VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ NIX/NIXOPS PRO PRŮBĚŽNOU INTEGRACI A NASAZENÍ SOFTWARE PŘI VÝVOJI

CONTINUOUS INTEGRATION AND DELIVERY BY NIX/NIXOPS IN SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ VLK**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Vlk Tomáš, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Využití Nix/NixOps pro průběžnou integraci a nasazení software při vývoji**
Continuous Integration and Delivery by Nix/NixOps in Software Development
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s přístupem průběžné integrace a nasazení (Continuous Integration and Delivery, CI/CD) při agilním vývoji software a analyzujte potřeby a možnosti podpory CI/CD. Popište realizaci CI/CD dle různých agilních metodik vývoje software.
2. Seznamte se s jazykem a systémem správy software Nix, operačním systémem NixOS (jádro linux) a s projektem NixOps.
3. Navrhněte postup uplatnění Nix a NixOps při CI/CD naplňující potřeby agilního vývoje software. Vytvořte sadu postupů a doporučení pro CI/CD pomocí Nix/NixOps pro různé druhy aplikací (webové, mobilní, vícevrstvé, vysoce distribuované a škálovatelné, atp.).
4. Po konzultaci s vedoucím řešení implementujte v Nix podpůrné nástroje pro uvedené postupy. Implementujte také ukázkové aplikace různých druhů využívající uvedené postupy a nástroje.
5. Výsledek popište, vyhodnoťte a zveřejněte jako open-source.

Literatura:

- Stellman, Andrew. Learning agile (First edition). ISBN 978-1-449-33192-4.
- Dolstra, Eelco, and Andres Löh. "NixOS: A purely functional Linux distribution." ACM Sigplan Notices. Vol. 43. No. 9. ACM, 2008.
- NixOS Manual [https://nixos.org/nixos/manual/]

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 21. října 2019

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, 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.

Abstract

This thesis deals with the application of the functional packaging system Nix and its ecosystem (NixOS, NixOps) for CI/CD in agile development. When using these technologies, the problems caused by different environments are virtually eliminated without the need of containerization. The thesis contains a description of the possibilities and the shortcomings of Nix/NixOps and it proposes a general procedure for the use of these technologies in individual phases of agile development and CI/CD. Thanks to Nix/NixOps, the implementation of CI/CD is very simple and the whole process is also reproducible. The output of the work is a set of the examples demonstrating the use of Nix/NixOps in various projects, which is available as open-source. Thanks to this set, the developers can use Nix quickly and easily in any project, without having to study a large amount of materials.

Klíčová slova

Agilní vývoj, Průběžná integrace, Průběžné nasazení, Správce balíčků, Správa konfigurace, Infrastruktura jako kód, Nix, NixOS, NixOps

Keywords

Agile development, Continuous integration, Continuous deployment, Package manager, Configuration management, Infrastructure as code, Nix, NixOS, NixOps

Citace

VLK, Tomáš. *Využití Nix/NixOps pro průběžnou integraci a nasazení software při vývoji.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

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

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tomáš Vlk
1. června 2020

Poděkování

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

Obsah

Kapitola 1

Úvod

Vývojáři software často pracují v týmech a potřebují nějakou formu organizace způsobu vývoje. Dnes převažují agilní metodiky, pro které je typický rychlý a iterativní vývoj. To znamená velký počet úkonů, které je potřeba provést při každé iteraci jako je analýza kódu, testování, změna infrastruktury u cloudových poskytovatelů nebo nasazení software. Proto je s agilním vývojem úzce spjatá praktika průběžné integrace a nasazení software, při které jsou tyto úkony automatizovány.

Pro spolehlivé otestování a nasazení aplikace je potřeba eliminovat problémy způsobené odlišným prostředím. To je, v dnešní době nepřeberného množství frameworků a nástrojů a jejich konfigurace, velmi složité. Ideálně by měli všichni vývojáři aplikaci vyvíjet ve stejném prostředí a neměl by být problém pro nového vývojáře se co nejrychleji a s minimálním úsilím zapojit do vývoje. Důležitější však je, aby prostředí, které používá vývojář, bylo totožné s prostředím CI 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.

Ekosystém balíčkovacího systému Nix nabízí deklarativní zápis balíčku, celého systému nebo dokonce celé infrastruktury. To vše s jistotou reprodukovatelnosti a bez potřeby kontejnerizace. Díky tomu je proces průběžné integrace a nasazení software velmi spolehlivý a agilní vývoj rychlejší. Není třeba se bát změny, vše se dá zreprodukovat lokálně a případná chyba se dá vždy jednoduše vrátit zpět. Celý koncept, díky kterému má Nix tyto vlastnosti, je velmi inovativní a mění od základu způsoby, jakými se doposud sestavuje a nasazuje software.

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.

Následující kapitola se zabývá problematikou agilního vývoje a analyzuje potřeby a možnosti průběžné integrace a nasazení software. Kapitola 3 představuje správce balíčků Nix a technologie kolem něj. Vysvětluje princip jejich fungování a popisuje jejich vlastnosti, výhody a nevýhody. Návrhem použití Nix pro jednotlivé fáze průběžné integrace a nasazení software se zabývá kapitola 4. V předposlední kapitole 5 je popsána implementace tohoto návrhu a zhodnocení dosažených výsledků této práce. V závěrečné kapitole 6 je shrnutí celé práce a návrhy na rozšíření.

Kapitola 2

Průběžná integrace a nasazení software při vývoji

Dříve se používal především vodopádový model vývoje software. Ten je velmi fixní a to způsobovalo mnoho problémů, zejména nedokončení projektu včas a vyšší náklady na vývoj. V dnešní době se používají hlavně iterační modely a ve většině případů jde o jistou formu agilního vývoje.

Při iterativním vývoji je důležitá průběžná integrace, se kterou může být spojené i průběžné nasazení software. Každý projekt má ale na integraci a nasazení jiné nároky. Pro velké monolitické aplikace je důležité, aby byly jednotlivé součásti velmi dobře otestovány. Při vývoji desktopové aplikace může být důležité sestavování různých variant. U webových aplikací přichází po integraci na řadu nasazení software na produkční server, jenž nahrazuje starší verzi. Průběžné nasazení software se tak využívá nejen při vývoji, ale i při provozu aplikace.

2.1 Agilní vývoj

Synonyma slova *agilní* jsou slova čilý, aktivní, horlivý nebo pružný. Agilní vývoj je odpovědí na nepružný a pomalý vývoj při použití vodopádového modelu. Je to v prvé řadě sada *metod* a *metodik*, které pomáhají týmu pracovat efektivněji a přijímat lepší rozhodnutí. Tyto metody a metodiky se zabývají všemi oblastmi tradičního softwarového inženýrství, včetně řízení projektu, návrhu a architektury software a zlepšování procesů. Tato sekce čerpá z knihy Learning Agile [22] a vysvětluje agilní vývoj a pojmy s ním spojené.

Nedílnou součástí agilního vývoje je komunikace v týmu. Členové týmu musí mít stejný *mindset (stejné nastavení mysli)*. To jim pomáhá sdílet informace s ostatními, takže mohou dělat důležitá projektová rozhodnutí společně namísto manažera, který dělá tato rozhodnutí sám. Agilní mindset je o otevření procesu plánování, návrhu a zlepšování celému týmu. Pokud se použijí samotné praktiky bez toho, aniž by členové týmu mezi sebou komunikovali a měli stejný přístup, nebude agilní vývoj fungovat tak dobře, jak by mohli očekávat.

Pro lepší porozumění agilního mindsetu a cílů metod a metodik byl vytvořen *Manifest*[1] shrnující zásadní hodnoty agilního vývoje:

- *Jednotlivci a interakce* před procesy a nástroji.
- *Fungující software* před vyčerpávající dokumentací.
- *Spolupráce se zákazníkem* před vyjednáváním o smlouvě.
- *Reagování na změny* před dodržováním plánu.

Jednotlivci a interakce před procesy a nástroji. Projekt může směřovat špatným směrem, pokud se lidé slepě řídí nastavenými procesy. Každý člen týmu má své vlastní motivace, nápady a preference. Je proto důležité lidem v týmu porozumět a pochopit, jak spolu pracují a jak práce každého člověka ovlivňuje všechny ostatní. Je mnoho agilních praktik, které podporují jednotlivce a komunikaci v týmu, jako jsou denní *standup meetingy* nebo *retrospektiva* na konci iterace nebo projektu.

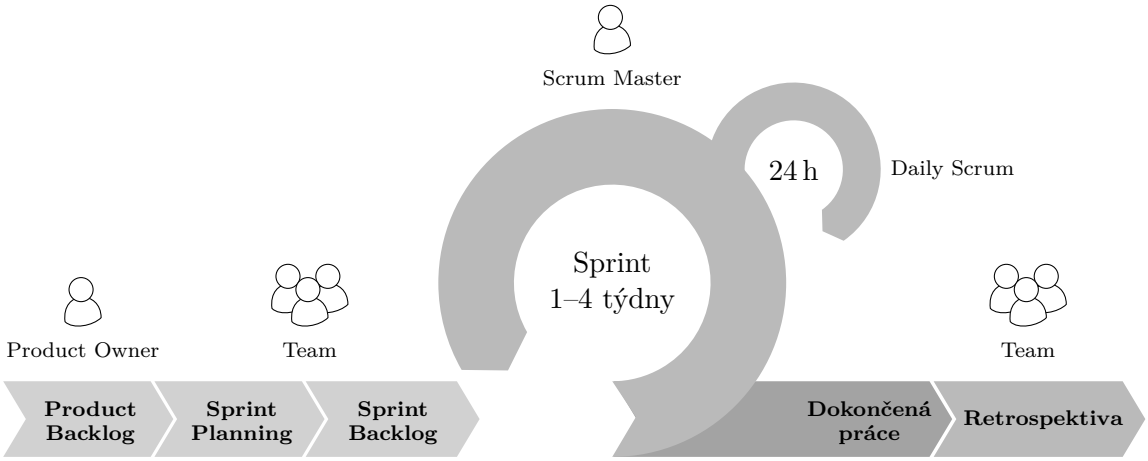
Fungující software před vyčerpávající dokumentací. V agilním vývoji se upřednostňuje fungující software, který znamená pro firmu nějakou hodnotu. Může ho například prodat zákazníkům a pak ho postupně vylepšovat. Neznamená to ale, že software nemusí mít dokumentaci. Je potřeba se zaměřit na dokumentaci, která pomůže pochopit daný problém nebo zlepší komunikaci mezi zákazníkem a vývojářem. Jsou to například různé skici uživatelského rozhraní (wireframe) nebo sekvenční diagramy. Častou dokumentační praktikou při agilním vývoji je psaní jednotkových testů předtím, než se začne vyvíjet neboli *test-driven development (vývoj řízený testy)*.

Spolupráce se zákazníkem před vyjednáváním o smlouvě. Pokud je to možné, je součástí týmu i samotný *zákazník (product owner)*. Nevyvíjí produkt, ale účastní se meetingů, přispívá nápady a co je nejdůležitější, cítí vlastnictví finálního produktu. Zákazník nejčastěji používá *user stories (uživatelské příběhy)* jako způsob spolupráce se zbytkem týmu. Souhrn všech user stories a dalších věcí, které musejí být udělány, se označuje jako *product backlog*.

Reagování na změny před dodržováním plánu. Žádný plán není přesný, a pokud se tým bude pevně držet špatného plánu, může tvořit špatný produkt. Tým musí počítat se změnami v průběhu vývoje. Měl by neustále komunikovat se zákazníkem a případně upravovat plán a reagovat na změny. Díky včasným změnám je možné ušetřit čas později, protože čím více je produkt hotový, tím těžší bude ho změnit. V agilních metodikách se používá často *task board*, kde jsou připnuty jednotlivé úkoly (většinou user stories) ve sloupcích podle jejich statusu. Kdokoliv z týmu (nejenom manažer) pak může úkoly přeuspořádat, nebo přidávat nové, a reagovat tak na změny od zákazníka.

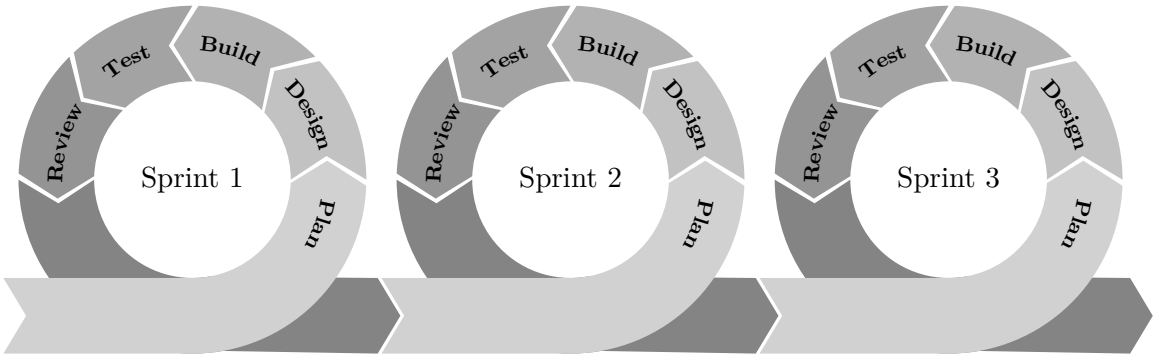
Agilní metodika je soubor praktik kombinovaných s nápady a radami. Vymezuje různé role a odpovědnosti pro členy týmu a doporučuje určité postupy pro každého z nich v různých fázích projektu [22]. Příkladem může být agilní metodika *Scrum*, jejíž proces je zobrazen na obrázku 2.1.

[1] Manifesto for Agile Software Development – https://agilemanifesto.org



Obrázek 2.1: Agilní metodika Scrum. Vývoj probíhá v iteracích, které se označují jako sprinty. Využívají se některé vybrané metody specifické pro Scrum jako je Sprint Backlog nebo Daily Scrum. Členové týmu mají role a odpovědnost za některé fáze. Například Scrum Master má dohled nad Sprintem a Product Owner vytváří Product Backlog.[2]

Vývoj probíhá v *iteracích (sprint)*, které jsou tvořeny jednotlivými fázemi, jak lze vidět na obrázku 2.2. Fáze a délky iterací se mohou lišit podle zvolené metodiky. Například u druhé nejčastěji používané metodiky *XP (Extreme programming)* je obvykle délka iterace jeden týden, kdežto u metodiky Scrum je iterace většinou dlouhá jeden měsíc. Na konci každé iterace je fáze review, ve které by měl být produkt představen zákazníkovi, aby mohl dát zpětnou vazbu, a mohl se tak upravit plán na další iteraci.



