíčku nebo integraci projektu s Nix musí být vytvořen Nix výraz.
- Jazyk Nix je DSL pro balíčky, ale jazyk nepoužívá žádný koncept balíčků (balíčky jsou funkce).
- Balíček jako funkce deklaruje nejen závislosti, ale i jiné parametry, což není rozlišeno.

```
{
  webserver = {
    deployment.targetEnv = "virtualbox";
    services.httpd.enable = true;
    services.httpd.virtualHosts = {
      "example.org" = {
        documentRoot = "/data";
    };
    fileSystems."/data" = {
      fsType = "nfs4";
      device = "fileserver:/";
    };
  };

  fileserver = {
    deployment.targetEnv = "virtualbox";
    services.nfs.server.enable = true;
    services.nfs.server.exports = "...";
  };
}
```

**Výpis 3.** Příklad infrastruktury sestávající se z webového serveru a souborového serveru. Webový server má připojený obsah souborového serveru do adresáře `/data`. Oba dva stroje budou virtualizovány pomocí VirtualBoxu.

- Pokud se změní některá nízkoúrovňová knihovna (například glibc), tak se musí skoro všechno znovu sestavit.
- Uložiště `/nix/store` není CAS (Content addressable storage) pro všechny objekty. Zdrojové soubory lze adresovat na základě jejich obsahu, ale výstupy derivací nikoliv.
- Problémová a zatím koncepčně nevyřešená je manipulace a uložení hesel a ostatních klíčů v konfiguračních souborech NixOS. Je potřeba si dát pozor, aby se klíče a hesla nezkopírovala do `/nix/store` při vyhodnocování výrazu.

# 3. Návrh použití Nix pro průběžnou integraci a nasazení software

V úvodu této práce byly nastíněny některé problémy existujících řešení CI/CD používaných při agilním vývoji. Většinou je zapotřebí použít při průběžné integraci a nasazení kontejnery, které nejsou vždy žádoucí.

Bez kontejnerizace není snadné reprodukovat prostředí CI/CD serveru a reprodukovatelně spustit proces průběžné integrace a nasazení lokálně. Není možné tak spolehlivě ověřit úspěšnost některých fází, ještě předtím, než jsou spuštěny na CI/CD serveru. Podobně je i problém bez kontejnerizace zajistit stejné prostředí použité na produkčním, CI/CD, testovacím nebo jiném serveru. To snižuje důvěru ke správnému nasazení a fungování aplikace. Pokud nastane chyba při nasazení nové verze aplikace, nemusí být vždy jednoduché provést rollback a to z toho důvodu, že se přepisují soubory aplikace nebo systému (používá se stavový správce balíčků). Nakonec není možné jednoduše změnit CI/CD server, protože každé CI/CD řešení, jako je Jenkins nebo CircleCI, má svůj vlastní formát zápisu jednotlivých fází.

Právě výše zmíněné problémy je možné eliminovat pomocí Nix. I když se tato práce zabývá pouze použitím Nix pro jednotlivé fáze agilního vývoje a CI/CD, je možné použít Nix i v jiných metodikách.

## 3.1 Použití Nix pro sestavení

Ve fázi sestavení jsou zásadní závislosti projektu. Podle podpory dané technologie v Nixpkgs je možné dodat k projektu závislosti třemi způsoby:

1. Závislosti jsou v repozitáři Nixpkgs jako balíčky. Například pro Haskell je dostupná většina závislostí.
2. Derivace závislostí je možné vygenerovat. Závislosti jsou zapsané strukturovaně v nějakém souboru a pomocí nástroje `*2nix` se vygenerují derivace závislostí. Příkladem je JavaScript a nástroj `node2nix`.
3. Závislosti se musí získat externím nástrojem a do definice balíčku dodat jako jedna derivace (FOD). Takový přístup se musí použít například u PHP a sestavovacího nástroje composer.

Poslední zmíněný způsob má několik problémů. Za prvé je nutné manuálně specifikovat hash pro výstup derivace závislostí. Za druhé nemusí externí nástroj podporovat reprodukovatelnost a po každém spuštění mohou být závislosti dodány v trochu jiné verzi. V takovém případě se musí vypnout sandbox a ztrácí se většina dobrých vlastností Nix.

## 3.2 Použití Nix pro testování

V rámci sestavení aplikace mohou být provedeny jednotkové testy. Po sestavení mohou přijít na řadu testy, které testují aplikaci jako celek. Tyto testy mohou být mnoha druhů a mohou mít různé potřeby. Jsou to například systémové testy, stress testy atd.