Obrázek 2.2: Fáze iterace při agilním vývoji. Každá iterace obsahuje fázi plánování, návrhu, sestavení, testování a review.[3]

Na rozdíl od vodopádového modelu se v agilním vývoji používají iterace a již předem se počítá s tím, že se v průběhu vývoje bude měnit zadání či plán projektu. Součástí agilního vývoje po skončení sprintu nebo projektu je vždy *retrospektiva* neboli zhodnocení efektivnosti použitých praktik a přijetí případných vylepšení pro další projekt.

[2]Obrázek byl vytvořen na základě obrázku v článku *A comprehensive guide on agile methods for modern software development* – https://www.peerbits.com/blog/agile-software-development.html.

[3]Inspirováno obrázkem z článku *Getting Started with Agile? Read this First* – https://www.itsma.com/getting-started-with-agile/.

2.2 CI/CD: průběžná integrace a nasazení software

Jak bylo popsáno v předchozí sekci, každá iterace obsahuje fázi vývoje, testování a dodání zákazníkovi (review). Vývojáři potřebují testovat své změny i několikrát denně. Některé testy mohou provádět na své stanici, jiné ale mohou trvat příliš dlouho na to, aby je vývojář spouštěl lokálně. Po provedení změn může být také potřeba provést statickou analýzu zdrojového kódu. Zároveň i proces dodání zákazníkovi, či nasazení, se může dít každý den nebo po každé iteraci. Je proto vhodné provádět testování a proces nasazení po každé změně zdrojového kódu automaticky.

Průběžná integrace (CI – Continuous Integration) je praktika, kdy členové týmu integrují svou práci do vzdáleného repozitáře velmi často, klidně několikrát denně. Každá integrace je ověřena automatizovaným sestavením (včetně testování), aby byly chyby integrace detekovány co nejrychleji [11]. Průběžná integrace zpravidla probíhá na samotném stroji tzv. CI serveru, na kterém proces průběžné integrace funguje následovně:

1. Nejprve vývojář vloží své změny do repozitáře.
2. CI server periodicky (například každých 5 minut) kontroluje repozitář a pokud detekuje změnu, započne integraci. Případně může proces integrace vyvolat samotná událost vložení kódu do repozitáře a periodická kontrola tak není potřeba.
3. Z repozitáře se stáhne aktuální verze zdrojových kódů a spustí se předem definované kroky integrace.
4. CI server generuje vývojářům zpětnou vazbu. Může to být skrze webové rozhraní, e-mail nebo týmový chat.
5. Pokud funguje CI server na základě periodické kontroly, pokračuje v periodické kontrole repozitáře, jinak je proces integrace dokončen.

Aby mohl proces automatické integrace fungovat, měl by software být přizpůsoben tak, aby sestavení bylo pokud možno otázkou jednoho příkazu [13]. Zdrojové soubory a sestavovací skript by neměly být závislé na prostředí, jako jsou pevně dané cesty k souborům nebo knihovnám. To zjednodušuje jak samotné nasazení procesu CI do vývoje, tak i samotný vývoj. Je to jeden z důvodů, proč by proces integrace měl probíhat na samostatném stroji, který připraví čisté prostředí pro každé sestavení. Díky tomu se totiž ověří, že není proces sestavení na něčem závislý.

CI server po integraci může produkovat výstupy (artifacts). Většinou se jedná o instalační soubor aplikace (release) nebo o automaticky generovanou dokumentaci. Samotné nasazení na cílový server pak funguje manuálně. Toto se označuje jako *průběžné dodávání (Continuous Delivery)*.

Průběžná integrace může být rozšířena o automatické *průběžné nasazení (Continuous Deployment)*. Cílem nasazení software je reprodukovat software z vývojového nebo testovacího prostředí do prostředí, kde má fungovat. Většinou se jedná o přenesení nějakého výstupu generovaného CI serverem na cílový stroj a jeho následnou instalaci. Continuous Delivery a Continuous Deployment mají stejnou zkratku CD a spojením s CI vznikne zkratka *CI/CD*. Rozdíl ve významu jednotlivých pojmů je ale minimální, a proto v této práci není mezi nimi implicitně rozlišováno a pojmem CI/CD je myšleno jak Continuous Delivery, tak i Continuous Deployment.

V souvislosti s pojmem Continuous Deployment je často zmiňován i pojem *DevOps*. Jedná se o přístup k nasazení software, kde vývojáři úzce spolupracují s týmem, který se stará o infrastrukturu a fungování software v produkci. Cílem je přivést agilní přístup do světa systémové administrace a IT operací [15]. Operace prováděné systémovými administrátory se provádí průběžně, stejně jako proces integrace nebo nasazení. Společně s nasazením software se používá například automatické *nasazení infrastruktury*, která je nejčastěji ve formě kódu *(IaC – Infrastructure as Code)* a nástroje pro *správu konfigurace (CMS – Configuration Management Software)*. Po nasazení se navíc software automaticky monitoruje a koordinuje.

2.3 Realizace CI/CD při agilním vývoji

Dodávání software je v agilním vývoji kvůli krátkým iteracím mnohem častější a to přináší nezbytnou režii. Praktika CI/CD usnadňuje používání agilního vývoje software v praxi.

- Automatizace na CI/CD serveru podporuje krátké iterace při vývoji. Vývojář může integrovat své změny do vzdáleného repozitáře i několikrát denně.
- Stav software je neustále pod dohledem. Když někde nastane problém, vývojář se o něm hned dozví. Navíc má od CI/CD serveru zpětnou vazbu celý tým, takže si členové týmu nemusí neustále předávat informaci o provedených změnách, jestli je software otestovaný nebo jestli je nová verze nasazená.
- Díky procesu CI/CD se ověří, že lze projekt sestavit a otestovat jen za pomocí zdrojových kódů a konfigurace CI/CD serveru, a že vývojář například nezapomněl přidat do repozitáře soubor, který má jen on lokálně.
- Testování software probíhá automaticky při každé změně v repozitáři. Vývojář nemusí všechny testy spouštět manuálně a lokálně.
- Součástí procesu integrace může být i kontrola programovacích konvencí (coding conventions).
- Díky jednotkovým a integračním testům podporuje CI/CD refaktorizaci.
- Provedení release aplikace je automatické a výstupy vytvořené CI/CD serverem jsou kdykoliv dostupné.
- Nasazení CI/CD podporuje kolektivní vlastnictví projektu. Díky automatickým testům a kontrole programovacích konvencí může kdokoliv upravit jakoukoliv část. [11]
- Bez průběžné integrace je možné software považovat za rozbitý, dokud ho někdo neotestuje a neprokáže tak, že funguje. Při používání průběžné integrace je software považován jako za funkční (za předpokladu dostatečně obsáhlé sady automatizovaných testů) s každou novou změnou. [15]

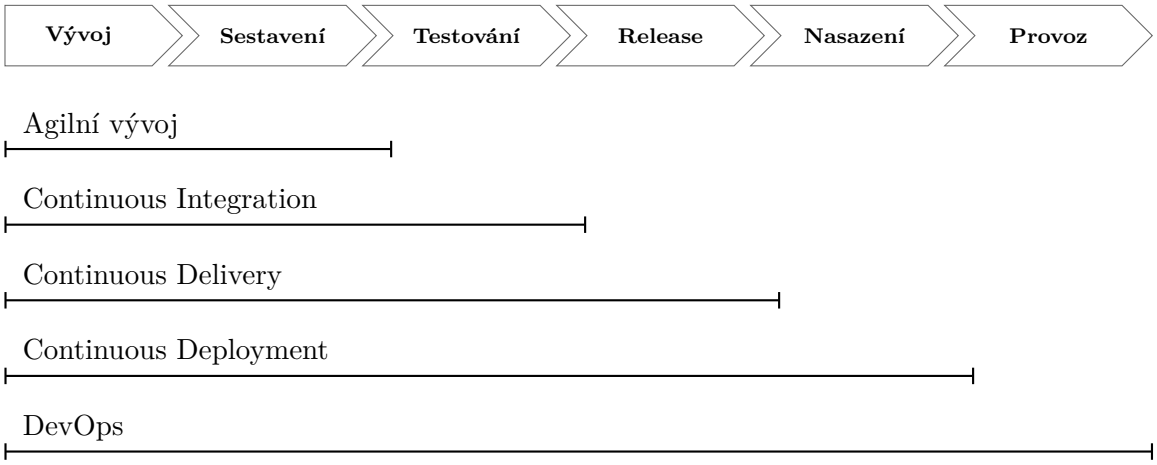
Realizace CI/CD se při jednotlivých agilních metodikách prakticky neliší. Z výhod CI/CD těží jak Scrum, tak XP. Nicméně metodika XP má na proces integrace vyšší nároky. Při XP je vývoj řízený testy a změny se integrují mnohem častěji. Je proto důležité, aby proces integrace na CI/CD serveru byl dokončen do 10 minut [22].

Jakmile CI/CD server detekuje změnu ve zdrojových souborech, prochází při integraci a nasazení několika fázemi [1]. Soubor těchto fází se nazývá *pipeline*. V rámci každé fáze může být vykonáno několik dílčích kroků. Některé kroky nebo i celé fáze ale mohou být vynechány, v závislosti na požadavcích daného software. Například pro aplikace psané v interpretovaných jazycích není potřeba provádět kompilaci. Pro desktopovou aplikaci šířenou jen jako instalační soubor není třeba provádět nasazení na server apod. Obecně se jedná o tyto fáze a dílčí kroky:

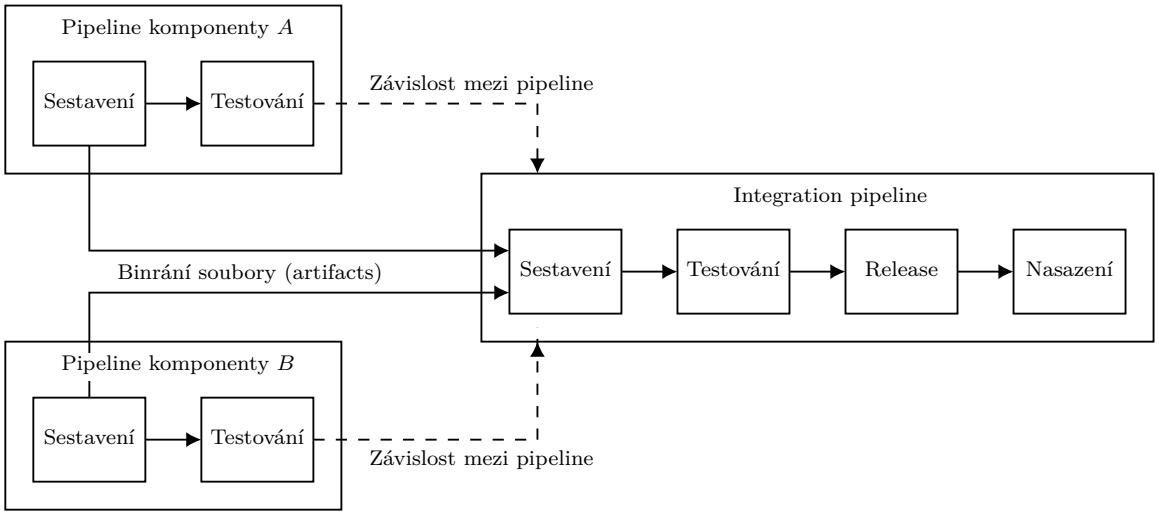
- Vývoj:
  - statická analýza kódu
  - kontrola programovacích konvencí
- Sestavení:
  - příprava prostředí včetně závislostí a nastavení proměnných prostředí
  - jednotkové testy
  - kompilace
- Testování:
  - systémové testy a další dlouhotrvající testy
  - testování na různých platformách
  - dynamická analýza kódu
- Release:
  - vytvoření instalačních souborů pro různé platformy
  - generování programové dokumentace
- Nasazení:
  - nasazení infrastruktury
  - modifikace stavu produkčního systému: prostředí, konfigurace, ...
  - nasazení aplikace na server
  - konfigurace aplikace včetně jakýchkoliv dat nebo stavu, které vyžaduje

Jednotlivé fáze a dříve představené pojmy jsou porovnány na obrázku 2.3. Častou praktikou je, že se některé fáze odloží nebo rozdělí. Například při změně kódu se mohou provést všechny fáze až na fázi release a nasazení. Tyto dvě fáze se pak nastaví tak, že nebudou vykonávány při každé změně, ale například jen jednou denně. Podobně lze rozdělit fázi testování a dlouhotrvající testy provádět později, aby CI/CD systém dával co nejrychlejší zpětnou vazbu. Jedná se o tzv. *two-staged build*. [13]

Pokud se aplikace skládá z více samostatných komponent, může mít každá komponenta svou vlastní pipeline. Komponenty mohou být na sobě závislé a mohou tak nastat složitější scénáře, kdy jednotlivé pipeline jsou závislé mezi sebou, jako je zobrazeno na obrázku 2.4. Pipeline aplikace, která je závislá na pipeline jiných komponent, se nazývá *intergation pipeline*. [15]



Obrázek 2.3: Společné fáze agilního vývoje a CI/CD a porovnání jednotlivých pojmů.[4]



Obrázek 2.4: Příklad pipeline aplikace, která využívá výsledky pipeline komponenty A a komponenty B. Obrázek byl převzat z [15] a upraven.

2.4 Existující systémy pro CI/CD

Proces CI/CD může být hodně náročný a frekvence integrací může být velká. Někdy je také potřeba v rámci CI/CD provést sestavení nebo testování na jiné platformě, než kterou používá CI/CD. Z toho důvodu používají CI/CD systémy agenty nebo virtualizaci, aby mohla být integrace spuštěna i na jiné platformě, než je platforma CI/CD serveru. Některá existující řešení CI/CD je potřeba zprovoznit na vlastním serveru, jiná je zase možné si koupit jako SaaS (Software as a service).

Jenkins. Jedním z nejznámějších open-source systémů pro CI/CD je *Jenkins*[5]. Ten je třeba zprovoznit na vlastním serveru, neboť není oficiálně nabízen k zakoupení jako SaaS.

[4] Obrázek byl převzat a upraven z článku *Continuous Integration, Delivery and Deployment Explained* – https://skillslane.com/continuous-integration-delivery-deployment.

[5] Jenkins – https://jenkins.io

Podporuje agenty na jiných platformách, ale prostředí musí být vždy nastaveno manuálně. Pro definování jednotlivých fází se využívají tzv. kroky. Sada dostupných kroků lze rozšířit pomocí rozšíření nebo se vždy může použít krok pro vykonání shell příkazu. Zápis se provádí skrze webové rozhraní nebo do textového souboru `Jenkinsfile` ve speciálním formátu.

CircleCI. Dalším velmi používaným open-source systémem je *CircleCI*[6]. Stejně jako Jenkins, lze CircleCI zprovoznit na vlastním serveru. Navíc je ale možné zakoupit jej jako SaaS. Podobně jako v Jenkins se také pro jednotlivé fáze definují kroky, které se mají vykonat. Na rozdíl od Jenkins se ale zapisují do souboru `.circleci/config.yml` ve formátu YAML. Proces integrace může být proveden v Docker kontejneru nebo virtualizovaně na jiných platformách.

Travis CI. Velmi populárním CI/CD systémem pro repozitáře hostované u společnosti GitHub je *Travis CI*[7]. I když je Travis CI open-source, je jeho typické použití jen jako SaaS. Definice fází se provádí pomocí souboru `.travis.yml`, kde se namísto kroků definují skripty ke spuštění. Stejně jako v CircleCI, nepoužívá Travis CI agenty a pouze virtualizuje některé platformy.

Gitlab CI/CD. Proces CI/CD je úzce spjatý s repozitářem zdrojových souborů. Služby pro hostování repozitáře tak velmi často nabízejí vlastní řešení CI/CD. Jedním z těchto systémů je *Gitlab CI/CD*[8], který patří pod společnost Gitlab. Služba je nabízena jako SaaS pro repozitáře hostované u společnosti GitLab a stejně jako Travis CI je kompletně open-source. Oproti Travis CI je ale možné provozovat vlastní agenty. Jako konfigurační soubor se používá `.gitlab-ci.yml`, kde se stejně jako ve všech předchozích systémech definují imperativní kroky po každou fázi. Jako prostředí pro sestavení je možné využít Docker kontejner, virtualizaci nebo samotný systém, na kterém je agent spuštěný.

Díky spoustě implementací CI/CD systémů si lze vybrat mezi provozováním na vlastním serveru a SaaS. U všech výše představených systémů se dá provést sestavení v Docker kontejneru nebo případně i na jiné platformě. Díky zápisu fází v textovém souboru, může být proces CI/CD spuštěn jen za pomocí zdrojových souborů bez dalšího nastavování. Každý CI/CD systém má ale svůj vlastní formát zápisu jednotlivých fází. To znamená, že je obtížné jednoduše změnit CI/CD server za provozu.

Může se ale stát, že na CI/CD serveru nebude proces integrace nebo nasazení úspěšný. Vývojář má sice zpětnou vazbu, ale jen velmi těžko dokáže reprodukovat stejné prostředí použité CI/CD serverem. Navíc není snadné spustit lokálně to, co dělá CI/CD server. Každé řešení totiž používá svůj formát popisu jednotlivých fází a vývojář nemá dané CI/CD řešení dostupné lokálně. Například CircleCI a Travis CI přistupují k tomuto problému tak, že umožňují SSH přístup do procesu neúspěšné integrace. Pokud je proces CI/CD prováděn v Docker kontejneru, může si vývojář spustit kontejner lokálně a ladit kroky prováděné CI/CD serverem. Nicméně vývojář nemůže jednoduše předcházet tomuto problému.

Po úspěšném procesu integrace přichází na řadu proces nasazení software do produkce. Opět je ale obtížné zajistit stejné prostředí na produkčním serveru jako na CI/CD serveru, kde byl software testován. Sestavený software může například potřebovat jistou knihovnu.

[6] CircleCI – https://circleci.com
[7] Travis CI – https://travis-ci.org
[8] Gitlab CI/CD – https://docs.gitlab.com/ee/ci/

Není ale záruka toho, že na cílovém stroji tato knihovna je. Navíc knihovna může být vyžadována v jisté verzi nebo variantě. Někdo to musí manuálně ověřit a případně tuto knihovnu na produkčním serveru doinstalovat ve správné verzi. Pokud je knihovna na produkčním serveru ve špatné verzi, nemusí být možné nainstalovat jednoduše jinou verzi, protože její změna může ovlivnit i ostatní aplikace, které jsou na ní závislé. Některé aplikace nemusí být s novou verzí kompatibilní a není tu možnost, jak mít v systému tuto knihovnu zároveň ve dvou verzích.

Dalším problémem spojeným s nasazením je jeho provedení za provozu. Nasazení nové verze software je většinou destruktivní operace, protože se přepisují soubory starší verze nasazované aplikace nebo závislosti při aktualizaci a špatně se pak provádí rollback při chybě. Zároveň může být součástí i nasazení infrastruktury, což celý proces nasazení ještě více komplikuje.

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 eliminovány. Používání kontejnerů ale není ideální. Ve spoustě případů není potřeba aplikace v systému, jenom kvůli nepatrně odlišnému prostředí, 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 [12].

Agilní vývoj společně s průběžnou integrací a nasazením přispívají k rychlému a kvalitnímu vývoji. V rámci každé iterace vývojáři velmi často spouští některé testy lokálně. Pokud je vše v pořádku, vloží své změny do repozitáře. Vývojář dostane od CI/CD serveru zpětnou vazbu o úspěchu či selhání. Pokud došlo k neúspěšnému sestavení, musí tento problém vývojář co nejrychleji odstranit, aby nebrzdil vývoj ostatních a aby nebyl projekt v nefunkčním stavu. Co když ale na vývojářově stanici projekt funguje? To je často způsobeno jiným prostředím na vývojářově stanici a na CI/CD serveru. Vývojář musí těmto selháním předcházet. Problémy s prostředím a nejistota úspěšného sestavení na serveru a následného nasazení způsobuje, že se vývojáři „bojí“ integrovat své změny velmi často. To jde proti myšlenkám agilního vývoje.

Kapitola 3

Nix a jeho ekosystém

V předchozí kapitole byl představen agilní vývoj software a s ním spojený proces CI/CD. Ačkoliv existuje nespočet nástrojů, které proces CI/CD implementují, stále je tu prostor pro zlepšení. Nix přichází s inovativními myšlenkami, které pomáhají mnoho problémů vyřešit.

Tato kapitola se zabývá správcem balíčků Nix a technologiemi, které s ním bezprostředně souvisí. Protože je ale vývoj Nix velmi rychlý a jeho možnosti jsou obrovské, obsahuje tato kapitola hlavně zásadní principy a vlastnosti jednotlivých technologií. Pro úplné pochopení je vždy potřeba číst aktuální verzi manuálu dané technologie ([17], [18], [19], [20], [21]), z kterých tato kapitola čerpá. Některé zásadní principy jsou obsažené v původních článcích [10], [6], [9], [7], [8] k jednotlivým technologiím.

Nix vytvořil Eelco Dolstra okolo roku 2004. Název Nix je odvozen od holandského slova „niks“, znamenající nic [6]. V roce 2005 objevil Hubbleův vesmírný dalekohled dva drobné měsíce Nix a Hydra trpasličí planetky Pluto[1]. Zřejmě proto pak byly další technologie (Hydra a Charon) okolo Nix pojmenovávány podle měsíců Pluta. Výjimkou je operační systém NixOS. Charon byl pak později přejmenován na NixOps.

3.1 Standardní balíčkovací systémy

Standardní správce balíčků je software, který umožňuje instalovat, aktualizovat a odstraňovat software z operačního systému. Balíčky jsou většinou distribuovány z jednoho centrálního repozitáře a uživatel si může přidat do systému další neoficiální repozitáře. Při instalaci balíčku se správce balíčků postará o to, aby byly společně s instalovaným softwarem v systému nainstalovány i všechny jeho závislosti. Při aktualizaci jsou instalované soubory nahrazeny novými soubory a při odstranění balíčku jsou tyto soubory definitivně odstraněny.

Jako standardní balíčkovací systémy je možné označit správce balíčků v nejrozšířenějších linuxových distribucí. Je to *APT* pro distribuce založené na Debianu, *DNF* pro distribuce používající RPM formát balíčků jako je Fedora nebo Redhat Enterprise Linux a *Pacman* používaný hlavně v linuxové distribuci Arch. Všechny tyto balíčkovací systémy jsou *stavové (stateful)*. 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.

[1] Zdroj: https://solarsystem.nasa.gov/moons/pluto-moons/in-depth/

Dalším problémem standardních balíčkovacích systémů je nemožnost mít několik verzí stejného programu/knihovny. Často se tento problém řeší jiným jménem balíčku (například končícím číslem verze) a po instalaci následně výběrem některé verze jako aktivní. V linuxové distribuci Ubuntu k tomu například slouží příkaz `update-alternatives`. Může ale nastat situace, kdy budou dva balíčky závislé na jiné verzi knihovny, přičemž pouze jedna verze knihovny může být v systému aktivní. Takový problém je ve standardních balíčkovacích systémech neřešitelný a nazývá se jako *dependency hell*. Tyto problémy způsobují malou spolehlivost a flexibilitu CI/CD systémů pro nasazení software. 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[2], 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. Dokonce je možné mít v systému více verzí stejného balíčku. 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 bezpečnostní 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. Příkladem je *Arch Build System (ABS)*[3], který může být využit společně s Pacman. Podobně funguje i *Homebrew* využívaný primárně na macOS. V Homebrew se distribuují balíčky primárně v binárním formátu, ale uživatel může zvolit i lokální kompilaci a následnou instalaci[4]. Takový balíčkovací systém lze tedy označit jako *hybridní*.

3.2 Nix: čistě funkcionální správce balíčků

Aby šlo problémy zmíněné v předchozí sekci efektivně řešit, je potřeba změnit přístup, jakým jsou v systému ukládány a referencovány závislosti. V unix-like systémech se spustitelné programy ukládají do standardních cest jako je `/bin` nebo `/usr/bin`. Sdílené knihovny jsou uloženy typicky v `/lib` nebo v `/usr/lib`. Obsah těchto adresářů pak tvoří jakýsi globální stav, který pak balíčkovací systémy modifikují. Problém ale nastává, pokud není přesně specifikována verze v názvu spustitelného programu nebo knihovny. Například OpenSSH v adresáři `/usr/bin/ssh` může být binární soubor nebo symlink (symbolický odkaz) jenom na jednu verzi. Jeden program může vyžadovat SSH ve verzi 8.1 a druhý ve verzi 7.5. To pak může způsobit problémy za běhu nebo i nefunkčnost celého programu. Dokonce se může stát, že i správná verze závislosti nebude fungovat. Nejčastěji je to zapříčiněno jiným způsobem sestavení závislosti (například jiný přepínač kompilátoru), se kterým program nepočítal. Programy nespecifikují explicitně svoje závislosti, jakmile jsou nainstalovány [10]. Využívají standardní cesty a předpokládají, že je dostupná správná verze a varianta.

[2] Reproducible builds – https://reproducible-builds.org
[3] Arch Build System – https://wiki.archlinux.org/index.php/Arch_Build_System
[4] Homebrew FAQ – https://docs.brew.sh/FAQ

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 úložiště programů `/bin` nebo `/usr/bin`. Všechny balíčky jsou instalovány do *Nix store*, což je jednoduše jenom adresář, ve výchozím nastavení `/nix/store`. V úložišti 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, nebo může být prázdný.

Každý objekt je uložen v `/nix/store` pod unikátním identifikátorem, skládajícím se z hashe a názvu samotného objektu. Používá se SHA-256 hash, který je oříznut na 160 bitů a zakódován pomocí notace Base32. 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 [4]. 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. Název objektu je v identifikátoru obsažen jen pro snazší orientaci, neslouží pro identifikaci jako hash.

Na obrázku 3.1 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 `/nix/store/rr3y0c6zyk...-hello-2.10` je závislý na objektu `/nix/store/6yaj6n8l92...-glibc-2.27`, konkrétně na souboru `lib/libc.so.6`. Výpis 3.1 obsahuje zkrácený výpis souboru derivace `/nix/store/808m4zryay...-hello-2.10.drv`, kde jsou vidět reference na jiné objekty. Podobně jako u derivací jsou závislosti textově zapsány i ve výstupech derivací. Konkrétně soubor `/nix/store/rr3y0c6zyk...-hello-2.10/bin/hello` obsahuje řetězec `/nix/store/6yaj6n8l92...-glibc-2.27/lib/libc.so.6`.

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 [3]. 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 [17].

Nix jako každý jiný správce balíčků nabízí příkazy pro instalaci a odinstalaci software. Na rozdíl od klasických správců balíčků ale Nix nemění stav systému a je tedy *bezstavový (stateless)*. Při instalaci nebo aktualizaci se nepřepisují soubory a nemění se závislosti již nainstalovaných balíčků, jako je tomu například u APT nebo RPM. Například pro instalaci balíčku `hello` stačí zadat příkaz:

```
$ nix-env -i hello
```

Obrázek 3.1: Objekty v `/nix/store` a závislosti mezi nimi. Pro lepší přehlednost jsou hashe v názvech objektů zkrácené a některé soubory jsou vynechané.