Většina testů by mělo být možné spustit pomocí shell příkazů. Takové testy jde v Nix provést pomocí speciální funkce `runCommand`. Příklad použití této funkce je ve výpisu 4. Může se klidně testovat i pomocí nějakého frameworku. Stačí tento framework uvést jako závislost, aby byl dostupný. Závislosti potřebné pro testování jsou tak oddělené od závislostí aplikace. Příkaz může i spustit virtuální stroj, a testovat tak aplikaci na jiných platformách.

```
{ pkgs ? import <nixpkgs> {} }:
with pkgs; jobs = rec {

  build = import ./default.nix {
    inherit pkgs;
  };

  tests = pkgs.runCommand
    "tests" { buildInputs = [ build ]; } ''
      echo "Hello world" > expected
      hello > given
      diff expected given > $out/result
    '';

  buildRaspberryPi = import ./default.nix {
    pkgs = pkgsCross.raspberryPi;
  };

  dockerImage = dockerTools.buildImage {
    name = "hello";
    tag = "latest";
    contents = [ build ];
    config = { Cmd = [ "/bin/hello" ]; };
  };
}
```

**Výpis 4.** Testování pomocí speciální derivace pro vykonání posloupnosti shell příkazů, příklad sestavení pro RaspberryPi pomocí cross-compilation a příklad vytvoření Docker image s aplikací.

Dále je možné využít možnosti, které nabízí NixOS. Repozitář Nixpkgs obsahuje pomocné funkce pro spuštění několika virtuálních NixOS systémů, které mohou být navíc propojené. Poté je možné například testovat dostupnost služeb na těchto systémech.

V neposlední řadě je možné využít pro účely testování i NixOps. Stačí nastavit jako cílové prostředí VirtualBox. Celá infrastruktura se pak zprovozní lokálně a může se dále ověřovat její korektnost. Případně je možné nasadit vše na kopii produkční infrastruktury. Protože je vše reprodukovatelné (aplikace, prostředí, infrastruktura), je tímto způsobem možné otestovat nasazení aplikace ještě před samotným nasazením. Spolehlivě a reprodukovatelně sestavit celou infrastrukturu, prostředí a balíčky umí jenom NixOps, jiné nástroje to nedokážou.

Mít ale další stejnou produkční infrastrukturu pro testování může být velmi nákladné. Je proto možné použít kompromisní řešení: nasadit část infrastruktury na jeden NixOS systém a zbytek virtualizovat nebo nasadit stejně jako v produkci. Záleží pak na typu aplikace, do jaké míry je možné se odchýlit od produkční infrastruktury pro systémové testy.

### 3.3 Použití Nix pro release

Sestavení může být provedeno v různých variantách. Stačí ve funkci balíčku deklarovat parametr pro změnu varianty, nebo zpětně přepsat nějaký atribut balíčku. Repozitář Nixpkgs dále nabízí mnoho pomocných funkcí, které usnadňují vytváření instalačních souborů, jako je .deb, .rpm, .snap, ... Pro použití aplikace v kontejnerech je možné jednoduše vygenerovat Docker image nebo OCI kontejner obsahující jenom aplikaci s jejími závislostmi a nic jiného. Díky podpoře cross-compilation je také velmi snadné sestavovat aplikaci pro jiné platformy. Příklad je ve výpisu 4.

Podobně jako při testování, je možné spustit jakýkoliv jiný příkaz a vygenerovat ve fázi release jiný výstup. Může se jednat například o generování programové dokumentace.

### 3.4 Použití Nix pro nasazení

Pro nasazení je možné použít správce balíčků Nix, je ale lepší použít NixOS, protože je pak možné s nasazovaným softwarem upravit i konfiguraci systému. Jednoduše se jenom aktivují a nastaví potřebné moduly aplikace. Konfigurace NixOS může být chápána jako analogie Dockerfile, protože z ní lze jedním příkazem vygenerovat celý systém, který bude obsahovat nasazovaný software. Na cílový NixOS pak už jenom stačí dodat tuto konfiguraci a přepnout na ni. Na rozdíl od Dockeru nebo virtualizace nemá toto řešení žádný vliv na výkon.

Pro jednotné prostředí se často používají kontejnery. Používání kontejnerů pro oddělení dat, systémových prostředků atd. není vždy zapotřebí. Naopak je většinou problém se sdílením dat mezi jednotlivými kontejnery a vyjádření jejich závislosti mezi nimi. I když použití NixOS a použití kontejnerů není vzájemně výlučné, nejsou kontejnery při použití NixOS prakticky potřeba. Oddělení a reprodukování prostředí pro aplikace zajistí samotný Nix potažmo NixOS. Pro jejich běh a orchestraci pak není potřeba další software a je možné využít standardní prostředky systému, jako je systemd. Virtualizační vrstva tak může být úplně eliminována a zároveň je zachovaná stejná funkcionalita jako při použití kontejnerů.