Poté lze jednoduše balíček aktualizovat:

```
$ nix-env -u hello
```

Není-li balíček pro aktualizaci specifikován, jsou aktualizovány všechny balíčky. Pokud je nějaký jiný balíček závislý na staré verzi balíčku `hello`, tak při jeho aktualizaci se tato závislost nezmění. V systému budou nainstalované dvě verze balíčku `hello` a stará verze bude stále dostupná. Programy závislé na balíčku `hello` budou stále používat starší verzi, dokud i u nich nedojde k aktualizaci. Díky tomu je instalace nebo aktualizace software velmi spolehlivá, protože se nemění globální stav a nemůže dojít k problémům se závislostmi již nainstalovaných balíčků. Pro odinstalaci balíčku `hello` pak slouží příkaz:

```
$ nix-env -e hello
```

Často bývá pro instalaci dostupných více verzí daného balíčku, nebo může být balíček pojmenován jinak. V seznamu všech balíčků se dá proto vyhledávat, a to pomocí regulárního výrazu:

```
$ nix-env -qa firefox-.*
```

Hashe umožňují mít nainstalovaných a provozovat několik verzí a variant programů zároveň. Některá verze ale musí být aktivována, aby byl program dostupný pro uživatele. K tomu slouží v Nix tzv. *profil*, který sdružuje vybrané verze různých programů a zprostředkovává je uživateli. Profil je adresář obsahující symlinky na vybrané objekty v `/nix/store`. Symlinky jsou uspořádány podle standardních cest v unix-like systémech, aby bylo jednoduché integrovat profil do systému. Například obsahuje adresář `bin`, který je pak přidán do proměnné prostředí `PATH` a všechny programy jsou pak dostupné uživateli v terminálu. To je jiný princip než například aktivní verze v APT pomocí `update-alternatives`, protože

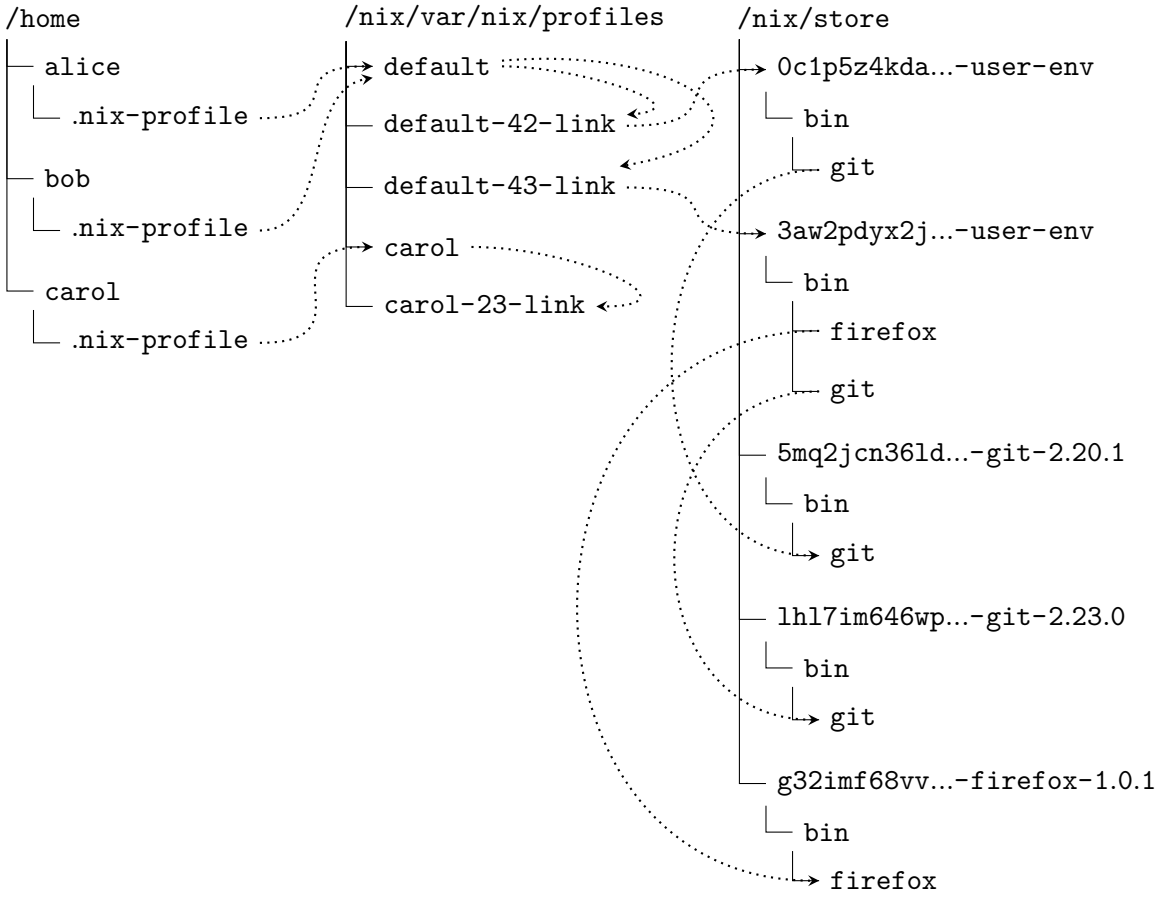
```
{
  "/nix/store/808m4zryay...-hello-2.10.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/rr3y0c6zyk...-hello-2.10"
      }
    },
    "inputSrcs": [
      "/nix/store/9krlzvny65...-default-builder.sh"
    ],
    "inputDrvs": {
      "/nix/store/534ikp0a63...-bash-4.4-p23.drv": [
        "out"
      ],
      "/nix/store/m24h1cnqz5...-hello-2.10.tar.gz.drv": [
        "out"
      ]

    ...
```

Výpis 3.1: Zkrácený výpis derivace pro balíček hello-2.10. Derivace obsahuje název výstupu, sestavovací skript, nastavení proměnných prostředí a všechny závislosti (vstupní soubory a derivace) [7].

profil je primárně jenom pro uživatele a samotné programy využívají stále explicitně závislosti přímo v `/nix/store`. U APT využívají jak uživatel, tak programy vždy jednu aktivní verzi, a proto nemohou dva programy deklarovat různé verze téže závislosti. Všechny balíčky jsou tedy instalovány do jednoho centrálního úložiště, ale nejsou dostupné uživateli. Profil je jakýmsi pohledem do tohoto úložiště a slouží ke zprostředkování nainstalovaných balíčků uživateli. Z pohledu Nix je profil jenom další objekt a je uložen v `/nix/store` stejně jako jakýkoliv jiný balíček.

Po každé instalaci software pomocí Nix je vytvořen nový profil. Stejně tak je i při každé odinstalaci vytvořen nový profil, ale nic, ani odinstalovávaný software, není odstraněno z `/nix/store`. Nově vytvořený profil se stane *aktivním profilem* uživatele. Aktivní profil je jednoduše symlink umístěný mimo `/nix/store` např. `~/.nix-profile` a ukazuje na nějaký profil do `/nix/store`. Díky tomu lze snadno přepínat mezi různými profily. Typickým použitím může být rollback zpět na starší profil po instalaci nebo odinstalaci balíčku. Rollback je možný i po odinstalaci, protože při odinstalaci se nic nemaže z `/nix/store`. Jeden soubor s pevně daným jménem bez hashe také umožňuje jednoduší integraci do systému. Například proměnná prostředí `PATH` může obsahovat jednoduše `~/.nix-profile/bin`. Důležité je také podotknout, že přepnutí na jiný profil, tedy vytvoření nového symlinku `~/.nix-profile/bin`, je v unix-like systémech atomická operace [6]. Díky tomu je použití profilů velmi spolehlivé a instalace, aktualizace nebo odinstalace software je atomická operace. Uživatel má buď starý set programů v `PATH` nebo nový set programů, ale nikdy nemá směs obou.

Příklad použití profilů více uživateli je na obrázku 3.2, kde při přepnutí z profilu `default-42-link` na profil `default-43-link` dojde z pohledu uživatele k aktualizaci balíčku `git` a instalaci balíčku `firefox`.



Obrázek 3.2: Příklad profilů v `/nix/store` a jejich zpřístupnění uživatelům. Tečkovaná šipka vyjadřuje cíl symlinku. Každý uživatel má ve svém domovském adresáři odkaz na aktivní profil, přičemž profily mohou být mezi uživateli sdíleny. Pro snazší přepínání existuje adresář `/nix/var/nix/profiles`, který slouží jako číselník dostupných profilů v `/nix/store` společně s odkazem na aktivní profil. Například profil `default` může ukazovat na profil `default-42-link` nebo `default-43-link`. Obrázek byl převzat z manuálu k Nix [17] a upraven.

Díky profilům neexistuje v Nix něco jako globálně nebo systémově nainstalovaný software pro všechny uživatele. Administrátor může předdefinovat profil pro nové uživatele, ale každý uživatel má své vlastní profily. Instalace software vytvoří nový profil jen pro aktuálního uživatele, takže každý uživatel může instalovat programy a neovlivňuje tak ostatní uživatele. Symlinky v profilu ukazují vždy na jednu verzi a variantu software, která je přesně určena pomocí hashe. Tato verze programu bude vždy stejná. Pokud by nějaký uživatel například chtěl nainstalovat upravenou verzi a podvrhnout ji ostatním uživatelům v systému, nepovede se mu to, protože tato upravená verze bude mít jiný hash. Není ale problém omezit instalaci a odinstalaci software jen pro administrátora a uživatelům jen přiřazovat různé profily.

Podobnou funkci jako profily nabízí i GNU Stow[5], který lze používat nezávisle na správci balíčků systému. Stow slouží pro hromadné vytváření symlinků a může být stejně jako profil použit také pro přepínaní různých verzí instalovaného software. Chybí ale integrace se správci balíčků pro snadné přepínání mezi různými kolekcemi balíčků při instalaci či odinstalaci. Stow je primárně určen pro software instalovaný přímo ze zdrojových souborů nebo pro spravování konfiguračních souborů v domovském adresáři.

Jak již bylo zmíněno, při odinstalaci, či jiné manipulaci s balíčky, se z `/nix/store` nic neodstraňuje. To vede k hromadění nepotřebných objektů v `/nix/store` a velkému zaplnění disku. Díky referencím v objektech ale lze jednoznačně určit, co je ještě potřeba a co už ne. Je tedy možné spustit *garbage collector* a odstranit nepotřebné objekty. Vyhledání nepotřebných objektů začíná v tzv. *GC root*, což je kořen stromu závislostí. Těchto kořenů může být v Nix zaregistrováno více. Garbage collector prochází pro každý GC root celým stromem závislostí a vytvoří si tak seznam všech dosažitelných objektů v `/nix/store`. To, čeho se nedosáhlo při procházení, může být z `/nix/store` bezpečně odstraněno. GC root je jednoduše jenom symlink na nějaký objekt v `/nix/store` obsahující reference na další objekty. Například každý profil je automaticky zaregistrován jako GC root, ale GC root může být definován i manuálně. Pro spuštění garbage collectoru slouží příkaz:

```
$ nix-collect-garbage
```

Pokud chce uživatel odstranit software ze systému, provede jeho odinstalaci. Protože je ale každý profil GC root, tak po spuštění garbage collectoru nebude software odstraněn kvůli starému profilu zanechanému kvůli rollbacku. Pokud chce uživatel definitivně odstranit software i staré profily spustí příkaz:

```
$ nix-collect-garbage --delete-old
```

Přesně a kompletně specifikované závislosti jsou velkou výhodou Nix. Díky tomu lze bezpečně a spolehlivě provést nasazení objektu na jiné zařízení. Stačí společně s objektem nasadit i všechny jeho závislosti (i nepřímé). Každý objekt je adresář nebo soubor v adresáři `/nix/store`. Pokud je objekt adresář, může obsahovat další soubory nebo adresáře. Závislost mezi objekty je vyjádřena tak, že některý ze souborů objektu obsahuje název jiného objektu. Pro objekty v `/nix/store` tak lze přesně určit relaci „závisí na“, která vyjadřuje závislosti mezi objekty. Označme tuto relaci jako R_z a množinu objektů v `/nix/store` jako N. Dále nechť $files(x)$ je množina všech souborů (rekurzivně ve všech podadresářích) objektu x a $contains(f, x)$ je predikát, který je pravdivý, pokud soubor f obsahuje název objektu x. Pak relace R_z je definována jako:

$$R_z = \{(x, y) \,|\, x, y \in N \wedge \exists f \in files(x) : contains(f, y)\} \tag{3.1}$$

Tranzitivní uzávěr R_z^* relace R_z vyjadřuje i nepřímé závislosti mezi objekty. Množina $R_z(x)$ specifikuje přímé závislosti a množina $R_z^*(x)$ pak specifikuje všechny závislosti objektu x:

$$\begin{aligned} R_z(x) &= \{b \,|\, (a, b) \in R_z \cap \{x\} \times N\} \\ R_z^*(x) &= \{b \,|\, (a, b) \in R_z^* \cap \{x\} \times N\} \end{aligned} \tag{3.2}$$

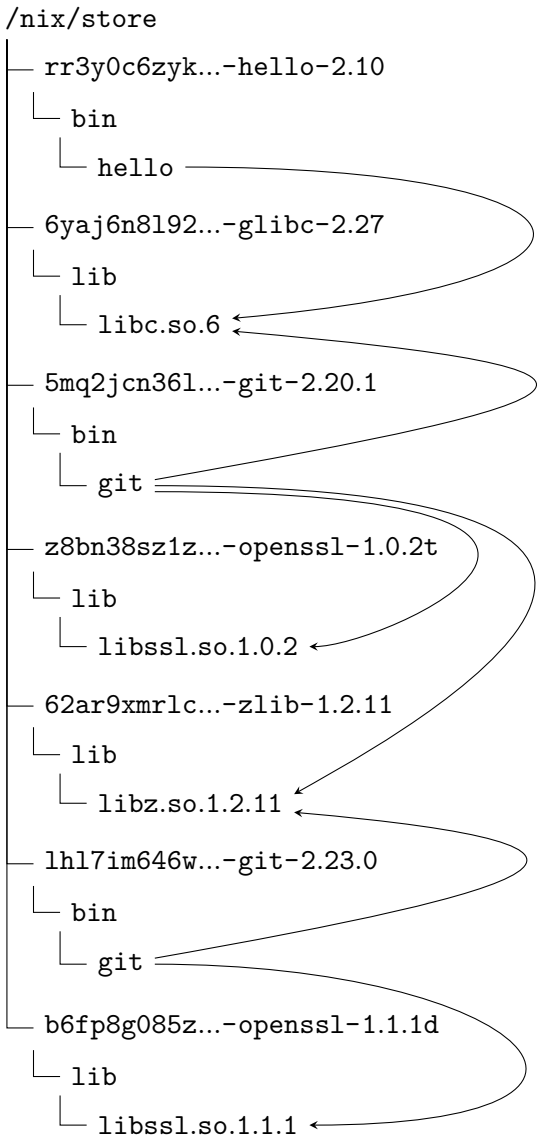
Tato množina společně s objektem x se nazývá *closure* objektu x, nebo též $closure(x)$. Jedná se tedy o objekt samotný společně se všemi jeho přímými i nepřímými závislostmi.

[5]GNU Stow – https://www.gnu.org/software/stow/

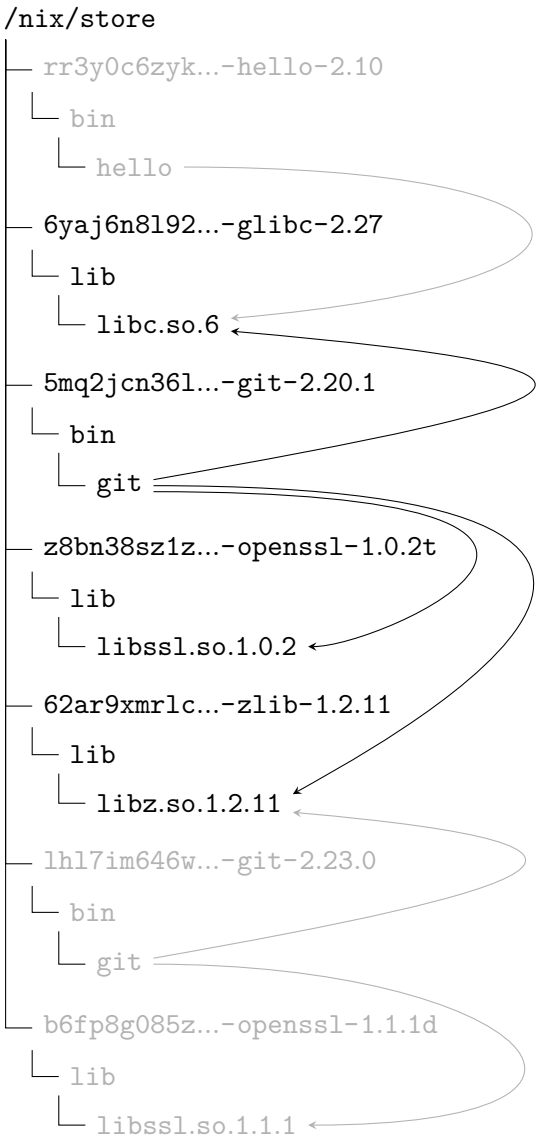
Rekurzivně se dá closure definovat i jako ([3]):

$$closure(x) = \{x\} \bigcup_{y \in R_z(x)} closure(y) \tag{3.3}$$

Sjednocením closure dvou objektů vznikne množina jejich závislostí, ale jejich společné závislosti budou obsaženy v této množině pouze jednou. Výše zmíněný garbage collector při procházení všech dosažitelných závislostí tak vlastně jenom dělá sjednocení closure všech GC root. Problém nastává, pokud má objekt x závislost y a objekt y je závislý na x, protože v takovém případě dojde k zacyklení. To se ale dá snadno detekovat a nemělo by k němu docházet. Příklad closure objektu v `/nix/store` je na obrázku 3.3b.



(a) Obsah Nix store.



(b) Příklad closure v Nix store pro balíček `5mq2jcn36ld...-git-2.20.1`.

Obrázek 3.3: Obsah Nix store a příklad closure. Obrázek byl převzat z disertační práce k Nix a upraven ([3]).

Pokud se nebere v potaz stav (konfigurace, databáze, ...), je closure plně soběstačný balíček. Vezmou-li se všechny soubory patřící do closure a dají se na jiný stroj (stejné platformy), bude program zcela určitě fungovat stejně jako na zdrojovém stroji. Closure obsahuje všechny svoje závislosti podobně jako Flatpak nebo Snappy. Na rozdíl od nich ale closure sdílí své závislosti s jinými closure v `/nix/store` a to díky jednoznačně určeným závislostem. Closure je tedy mnohem efektivnější, co se diskového místa týče.

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 potřeby 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. V této práci bude termínem Nix označován balíčkovací systém, pokud tím bude přímo myšlený jazyk Nix, tak to bude explicitně zmíněno. Zadefinování balíčku pro Nix ve speciálním jazyku je pro většinu software již hotové a uživatel pak už jenom instaluje balíčky jako v jakémkoliv jiném správci balíčků. Pokud ale některý balíček chybí, je potřeba ho vytvořit. Vytvářením balíčku pro Nix a kolekcí balíčků udržovanou komunitou se zabývají následující podsekce.

Doménově specifický jazyk Nix

Tato podsekce slouží pro stručné uvedení do jazyka Nix, pro účely pochopení výpisů kódu v následujícím textu. Podsekce shrnuje jeho vlastnosti a představuje nejčastější konstrukty. Detailnější popis lze nalézt v oficiální dokumentaci [17].

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 potřeba snadno tvořit varianty balíčků a vytvářet grafy závislostí mezi derivacemi. Podobně jako v jiných funkcionálních jazycích, jsou proměnné neměnné (immutable) a nelze zapsat statements (imperativní tvrzení), ale pouze expressions (plně definované výrazy). To například znamená, že nelze použít podmíněný příkaz `if` bez `else` větve.

Jedním z nejdůležitějších konstruktů jazyka je *množina atributů (attribute set)*, což je jednoduše výčet dvojic klíč-hodnota, kde hodnota je nějaký výraz. Ve výpisu 3.2a je příklad množiny atributů společně s přehledem dalších datových typů. Jak lze vidět na řádku 4, k atributům lze přistupovat hierarchicky přes symbol `.` (tečka). Protože je před celou množinou atributů ve výpisu 3.2a klíčové slovo `rec`, je celá množina rekurzivní. To znamená, že každý atribut může být vyhodnocen pomocí jiného. Například na řádku 4 lze využít již definovaného atributu `d`. Naproti tomu ve vnořené nerekurzivní množině atributů na řádku 5 nelze využívat atributy mezi sebou. Je také možné vložit proměnnou (atribut)

přímo do řetězce pomocí zápisu `${}`. Příklad je na řádku 12, kde je obsah atributu `e` vložen do víceřádkového řetězce `f`. Na řádku 3 je pak vidět datový typ *seznam* obsahující hodnoty atributů `a` a `c`. V jazyku Nix je seznam heterogenní datová struktura.

```
1  rec {
2    a = 3.14;
3    b = [ a c ];
4    c = 2 * d.foo;
5    d = {
6      foo = 42;
7      bar = a;
8    };
9    e = "hello";
10   f = ''
11     message:
12     ${e} world
13   '';
14   g = null;
15   h = true;
16   i = /bin/sh;
17 }
```

(a) Přehled datových typů.

```
1  let
2    pow = a: a*a;
3    add = x: y: x+y;
4    powAS = {a}: a*a;
5    addAS = {x, y}: x+y;
6    x = builtins.div 3 0;
7  in
8    rec {
9      a = pow 2;
10     b = add 42 1;
11     c = powAS {a = 2;}
12     d = addAS
13       {y = 1; x = 42;};
14     e = add 1;
15     f = e 1;
16     g = add (pow 2) 3;
17   }
```

(b) Různé způsoby definice a aplikace funkce.

```
1  # soubor answer.nix
2  if 1+1 == 2 then
3    42
4  else
5    0
6
7  # soubor a.nix
8  {
9    foo = ./answer.nix;
10 }
11
12 # soubor bar.nix
13 {a ? import ./a.nix}:
14 with a;
15 {
16   inherit foo;
17 }
```

(c) Import Nix výrazu z jiného souboru.

Výpis 3.2: Přehled konstruktů v jazyku Nix.

Důležitým prvkem funkcionálního jazyka jsou funkce. Ve výpisu 3.2b je přehled různých způsobů definice a aplikace funkce. Aplikace funkce se provádí symbolem mezery. Definice funkce je ve tvaru `argument: tělo funkce`. V jazyku Nix se rozlišuje mezi dvěma způsoby zápisu parametrů funkce. Za prvé je to zápis pomocí pozičních argumentů, které jsou použity na řádku 2 a 3. Za druhé to jsou parametry zapsané pomocí deklarace atributů, jejichž příklad je na řádků 4 a 5. Pokud se definují parametry pozičně, jde vlastně o vnořené funkce s jedním parametrem. Díky tomu lze využít *částečnou aplikaci*, jak je vidět na řádku 14 a 15. V případě deklarace parametrů pomocí atributů, není možné využít částečnou aplikaci, ale parametry mohou být předány funkci v jakémkoliv pořadí, protože se funkci při aplikaci předává množina atributů. Příklad takovéto aplikace je na řádku 11, 12 a 13. Pokud se použije částečná aplikace jako na řádku 14, uloží se do atributu funkce. Definice funkce je tedy také datový typ, konkrétně se tento datový typ nazývá *lambda*. Všechny funkce jsou tedy anonymní a jejich „pojmenování“ je možné jenom díky přiřazení do atributu. Jazyk Nix dále nabízí několik vestavěných funkcí, ke kterým lze přistoupit přes klíčové slovo `builtins`.

Jedním z datových typů je i ne úplně typický datový typ *path*. Je to cesta k nějakému souboru a tento datový typ je rozpoznáván automaticky podle zápisu. Proto je ve výpisu 3.2c potřeba uvést v atributu `foo` aktuální adresář symbolem `./`, neboť samotným uvedením `answer.nix` by nebylo jednoznačné, zda-li se jedná o soubor `answer.nix`, nebo se jedná o množinu atributů `answer` a výběr atributu `nix`. Pomocí klíčového slova `import` se pak dají vyhodnocovat soubory obsahující nějaký výraz (funkce, číslo, . . .). Soubor `bar.nix` obsahuje definici funkce s jedním parametrem, který má výchozí hodnotu. Tato výchozí hodnota je specifikována výrazem za symbolem `?` (otazník). Atribut `foo` v souboru `a.nix` je datového

typu path a obsahuje cestu souboru `answer.nix`. Zatímco parametr `a` v souboru `bar.nix` obsahuje vyhodnocený výraz ze souboru `a.nix`. V tomto případě bude tedy parametr `a` obsahovat množinu atributů s jedním atributem `foo`, který obsahuje nevyhodnocenou cestu k souboru `answer.nix`.

Ve výpisu 3.2b jsou funkce definované pomocí proměnných, které jsou později využívány v rekurzivní množině atributů. K definici lokálních proměnných slouží konstrukt `let ... in`. Proměnné v bloku `let ... in` mohou být definované i rekurzivně a jsou pak dostupné ve jmenném prostoru výrazu za klíčovým slovem `in`. Na řádku 6 je definice proměnné, která obsahuje dělení nulou. Proměnná `x` ale není potřeba, a proto díky lazy evaluation vůbec nedojde k jejímu vyhodnocení. Do jmenného prostoru nějakého výrazu je možné zpřístupnit atributy z množiny atributů také pomocí klíčového `with`. Například ve výpisu 3.2c je na řádku 16 možné využít atribut `foo` z množiny atributů `a` ze souboru `a.nix`.

Klíčové slovo `inherit`, použité na řádku 16 ve výpisu 3.2c, slouží jako syntaktický cukr pro definování atributů pomocí proměnné stejného jména. V tomto případě se jedná o zkrácený zápis `foo = foo`. Za klíčovým slovem `inherit` může následovat více atributů a lze i předřadit všem atributům nějakou zdrojovou množinu atributům. Například použití:

```
inherit x y z;
inherit (src-set) a b c;
```

je ekvivalentní [17] k zápisu:

```
x = x; y = y; z = z;
a = src-set.a; b = src-set.b; c = src-set.c;
```

Funkce mohou mít velký počet parametrů. Pokud je potřeba aplikovat funkci několikrát pouze s jedním odlišným parametrem, je potřeba znovu předat i všechny ostatní parametry. Pro lepší udržovatelnost kódu se v Nix často využívá možnosti „přepisu“ atributů již aplikované funkce. Toho lze dosáhnout tak, že funkce bude vracet množinu atributů, kde jedním z těchto atributů bude funkce nabízející možnost změnit nějaký parametr. Výsledek změny by pak navíc mělo být možné rekurzivně změnit znovu. Implementace může vypadat jako ve výpisu 3.3. Operátor `e1 // e2` slouží ke sjednocení dvou množin atributů `e1` a `e2`, kde atributy se stejným názvem mají výslednou hodnotu z množiny `e2`. Zadefinovat funkci `makeOverridable` rekurzivně pomocí sebe samé je možné díky uložení funkce v rekurzivní množině atributů (řádek 1).

```
1 rec {
2   makeOverridable = f: origArgs:
3     let
4       origRes = f origArgs;
5     in
6       origRes // {
7         override = newArgs: makeOverridable f (origArgs // newArgs);
8       };
9 }
```

Výpis 3.3: Příklad implementace „přepsatelné“ funkce. Kód byl převzat z Nix Pills [2].