Nejkomplexnějším způsobem nasazení pomocí Nix je využít NixOS společně s NixOps. V takovém případě je možné nasadit software, změnit konfiguraci systému a případně změnit i infrastrukturu. Namísto NixOps je možné společně s NixOS samozřejmě použít i jiné nástroje, například Terraform.

## 4. Implementace sady příkladů

Příklady demonstrují použití Nix/NixOps pro různé druhy aplikací: desktopové, mobilní, webové, vícevrstvé, distribuované, ... Součástí je i krátký informativní soubor obsahující požadavky pro použití a přehled dostupných příkladů. Vše je zveřejněno jako open-source pod licencí GPL na autorově účtu na GitHubu[5].

### 4.1 Dostupné technologie v příkladech

Při implementaci příkladů, jsem se snažil pokrýt nejpopulárnější technologie. Příklady budu postupně doplňovat a upravovat tak, aby byly funkční s aktuální verzí Nix balíčků. Zatím příklady pokrývají tyto jazyky a sestavovací nástroje:

- C – Autotools
- Go – Go modules
- Haskell – Cabal
- Java – Maven, Ant, Gradle
- JavaScript – NPM
- PHP – Composer

Pro každý programovací jazyk může být používáno více sestavovacích nástrojů, které mohou nebo nemusí i fungovat jako nástroj pro správu závislostí. Například pro Javu je Ant pouze sestavovací nástroj bez správy závislostí a Ivy je pouze nástroj pro správu závislostí. Oproti tomu například Maven je sestavovací nástroj a zároveň i nástroj pro správu závislostí.

Právě reprodukovatelné získání závislostí je pro integraci projektu s Nix zásadní. Existují tři způsoby získání závislostí pomocí Nix, jak bylo popsáno výše v sekci 3.1. Přehled dostupných způsobů integrace v sadě příkladů, pro jednotlivé nástroje pro správu závislostí, je v tabulce 1.

| | Nixpkgs | `*2nix` | FOD |
|---|---|---|---|
| Go modules | | | ✓ |
| Cabal | ✓ | | |
| Maven | | | ✓ |
| Gradle | | | ✓ |
| NPM | | ✓ | ✓ |
| Composer | | | ✓ |

**Tabulka 1.** Dostupné příklady úrovně integrace s Nix pro jednotlivé nástroje pro správu závislostí.

Každý příklad je vytvořen tak, aby byl plně samostatný. Mnoho kódu se tak opakuje, ale adresář s projektem díky tomu není závislý na jiných souborech zvenčí. Kromě zdrojových souborů, obsahuje každý příklad pět až devět `.nix` souborů pro integraci s Nix a CI/CD:

- `nixpkgs.nix` – zafixování verze a konfigurace Nixpkgs. V tomto souboru je také možné zadefinovat vlastní balíčky.
- `app.nix` – popis sestavení balíčku jako funkce. Tento soubor je nezávislý na použitém repozitáři balíčků a může být snadno integrován do hierarchie Nixpkgs nebo do vlastního repozitáře balíčků.
- `default.nix` – volání funkce balíčku ze souboru `app.nix` a dodání závislostí z `nixpkgs.nix`.
- `shell.nix` – obaluje balíček v souboru `default.nix` pro vývoj. Mění atributy výsledného balíčku, jako je například atribut `src` a přidává nástroje potřebné pouze pro vývoj.
- `ci.nix` – sada dostupných akcí pro CI/CD.
- `module.nix` – NixOS modul aplikace, který může být jednoduše integrován do stromu NixOS modulů v repozitáři Nixpkgs, nebo do vlastního repozitáře modulů.
- `cd.nix` – logický popis infrastruktury.
- `cd-*.nix` – fyzický popis infrastruktury. Například soubor `cd-vbox.nix` může popisovat nasazení do virtuálních strojů pomocí VirtualBoxu a soubor `cd-cloud.nix` může popisovat způsob nasazení k nějakému z cloudových poskytovatelů.

Jednotlivé derivace (akce) lze komponovat do sebe a tvořit mezi nimi závislosti. Jedna z vlastností Nix je, že derivace, které na sobě nejsou závislé, mohou být vykonány paralelně. To umožňuje elegantní definování, které akce mohou být vykonány paralelně, a které musí být vykonány sekvenčně. Například lze vykonat různé testy nebo vytvořit různé instalační soubory zároveň, ale proces nasazení může být vykonán až po úspěšném vykonání všech testů. Příklad takové definice je ve výpisu 5. V samotném CI/CD systému se už pak vůbec nemusí specifikovat fáze integrace a nasazení a spustí se už jenom jedním příkazem akce `pipeline`.