Použití pak vypadá následovně (symbol → značí výsledek vyhodnocení předcházejícího výrazu):

```
addXYZ = {x, y, z}: { result = x + y + z; }
x = makeOverridable addXYZ { x = 3; y = 5; z = 7; }
→ { result = 15; override = <<lambda>>; }

y = x.override { y = 10; }
→ { result = 20; override = <<lambda>>; }
```

Podobně lze tento koncept dále rozšířit na pozdní změnu atributů nějaké množiny atributů. Namísto atributu `override` se pak používá atribut `overrideAttrs`. Jenže původní atributy mohou být využívány v jiných výrazech a někdy je potřeba, aby se změna projevila i v nich. V následujícím příkladě se jedná o atribut `z`:

```
set = rec { x = "foo"; y = "bar"; z = x + y; }
→ { x = "foo"; y = "bar"; z = "foobar" }

set.overrideAttrs (oldAttrs: { x = "hello"; y = "world"; })
→ { x = "hello"; y = "world"; z = "foobar"; }

set // { x = "hello"; y = "world"}
→ { x = "hello"; y = "world"; z = "foobar"; }
```

Ke změně atributů tak, aby se změna projevila i tam, kde jsou používány, lze využít *operátor pevného bodu*. Ten umožňuje díky lazy evaluation použít tolik rekurzí, kolik je potřeba, aby se výraz vyhodnotil. V Nix se používá tato definice operátoru pevného bodu[6]:

```
fix = f: let result = f result; in result
```

Množina atributů se poté definuje jako funkce namísto rekurzivní množiny a pro její vyhodnocení se použije operátor pevného bodu. Aby bylo možné navíc změnit nějaké atributy, musí se funkce pro výpočet pevného bodu nepatrně upravit. V následujícím příkladě má definice množiny `newset` tvar operátoru pevného bodu s možností „přepisu“ atributů.

```
set = self: { x = "foo"; y = "bar"; z = self.x + self.y; }
fix set
→ { x = "foo"; y = "bar"; z = "foobar"; }

overrides = { x = "hello"; y = "world"; }

let newset = set (newset // overrides); in newset
→ { x = "foo"; y = "bar"; z = "helloworld"; }

let newset = set (newset // overrides); in newset // overrides
→ { x = "hello"; y = "world"; z = "helloworld"; }
```

Nová množina atributů `newset` tak odpovídá původní množině atributů `set`, ve které byly přepsány atributy `x` a `y` a to se zároveň projevilo ve vyhodnocení atributu `z`. Příklad byl převzat z Nix Pills [2].

[6] `fixed-point.nix` – https://github.com/NixOS/nixpkgs/blob/master/lib/fixed-points.nix

Derivace a její výstup

Nix je doménově specifický jazyk určený pro deklarativní specifikaci všech aspektů balíčku jako je získání zdrojových souborů, sestavení ze zdrojových souborů, definování závislostí a definování metadat balíčku. Balíčky se definují pomocí derivací a jsou sestaveny vykonáním derivace. Při sestavování balíčku se nejdříve rekurzivně sestaví všechny závislosti, a nakonec se spustí sestavovací skript s nastavenými proměnnými prostředí podle definice v derivaci.

Uživatel (vývojář) nikdy nevytváří derivace manuálně, k tomu slouží jazyk Nix. Derivace se vytváří pomocí klíčového slova (vestavěné funkce) `derivation` a je to jediný konstrukt jazyka, který produkuje nějaký výstup do `/nix/store`. Z pohledu jazyka je derivace jenom množina atributů. Pokud Nix výraz obsahuje derivaci, tak se po vyhodnocení jako mezikrok nejdříve převede na `.drv` soubor a potom se teprve vykoná a vznikne výstup derivace. V souboru `.drv` je derivace zapsaná ve speciálním formátu, který lze převést na formát JSON (příklad byl ve výpisu 3.1). K sestavování derivace se používá `.drv` soubor namísto Nix výrazu z toho důvodu, že stejnou derivaci je možné zapsat více různými Nix výrazy, ale derivace s totožným nastavením převedená na `.drv` soubor bude vždy stejná. To umožňuje jednoznačně identifikovat objekty (derivace) v `/nix/store` nezávisle na formátu zápisu pomocí jazyka Nix. Navíc je možné v příštích verzích Nix modifikovat jazyk Nix, a přitom zachovat stejné rozhraní pro sestavování derivací. [6]

Nejjednodušší příklad vytvoření derivace je ve výpisu 3.4. Proměnná `$out` je speciální proměnná, kterou automaticky Nix nastaví sestavovacímu skriptu, obsahující cestu výstupu derivace. Takováto derivace může být sestavena pomocí příkazu `nix-build`.

```
1  derivation {
2    name = "hello";
3    builder = "/bin/sh";
4    args = [
5      "-c"
6      "echo hello! > $out"
7    ];
8    system = "x86_64-linux";
9  }
```

Výpis 3.4: Příklad jednoduché derivace produkující soubor obsahující text: hello!. Převzato z prezentace od Grahama Christensena[7].

Když se vyhodnocuje derivace, tak se rekurzivně vyhodnocují i všechny vstupní derivace a kopírují se zdrojové soubory do `/nix/store`. V tomto případě derivace nemá žádné zdrojové soubory. Dále vznikne soubor `/nix/store/rj25an6k5n...-hello.drv`, jehož obsah je ve výpisu 3.5. Tento příklad derivace neobsahuje odkazy na jiné derivace ani na zdrojové soubory. Obsahuje ale název svého výstupu `/nix/store/krazqsvqny...-hello`.

[7] NixCon 2019: *Nix: How and Why it Works* – https://www.youtube.com/watch?v=lxtHH838yko

```
{
  "/nix/store/rj25an6k5n...-hello.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/krazqsvqny...-hello"
      }
    },
    "inputSrcs": [],
    "inputDrvs": {},
    "platform": "x86_64-linux",
    "builder": "/bin/sh",
    "args": [
      "-c",
      "echo hello! > $out"
    ],
    "env": {
      "builder": "/bin/sh",
      "name": "hello",
      "out": "/nix/store/krazqsvqny...-hello",
      "system": "x86_64-linux"
    }
  }
}
```

Výpis 3.5: Vytvořená derivace `/nix/store/rj25an6k5n...-hello.drv` z kódu ve výpisu 3.4.

Po sestavení derivace budou v `/nix/store` dva nové soubory: derivace `rj25an6k5n...-hello.drv` a výstup derivace `krazqsvqny...-hello`. Soubor `krazqsvqny...-hello` obsahuje jenom řetězec `hello!`. Sestavení derivací je možné přirovnat ke kompilaci zdrojových souborů jazyka C [2]:

- soubor `.nix` odpovídá zdrojovému souboru `.c`
- soubor `.drv` odpovídá objektovému souboru `.o`
- výstup derivace (složka nebo soubor) je analogický k binárnímu souboru

Pokud se manuálně spouští příkaz `nix-build`, je vytvořen v pracovním adresáři soubor `result`, který je symlinkem na výstup derivace v `/nix/store`. Tento symlink je automaticky zaregistrován i jako GC root. Pokud je soubor `result` odstraněn, je zaregistrovaný GC root (symlink) nefunkční. To garbage collector automaticky rozpozná a tento GC root odstraní před samotným procházením stromu závislostí.

Získání hashe pro identifikaci objektu v `/nix/store` je trochu komplikovanější, než bylo uvedeno na začátku sekce 3.2. Pokud se počítá hash ze zdrojového souboru nebo souboru derivace, tak se k hashi obsahu souboru přidá několik metadat jako je použitý hashovací algoritmus a typ objektu. Z výsledného řetězce (hash a metadata) se poté znovu udělá hash a ten se ořízne a zakóduje pomocí Base32 notace. Pokud se hash počítá z více souborů, tak se musí soubory nejdříve zabalit do *NAR (Nix Archive)*. Nestačí například klasický TAR archiv, protože ten nemusí být vždy reprodukovatelný (časové značky, pořadí souborů, ...).

U výstupu derivace je prvotní hash získán jako hash z „pracovní verze“ derivace. V právě zpracovávané derivaci totiž ještě není známa výstupní cesta (protože se teprve počítá). V pracovní verzi derivace je tak prázdná výstupní cesta, a navíc jsou všechny vstupní derivace nahrazeny hashem výstupu stejným způsobem. Díky tomu se při jakékoliv změně závislosti (i nepřímé) změní i hash výstupu derivace. Detailní popis počítání hashe je uveden v Nix Pills [2].

Předchozí příklad derivace je velmi nízkoúrovňový a při definici balíčku se nepoužívá. Raději se používají pomocné funkce, které zapouzdřují časté úkony a usnadňují a zpřehledňují tak zápis balíčků. Příklad definice balíčku pro GNU Hello[8] je ve výpisu 3.6. Tento příklad neobsahuje popis sestavení balíčku. Nix obsahuje výchozí sestavovací skript napsaný v jazyku Bash, který předpokládá sestavení pomocí nástroje Make. GNU Hello slouží jako referenční program pro Autotools[9] a z toho důvodu je výchozí sestavovací skript vytvořen tak, aby korespondoval se standardním procesem sestavení využitý v tomto referenčním balíčku jako je:

```
$ tar xf hello-2.10.tar.gz
$ ./configure
$ make
$ sudo make install
```

Funkce `stdenv.mkDerivation` na řádku 3 ve výpisu 3.6 je pomocná funkce, která obaluje vestavěnou funkci `derivation` a standardní sestavovací skript. Atribut `stdenv` je jednoduše také derivace, která definuje závislost na základních nástrojích (make, gcc, coreutils, bash, ...) a standardní knihovně jazyka C (glibc). Z toho důvodu nemusí balíček GNU Hello definovaný pomocí Nix obsahovat ani definici závislostí. Odpovídající derivace byla již prezentována ve výpisu 3.1.

Jak lze vidět na výpisu 3.6, je celý balíček 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.

Na řádku 7 obsahuje atribut `src` cestu ke zdrojovým souborům. V tomto příkladě se používá funkce `fetchurl`, která stáhne zdrojové soubory v podobě archivu z internetu. Aby byla funkce balíčku čistá a výraz `src` referenčně transparentní, musí být dodán i hash, který slouží pro ověření integrity staženého souboru. Tento hash podmiňuje, že obsah staženého souboru bude vždy stejný, a že vyhodnocení atributu `src` bude také vždy stejné.

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

[8] GNU Hello – https://www.gnu.org/software/hello/
[9] GNU Autotools – https://www.gnu.org/software/automake/

```
{ stdenv, fetchurl }:

stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.10";

  src = fetchurl {
    url = "mirror://gnu/hello/${pname}-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
  };

  doCheck = true;

  meta = with stdenv.lib; {
    description = "A program that produces a~familiar greeting";
    longDescription = ''
      GNU Hello is a~program that prints "Hello, world!".
      It is fully customizable.
    '';
    homepage = https://www.gnu.org/software/hello/manual/;
    license = licenses.gpl3Plus;
    maintainers = [ maintainers.eelco ];
    platforms = platforms.all;
  };
}
```

Výpis 3.6: Jednoduchý příklad definice balíčku. Kód byl převzat z definice balíčku GNU Hello v repozitáři balíčků Nixpkgs (tento repozitář bude popsán v následujících podsekcích).

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. [17]

Na Linuxu je sandbox ve výchozím nastavení zapnutý a na ostatních platformách vypnutý [17]. Použití sandboxu při sestavování se dá vypnout ad-hoc. I když je ale sandbox zapnutý, nemusí být sestavení aplikace reprodukovatelné. Sandbox zaručuje co nejlepší podmínky pro reprodukovatelnost, ale samotný sestavovací skript musí být vytvořen tak, aby byl vždy deterministický. Neměl by například při sestavení používat aktuální čas, nebo náhodná čísla.

Díky použití čistých funkcí, sandboxu a deterministickému sestavovacímu skriptu je sestavení derivace deterministické a reprodukovatelné. Stačí derivaci sestavit pouze jednou a pro každé další spuštění sestavení derivace se vrátí již dostupný výsledek. Navíc mohou být závislosti derivace sestavovány souběžně. Derivace jsou ve výchozím nastavení sestavovány sekvenčně, ale může být jednoduše nastaveno, aby se derivace sestavovaly paralelně:

```
$ nix-build -A hello --max-jobs auto
```

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 3.6 je FOD.

Proces sestavení pomocí standardního sestavovacího skriptu prochází několika fázemi[10]. Fáze jsou zobrazeny na obrázku 3.4. Všechny atributy derivace jsou v sestavovacím skriptu dostupné jako proměnné prostředí. Úprava standardního procesu sestavení tak spočívá v předefinovaní celé fáze pomocí atributu derivace — jak je ukázáno ve výpisu 3.7a — nebo se jenom přidají nějaké příkazy před, respektive po dané fázi. V druhém případě se definují atributy jako `preInstallPhase`, respektive `postInstallPhase`.

Obrázek 3.4: Fáze standardního sestavovacího skriptu. Fáze jsou vykonávány sekvenčně počínaje fází Unpack a konče fází Dist.

```nix
{ stdenv, fetchurl }:

stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.10";

  ...

  buildPhase = ''
    gcc hello.c -o hello
  '';

  installPhase = ''
    mkdir -p $out/bin
    cp hello $out/bin
  '';
}
```

(a) Příklad redefinice fází standardního sestavovacího skriptu. Ve výpisu jsou redefinované pouze fáze Build a Install. Ostatní fáze mají výchozí chování.

```nix
{ stdenv, fetchurl,
  clang, lib, fooDep, barDep }:
stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.10";

  ...

  buildInputs = [ lib ];
  nativeBuildInputs = [ clang ];
  propagatedBuildInputs = [
    fooDep
  ];
  propagatedNativeBuildInputs = [
    barDep
  ];
}
```

(b) Příklad deklarace různých typů závislostí. Nejdříve je potřeba závislost přidat do parametrů funkce balíčku a poté přiřadit do správného seznamu.

Výpis 3.7: Ukázky specifického nastavení derivace při použití pomocné funkce `stdenv.mkDerivation`.

[10]Definice fází – https://github.com/NixOS/nixpkgs/blob/master/pkgs/stdenv/generic/setup.sh

Pokud vytváří derivace nějaký soubor, musí jej uložit do cesty `$out`, která odpovídá cestě výstupu derivace v `/nix/store`. Například ve výpisu 3.7a je na řádku 15 spustitelný soubor zkopírován do cesty `$out/bin`. Výstupů je ale možné definovat více než jenom výchozí `out`. Například výstup `man` pro manuálové stránky, nebo výstup `doc` pro dokumentaci. V takovém případě budou výstupy uloženy v `/nix/store` odděleně v samostatných složkách/souborech s příponou názvu výstupu. Například výstup `doc` balíčku `hello`, bude uložen v adresáři `/nix/store/svs57njs7lp...-hello-2.10-doc`.

Derivace musí vyprodukovat nějaký výstup. To znamená, že musí být vytvořen soubor nebo adresář uložený v proměnné `$out`, jinak je sestavení derivace neúspěšné. Každý výstup derivace by měl být strukturovaný dle konvencí. Například spustitelné soubory by měly být uloženy v adresáři `bin`, konfigurační soubory v adresáři `etc`, hlavičkové soubory pro kompilátory v adresáři `include` a knihovny pro dynamický linker v adresáři `lib`. Díky tomu je možné přistupovat k nainstalovaným balíčkům jednotně a jednoduše je integrovat do systému.

Definice závislostí

Kromě postupu sestavení je možné v derivaci specifikovat závislosti. Protože je Nix source based balíčkovací systém, je potřeba rozlišovat mezi jejími různými druhy. Některé závislosti jsou zapotřebí jen při sestavení balíčku. Takové závislosti se označují jako *build-time* závislosti. Ty, které jsou potřeba za běhu programu, se označují jako *run-time* závislosti. Tyto dvě kategorie závislostí nejsou vzájemně výlučné. Závislost může být potřeba jak při sestavení, tak za běhu programu. V binary based balíčkovacích systémech se definují pouze run-time závislosti.

Typickým příkladem build-time závislosti je kompilátor. Jakmile je balíček sestaven, není už kompilátor potřeba. Typickou run-time závislostí je sdílená knihovna u dynamicky slinkovaného programu. Pokud je ale program slinkovaný staticky, je tato knihovna build-time závislostí.

Balíček je specifikován v jazyku Nix a po jeho vyhodnocení je uložena v `/nix/store` jeho derivace (`.drv` soubor). Derivace obsahuje popis sestavení balíčku a odkazy na všechny jeho závislosti. Po sestavení derivace vznikne výstup derivace. Závislost mezi objekty v `/nix/store` je vyjádřena tak, že některý ze souborů objektu obsahuje název jiného objektu. V případě souboru `.drv` jsou závislosti uloženy strukturovaně ve speciálním formátu a dají se tak jednoduše získat. V případě výstupu derivace je situace složitější.

Při sestavování derivace jsou známy všechny její závislosti. Výstup derivace by ale neměl mít stejné závislosti jako derivace samotná. Například kompilátor pro běh programu není potřeba. Je proto potřeba pro výstup derivace, stejně tak jako u `.drv` souboru, identifikovat všechny jeho závislosti. Díky tomu, jak jsou v Nix pojmenovány objekty, je možné tyto závislosti automaticky rozpoznat. Po sestavení se v obsahu všech souborů výstupu derivace vyhledávají názvy závislostí, konkrétně podřetězce ve tvaru `/nix/store/[a-z0-9]{32}`. Pokud je takovýto podřetězec nalezen, znamená to run-time závislost na obsahu této cesty.

Závislosti specifikované v `.drv` souboru jsou tedy build-time i run-time závislosti balíčku. Cesty k objektům v `/nix/store` zapsané v souborech výstupu derivace jsou už pouze run-time závislosti balíčku. Nix rozpoznává run-time závislosti z buld-time závislostí automaticky. Například v popisu balíčku ve výpisu 3.7a je dostupná standardní knihovna jazyka C (glibc), protože je použita derivace `stdenv`. Při sestavení pak vznikne binární soubor `hello` ve formátu ELF, kde v hlavičce `RUNPATH` bude řetězec `/nix/store/6yaj6n8192...-glibc-2.27/lib`. Soubor `hello` má tedy run-time závislost na standardní knihovně jazyka

C. Díky automatickému rozpoznávání závislostí, jsou tak závislosti přesně a kompletně určeny.

Protože jsou objekty v `/nix/store` neměnné, stačí provést skenování závislostí pouze jednou a uložit výsledky. K tomu Nix používá SQLite databázi umístěnou v `/nix/var/nix/db/db.sqlite`. Díky tomu je graf závislostí objektu dostupný rychle a používání příkazů Nix je tak pohodlnější.

Ve výpisu 3.7b je příklad deklarace závislostí balíčku při použití pomocné funkce `stdenv.mkDerivation`. Balíček je funkce a každá závislost se nejdříve musí uvést jako její parametr. Poté je závislost přiřazena do seznamu k atributu odpovídající typu závislosti. Například na řádku 9 je závislost `lib` v seznamu `buildInputs`. Do tohoto seznamu se přidávají run-time závislosti, které mohou být potřebné i k sestavení například kvůli statickému slinkování (jedná se většinou o knihovny nebo interpret). Na řádku 10 je pak závislost `clang`, která je v seznamu `nativeBuildInputs`. V tomto seznamu jsou build-time závislosti, které by neměly být rozpoznávány jako run-time závislosti, např. kompilátor.

V některých případech ale sestavení programu nezpůsobí vepsání run-time závislosti do výstupu derivace. Například interpretované programy v jazyku Python nebudou obsahovat textové řetězce názvu závislostí. Proto musí být do výstupu derivace explicitně vloženy. Navíc pokud se jedná o knihovnu v jazyku Python, tak by se měly závislosti propagovat do dalších, na tomto balíčku závislých, balíčků. Z toho důvodu se navíc dají specifikovat `propagated*` varianty build-time a run-time závislostí, které jsou použity na řádku 11 a 14 výpisu 3.7b. Takové závislosti jsou standardním sestavovacím skriptem uloženy do výstupu derivace do adresáře `nix-support`. Díky tomu je možné automatickým skenováním tyto run-time závislosti nalézt. Samotný Python program pak musí být spuštěn pomocí pomocného skriptu, který závislosti z `nix-support` nejdříve importuje.

Sestavování derivací má jako cílovou platformu tu, na které je Nix nainstalován. Nix ale podporuje i *cross-compilation*. Pro plnou podporu cross-compilation ale může být specifikace závislostí poněkud složitější. Podrobnou dokumentaci ke cross-compilation lze nalézt v manuálu k Nixpkgs [18].

Nixpkgs: kolekce Nix balíčků

Balíček je v Nix typicky zadefinován dle konvence jako funkce, která má jako parametry závislosti a případně i parametry pro různé varianty. Tato funkce musí být poté někde aplikována se správnými parametry, aby vznikla kompletní definice balíčku. Balíčků může být definováno více a mohou být závislé mezi sebou. V nejjednodušším případě může jít o rekurzivní množinu atributů obsahující všechny balíčky jako atributy. Balíčkům pak mohou být předávány jako parametry jiné balíčky. Výhoda toho, že jsou balíčky definovány jako funkce a že jsou pak všechny navzájem propojené je, že když se změní jeden balíček, tak všechny na něm závislé balíčky budou ihned používat novou definici.

Balíček tedy pouze deklaruje své závislosti a až v kolekci balíčku, jako ve výpisu 3.8, jsou závislosti teprve přesně definovány. Protože jazyk Nix podporuje lazy evaluation, mohou být všechny balíčky definované na jednom místě. Samotné sestavení a instalace daného balíčku pak je jenom o výběru atributu v množině atributů. Pokud se seznam všech balíčků nachází v souboru `all-packages.nix` provede se instalace balíčku `hello` následovně:

```
$ nix-env -f all-packages.nix -i -A hello
```

```nix
rec {
  gcc8 = import gcc.nix { inherit coreutils; };
  gcc = gcc8
  coreutils = import coreutils.nix {};
  hello = import hello.nix { inherit gcc coreutils; };
  foo = import foo.nix { inherit coreutils; dev = false; }
  fooDev = import foo.nix { inherit coreutils; dev = true; };
  ...
}
```

Výpis 3.8: Kolekce všech dostupných balíčků jako rekurzivní množina atributů. Každý balíček je funkce, které musí být předány závislosti a případně další hodnoty.

Repozitář *Nixpkgs*[11] obsahuje všechny balíčky v podobném souboru `all-packages.nix`. Nixpkgs je jednoduše jenom kolekce `.nix` souborů balíčků roztříděných do různých adresářů podle jejich typu a několik málo `.nix` souborů tvořících jejich kompozice. Podobná stromová struktura všech balíčků se používá i v balíčkovacích systémech ABS a Homebrew.

Pokud se používá výchozí instalace Nix, pracují všechny Nix příkazy právě s repozitářem Nixpkgs. V systému se nachází lokální kopie celého repozitáře. Cesta k tomuto repozitáři je uložena ve speciální Nix proměnné `<nixpkgs>`, která je specifikovaná v proměnné prostředí `NIX_PATH`. Všechny příkazy pak této proměnné využívají a není proto potřeba například při použití `nix-build` explicitně uvádět soubor se seznamem všech balíčků. Proměnná `<nixpkgs>` je také dostupná ve všech Nix výrazech.

Repozitář Nixpkgs se mění každým dnem, a ne vždy jsou všechny změny otestovány. Proto se balíčky distribuují skrze tzv. *kanály balíčků (channels)*. Jedná se o ověřené kopie repozitáře Nixpkgs, které jsou postupně aktualizovány po otestování změn v Nixpkgs. Existuje například kanál balíčků `nixos-20.03` nebo `nixpkgs-unstable`. Uživatel pak může zvolit, jaký kanál balíčků se má používat jako výchozí.

Pro každý program a knihovnu musí být většinou manuálně vytvořen popis sestavení v jazyku Nix. Naštěstí komunita kolem Nix je velká (v roce 2019 více než tisíc aktivních přispěvatelů[12]) a většina balíčků je už vytvořených a jsou udržovány. V době psaní této práce obsahoval repozitář Nixpkgs více než *60 tisíc*[13] balíčků. Navíc Nix používá čím dál více velkých firem a organizací, například Mozilla[14], CERN[15] nebo Tumblr[16].

Pro vývojáře pracující na svých privátních projektech tak už jen zbývá, aby svůj projekt zadefinovali také jako balíček. Pokud závislosti k tomuto projektu jsou v repozitáři Nixpkgs, stačí je importovat pomocí proměnné `<nixpkgs>`. Pokud ale některá závislost chybí, nebo není ve správné variantě, musí vývojář kolekci balíčků v Nixpkgs rozšířit nebo upravit. To lze provést několika způsoby:

- Je možné udělat vlastní kopii celého repozitáře Nixpkgs a upravit nebo přidat `.nix` soubory podle potřeb. Je ale obtížné udržovat takto velký repozitář a automaticky backportovat změny z Nixpkgs.

[11] Repozitář Nixpkgs – https://github.com/NixOS/nixpkgs
[12] NixCon 2019: *NixOS 19.09 release talk* – https://www.youtube.com/watch?v=pfg9ykBo9oM
[13] Vyhledávání balíčků v Nixpkgs – https://nixos.org/nixos/packages.html
[14] nixpkgs-mozilla – https://github.com/mozilla/nixpkgs-mozilla
[15] LHCb – https://cds.cern.ch/record/2700235/files/10.1051_epjconf_201921405005.pdf
[16] Jetpants – https://github.com/tumblr/jetpants/tree/master/testing

- Další možností je komunitní projekt *Nix User Repository (NUR)*, kde každý vývojář může zaregistrovat svůj repozitář se svými balíčky. Po registraci je možné přistupovat k těmto balíčkům přes atribut `nur`. Například balíček `helloworld` uživatele `alice` bude dostupný skrze atribut `pkgs.nur.alice.helloworld`. Problémem tohoto přístupu je, že balíčky v zaregistrovaném repozitáři mohou být závislé na jiné verzi Nixpkgs než ta, která je aktuálně dostupná, a proto nemusí vždy fungovat.

- Dále je možné použít tzv. *overlays*. Importovaná proměnná `<nixpkgs>` v Nix výrazu je funkce s jedním parametrem. Tento parametr je množina atributů, ve které může být specifikován atribut `overlays` obsahující redefinici balíčků ve speciálním tvaru. Balíčky jsou v `overlays` definovány pomocí pevného bodu, a proto se jejich změna projeví ve všech závislých balíčcích. Seznam všech redefinic nebo přidaných balíčků pomocí overlays může být uložen v konfiguračním souboru `~/.config/nixpkgs/config.nix`. Tento soubor je automaticky načítán při použití Nixpkgs. [18]

- Poslední možností je projekt Nix Flakes[17]. Jedná se o náhradu za kanály balíčků a zároveň by měl umožňovat rozšíření dostupných balíčků v systému o repozitáře třetích stran. Tento projekt je ale stále ve vývoji a jeho specifikace ještě není finální.

Instalace a konfigurace Nix

Samotný Nix lze nainstalovat na jakékoliv linuxové distribuci (i686, x86_64, aarch64) a na macOS (x86_64) [17]. Na Windows není Nix nativně podporovaný, ale pomocí WSL (Windows Subsystem for Linux) je možné ho používat i na Windows. Instalace je možná ve dvou módech: *single user* a *multi user*. Pokud je Nix nainstalován v single user módu, může Nix používat pouze uživatel, který ho nainstaloval, a to hlavně kvůli vlastnictví souborů v `/nix/store`. Pokud je zvolena multi user instalace, tak mohou Nix používat všichni uživatelé. V tomto případě vlastní soubory v `/nix/store` speciální uživatel a všechny Nix příkazy komunikují s *Nix démonem*. Každý uživatel může instalovat balíčky bez speciálního oprávnění, protože díky hashi se nemohou uživatelé ovlivnit navzájem a nemohou nainstalovat nějaký program globálně. Většina standardních správců balíčků funguje právě v single user módu, kde tím jedním uživatelem s výhradním právem je uživatel root. S objekty v `/nix/store` by neměl uživatel manuálně manipulovat, protože by mohl narušit jejich integritu. Všechny soubory jsou proto read-only a vytváření objektů zprostředkovávají dostupné příkazy.

Zvolený mód instalace ovlivňuje i možnosti sandboxu pro sestavování derivací. Při single user módu nelze vytvořit plně oddělené prostředí, protože sestavení není spuštěno s právy uživatele root. I tak ale sandbox částečně funguje a ve většině případů bude i dostačující. Jak bylo zmíněno výše, v multi user instalaci komunikují všechny příkazy s běžícím démonem. Ten je spuštěn pod uživatelem root a díky tomu je možné pro sestavování používat plně oddělené prostředí.