### 4.2 Zhodnocení použití Nix pro CI/CD

V závěru sekce 2 byly shrnuty nevýhody Nix. Jednou z nich je i nutnost, pro každý projekt nebo balíček, vytvořit Nix výraz. Právě tuto nevýhodu řeší předkládaná sada příkladů, která nabízí obecný postup vytvoření balíčku pro jakýkoliv projekt a zároveň demonstruje komplexní použití Nix pro vybrané technologie. Pro vývojáře by neměl být problém najít v příkladech svou doménu zájmu a příklad snadno modifikovat a adaptovat na svůj projekt. Ne u všech technologií je ale zatím možné dodat závislosti pomocí Nix jednotlivě a musí být dodány jako jeden balíček.

[5] https://github.com/vlktomas/nix-examples

```
jobs = rec {
  ...
  # toto může být vykonáno paralelně
  long-tests = [
    stressTests platformsTests
  ];

  # toto jedině sekvenčně
  pipeline = mkPipeline [
    build tests release deploy
  ];

  # vytvoření závislostí mezi fázemi
  mkPipeline = phases: (
    foldl mkDependency null phases
  );
  mkDependency = prev: next:
    next.overrideAttrs
      (oldAttrs: { prev = prev; });
}
```

**Výpis 5.** Ukázka definice CI/CD pipeline a paralelního a sekvenčního vykonávání v `ci.nix`.

Při využití Nix, jako v příkladech, je možné reprodukovatelně spustit proces CI/CD lokálně bez potřeby kontejnerizace a otestovat tak například úspěšnost některých fází. Pokud se navíc použije jiný fyzický popis infrastruktury, je možné otestovat úspěšnost nasazení aplikace, ještě předtím, než bude nasazení provedeno CI/CD serverem na produkční infrastruktuře.

Díky NixOS je možné zajistit stejné prostředí pro testování aplikace a pro následné nasazení aplikace a kompletně tak odstranit problémy způsobené odlišným prostředím. Každé nasazení je aplikace je navíc nedestruktivní operace. Nové i staré verze software, knihoven a konfigurace jsou na cílovém stroji uloženy po nasazení v `/nix/store`. Případný rollback je tak jenom o aktivaci jiné verze aplikace.

Proces CI/CD není závislý na CI/CD řešení a je možné jednoduše použít jiné řešení. Stačí mít dostupný Nix a spustit jeden příkaz. Na druhou stranu, není integrace Nix s klasickými CI/CD řešeními bezproblémová. Je potřeba řešit uložiště `/nix/store`, způsob interpretace výsledků a zpřístupnění výstupů vytvořených pomocí Nix v CI/CD, jako jsou například instalační soubory nebo vygenerovaná dokumentace.

## 5. Závěr

V této práci byly popsány principy balíčkovacího systému Nix a jeho ekosystému a byly představeny silné a slabé stránky těchto technologií. Dále pak bylo navrženo použití Nix v jednotlivých fázích CI/CD při agilním vývoji.

Pomocí Nix je možné se vyvarovat problémům způsobeným odlišným prostředím a vytvořit tak mnohem spolehlivější proces průběžné integrace a nasazení. Výstupem práce je sada příkladů demonstrující použití a možnosti Nix, díky kterým může vývojář jednoduše použít Nix ve svých projektech. Tato sada příkladů je zveřejněna jako open-source na adrese https://github.com/vlktomas/nix-examples.

Zajímavým rozšířením by bylo vytvoření obsáhlé knihovny obsahující nejrůznější metody testování. Dále pak i rozšíření předkládané sady příkladů o další programovací jazyky, respektive sestavovací nástroje.

Použití Nix není jednoduché a je potřeba nastudovat velmi mnoho materiálů. Nicméně výhody, které Nix nabízí, nesporně převažují toto počáteční úsilí. Velká část práce bude potřeba ještě v oficiální podpoře sestavovacích nástrojů, přidávání chybějících balíčků a dokumentaci. Jinak je ale velmi pravděpodobné, že se Nix nebo podobné řešení stane postupem času standardem jak na serverech, tak i na desktopových stanicích.

## Poděkování

Chtěl bych poděkovat panu RNDr. Marku Rychlému, Ph.D. za vedení a cenné rady při vytváření této práce. Dále bych chtěl poděkovat své přítelkyni a rodině za podporu a trpělivost.

## Literatura

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[2] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[3] NixOS organization. Nix package manager guide. https://nixos.org/nix/manual/.

[4] Eelco Dolstra. *The purely functional software deployment model*. PhD thesis, Utrecht University, Utrecht, 2006.

[5] Eelco Dolstra, Andres Löh, and Nicolas Pierron. Nixos: A purely functional linux distribution. *J. Funct. Program.*, 20:577–615, 2010.

[6] Eelco Dolstra. Purely functional configuration management with nix and nixos, Jun 2014. https://infoq.com/articles/configuration-management-with-nix.

[7] NixOS organization. Nixos manual. https://nixos.org/nixos/manual/.

[8] NixOS organization. Nixops user's guide. https://nixos.org/nixops/manual/.