Po instalaci jsou dostupné následující příkazy:

- `nix-build` – vytvoření a sestavení derivace
- `nix-channel` – manipulace s kanály balíčků
- `nix-collect-garbage` – spuštění garbage collectoru

[17] NixCon 2019: *Nix Flakes* – https://www.youtube.com/watch?v=UeBX7Ide5a0

- `nix-copy-closure` – kopírování closure přes SSH
- `nix-daemon` – démon spuštěný v multi user módu
- `nix-env` – manipulace s profily
- `nix-hash` – utilita pro vytváření hashe souborové cesty
- `nix-instantiate` – vytvoření derivace
- `nix-prefetch-*` – různé pomocné utility pro stažení souboru a získání jeho hashe
- `nix-shell` – spuštění subshellu, ve kterém jsou dostupné závislosti definované v derivaci
- `nix-store` – manipulace s úložištěm objektů

Dále je navíc dostupný příkaz `nix`, který je ale zatím stále označený jako experimentální. Tento nový příkaz má modernější rozhraní a měl by pomocí podpříkazů (podobně jako například podpříkazy programu `git`) postupně nahradit všechny výše zmíněné příkazy[18].

Po instalaci je Nix automaticky integrován do prostředí uživatele. Ve výchozím nastavení by mělo vše fungovat dle očekávání. Specifické požadavky lze nakonfigurovat v globálním konfiguračním souboru `/etc/nix/nix.conf` nebo pro každého uživatele zvlášť v souboru `~/.config/nix/nix.conf`.

3.3 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 úložiště 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 úložištěm konfiguračních souborů v adresáři `/etc`.

Nekompatibilní verze konfiguračních souborů se projevují nejčastěji při aktualizaci balíčků. Pokud uživatel konfigurační soubory upravil, musí být nějakým způsobem změny aplikovány i na novější verzi konfiguračních souborů. Je několik možností, jak se v tomto případě postupuje:

- Změny se ignorují a do systému se nainstaluje nová verze konfiguračních souborů.
- Zachová se stará verze konfiguračních souborů.
- Je vyžadován manuální zásah od uživatele.
- Společně s aktualizací balíčku je dodán skript, který se může pokusit provést migraci konfiguračních souborů. [7]

[18] NixCon 2017: *Nix 1.12 by Eelco Dolstra* – https://www.youtube.com/watch?v=XVIKScU7Uf4

Po kterékoliv akci zmíněné výše nemusí být vždy jisté, jestli bude aktualizovaný balíček správně fungovat. Navíc není nikde záznam o provedených změnách v konfiguračních souborech. Správce systému nemůže před samotnou aktualizací jednoduše otestovat stav systému, který nastane po aktualizaci. Pokud se něco pokazí, nemůže navíc snadno provést rollback aktualizace. Mnoho nainstalovaných balíčků v různých verzích a mnoho upravených konfiguračních souborů velmi ztěžuje reprodukovatelnost samotného operačního systému.

Pro sledování historie úprav existuje například nástroj Etckeeper[19]. Ten uchovává konfigurační soubory v repozitáři Git. Etckeeper ale není propojený s balíčkovacím systémem, takže změna verze balíčku neprovede automaticky změnu na odpovídající verzi konfiguračních souborů. Navíc nemá Etckeeper informaci o tom, jak aplikovat změnu verze konfiguračních souborů. Někdy je potřeba restartovat nějakou službu systému nebo vykonat specifický příkaz či skript.

Hlavním problémem klasických unix-like operačních 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 [7]. 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 konfigurace (CMS) a jsou to například nástroje Ansible[20] a Puppet[21]. 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* [5].

Jak už název napovídá, operační systém *NixOS* používá bezstavový správce balíčků Nix. Celý systém je deklarativně popsán pomocí jednoho Nix výrazu 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* [5]. Následující podsekce z velké části čerpají z článku [7] a manuálu k NixOS [19].

Využití Nix pro sestavení systému

V operačním systému NixOS jsou všechny komponenty (včetně jádra, balíčků a konfiguračních souborů) sestaveny pomocí Nix. Konfigurační soubory jsou po sestavení uloženy v `/nix/store` a z pohledu Nix se jedná o 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.

Programy mají explicitní závislost na svých konfiguračních souborech s přesně definovaným obsahem (hash derivace konfiguračních souborů). Konfigurační soubory mohou obsahovat odkazy na jiné programy a mohou být tedy závislé na dalších balíčcích. Nix zajistí, že všechny závislosti programu, které jsou potřeba, budou dostupné, a že nebudou odstraněny po spuštění garbage collectoru. Díky tomu má tento systém podobné vlastnosti jako správce balíčků Nix. Je reprodukovatelný, podporuje atomické změny a je možné jed-

[19] Etckeeper – https://etckeeper.branchable.com
[20] Ansible – https://www.ansible.com
[21] Puppet – https://puppet.com

noduše provést rollback. Při aktualizaci nebo rollbacku má každý balíček vždy správnou verzi konfiguračních souborů. Kvůli použití balíčkovacího systému Nix avšak systém NixOS nevyhovuje standardu FHS[22].

Ne všechny konfigurační soubory je bohužel možné uložit v Nix store. Například soubor `/etc/resolv.conf` je upravován DHCP klientem za běhu systému a nelze jej tedy spravovat deklarativně. Soubory jako `/etc/hosts` jsou sice statické, ale spousta programů spoléhá na jejich existenci v adresáři `/etc` a nenabízí možnost změny jejich cesty. Konfigurační soubory modifikované za běhu (jako je `/etc/resolv.conf`) nejsou proto spravovány pomocí Nix a musí být uloženy přímo v adresáři `/etc` (v NixOS 20.03 se jedná o 11 souborů). Ostatní globální konfigurační soubory, které jsou statické (jako je `/etc/hosts`), jsou vygenerovány pomocí Nix, uloženy v Nix store, a poté jsou pomocí symlinku vloženy do adresáře `/etc`. To zanáší do konfigurace systému globální stav, ale jedná se jenom o malou množinu konfiguračních souborů. Podobně musí existovat i symlinky `/bin/sh` a `/usr/bin/env`, kvůli skriptům, které obsahují odkaz na interpret (shebang).

Všechny spustitelné programy v systému jsou uloženy v adresáři `/nix/store`, ve kterém jsou všechny soubory přístupné a mají nastavená stejná práva přístupu. To znamená, že v tomto adresáři nemohou mít spustitelné binární soubory nastaven příznak *setuid*, který potřebuje pro svůj běh například program `passwd`.

Konfigurace systému a moduly

Celý systém je sestaven na základě vyhodnocení výrazu v hlavním konfiguračním souboru systému `/etc/nixos/configuration.nix`. Příklad takovéhoto souboru je ve výpisu 3.9a. Po každé změně tohoto souboru stačí spustit s privilegovaným oprávněním příkaz:

```
$ nixos-rebuild switch
```

a celý systém se přepne do nové konfigurace. Po každém spuštění tohoto příkazu se generují nové konfigurace systému, podobně jako profily při instalaci balíčků a stejně tak lze mezi nimi i přepínat nebo provést rollback. Společně s novou konfigurací se vytváří i nová položka v zavaděči GRUB odpovídající nové konfiguraci. Uživatel má tak před spuštěním systému NixOS na výběr mezi aktuální konfigurací a seznamem předcházejících konfigurací systému.

Při přepnutí na jinou konfiguraci v běžícím systému se automaticky rozpozná, které služby musejí být restartovány a které ne. Vytvoří se *aktivační skript*, který je poté spuštěn s právy uživatele root. Restart služeb ale není atomický a existuje tedy časové okno, kdy může být systém v nekonzistentním stavu. Nicméně díky nové položce v GRUB bude po restartu systém vždy v požadovaném stavu. Stejně jako profily, mohou být staré konfigurace odstraněny a spuštěn garbage collector.

V předchozí sekci bylo zmíněno, že v Nix store nemohou mít spustitelné soubory nastavený příznak setuid. Součástí aktivačního skriptu pro přepnutí konfigurace jsou proto příkazy na vytvoření obalovacích programů nad těmito speciálními spustitelnými soubory, které jsou následně uloženy v `/run/wrappers/bin`. Tyto obalující programy mají nastavený příznak setuid a spouštějí pak daný spustitelný soubor v `/nix/store`. Díky tomuto konceptu mohou být v systému používány i programy, které jsou uložené v Nix store a které jinak vyžadují pro svůj běh příznak setuid. Protože k přepnutí konfigurace jsou zapotřebí privilegovaná oprávnění, nemůže uživatel bez dostatečných práv vytvořit vlastní obalovací programy s příznakem setuid.

[22] Filesystem Hierarchy Standard – https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf

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

  imports = [
    ./hello.nix
  ];

  time.timeZone="Europe/Prague";

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

  users.users.tom = {
    isNormalUser = true;
    home = "/home/tom";
  };

  services.openssh.enable = true;
  services.hello.enable = true;

  environment.systemPackages = [
    pkgs.vim
  ];
}
```

(a) Příklad hlavního konfiguračního souboru systému NixOS, který importuje modul ze souboru `hello.nix`. V systému je připojené zařízení `/dev/sda1`, vytvořen uživatel `tom`, povoleny služby `openssh` a `hello` a nainstalován program `vim`.

```nix
{ config, lib, pkgs, ... }:
with lib; with types;
let
  cfg = config.services.hello;
in {

  # rozhraní
  options.services.hello = {
    enable =
      mkEnableOption "Hello";
  };

  # implementace
  config = mkIf cfg.enable {
    systemd.services.hello = {
      wantedBy = [
        "multi-user.target"
      ];
      path = [ pkgs.echo ];
      script = ''
        echo "Hello world"
      '';
    };
  };
}
```

(b) Příklad definice modulu služby `hello` v souboru `hello.nix`. Pokud je služba `hello` povolena, je upravena konfigurace modulu `systemd`, jenž způsobí spuštění služby `hello` po spuštění systému.

Výpis 3.9: Modulární konfigurace systému NixOS.

Možnosti konfigurace celého systému jsou definovány v *modulech*. Ty jsou stejně jako hlavní konfigurační soubor napsané v jazyku Nix a jsou importovány (například řádek 4 ve výpisu 3.9a) do hlavního konfiguračního souboru. Každý modul definuje možnosti, které nabízí, a může ovlivňovat jiné moduly. Například modul ve výpisu 3.9b definuje možnost pro zapnutí služby `hello` a vkládá novou službu do modulu `systemd`. Stejně jako balíčky pro Nix, jsou moduly uloženy v repozitáři Nixpkgs.

Každý modul je funkce ve tvaru:

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

Atributy `imports`, `options` nebo `config` jsou volitelné a pokud je v modulu definován jiný atribut než tyto tři, tak se tento automaticky vloží do množiny atributů `config`. Jedná se o konvenci pro lepší čitelnost kódu modulu. Například hlavní konfigurační soubor ve výpisu 3.9a je ve skutečnosti také modul, který nedefinuje atribut `options` a všechny v něm uvedené atributy se automaticky vloží do množiny atributů `config`.

Funkce modulu dostane jako argument konfiguraci systému vyhodnocenou ze všech modulů (i ze svého). Vstup modulu je tedy závislý na jeho výstupu a může dojít k cyklické závislosti. Z toho důvodu je ve výpisu 3.9b na řádku 14 použita pomocná funkce `mkIf`, která tuto cyklickou závislost odstraňuje. Funguje tak, že vkládá vyhodnocení podmínky do vnitřních atributů předané konfigurace a odkládá tak vyhodnocení do jiných modulů. Díky lazy evaluation pak může být i takováto cyklická závislost modulu na sebe samém vyhodnocena. [7]

Zajímavou možností konfigurace systému NixOS jsou *NixOS kontejnery*. Ty obalují systemd kontejnery (systemd-nspawn) tak, aby mohly být definovány deklarativně v konfiguraci systému. Tyto kontejnery jsou podobné Docker kontejnerům, akorát se o jejich běh stará systemd a k jejich správě se používá příkaz `nixos-conatiner`. Každý kontejner je samostatný systém NixOS popsaný deklarativně stejným způsobem jako hostitelský systém. Příklad definice NixOS kontejneru v konfiguraci systému je ve výpisu 3.10. Mezi kontejnerem a hostitelským systémem se navíc vytvoří virtuální síť, aby spolu mohly komunikovat. Důležité ale je, že kontejnery sdílí s hostitelským systémem adresář `/nix/store`, takže jejich vytvoření je rychlé a jejich velikost je minimální.

```
1  { pkgs, config, ... }: {
2
3    ...
4
5    containers.database = {
6      # modul nebo konfigurace celého systému
7      config = { config, pkgs, ... }: {
8          services.postgresql.enable = true;
9          services.postgresql.package = pkgs.postgresql_9_6;
10       };
11     };
12   }
13 }
```

Výpis 3.10: Příklad definovaní NixOS kontejneru v konfiguraci hostitelského systému NixOS. Do atributu `config` může být přiřazena jakákoliv konfigurace systému NixOS. V tomto případě bude v kontejneru čistý systém se zapnutou službou `postgresql`. Příklad byl převzat z [19].

Aktuálně jsou k dispozici tisíce možností konfigurace a lze v nich jednoduše vyhledávat[23]. Díky reprodukovatelnosti je možné konfigurace i velmi snadno testovat. NixOS totiž nabízí příkaz pro sestavení systému a spuštění ve virtuálním stroji pomocí QEMU/KVM:

```
$ nixos-rebuild build-vm
```

[23] Search NixOS options – https://nixos.org/nixos/options.html

Takto vytvořený virtuální stroj načte jádro přímo z běžícího systému a připojí dovnitř i adresář `/nix/store`, takže spuštění je rychlé a bez nutnosti vytváření obrazu virtuálního stroje. Také je možné testovat novou konfiguraci v běžícím systému bez přidání položky do zavaděče GRUB, nebo aktivovat novou konfiguraci až po restartu systému.

Kromě globální konfigurace systému může uživatel měnit systém i imperativně pomocí profilů. Profily jsou oddělené od konfigurací systému, takže instalace balíčku pomocí Nix nemění konfiguraci systému. Stejně tak i kanály balíčků se nastavují pro každého uživatele zvlášť.

Na oficiálních stránkách NixOS[24] lze stáhnout obraz virtuálního stroje s nainstalovaným systémem NixOS a prostředím KDE ve formátu OVA. Další možností je stažení ISO souboru, a to varianty s grafickým prostředím nebo minimální distribuce. Je nutno podotknout, že systém NixOS zatím nedisponuje typickým grafickým instalátorem a manuální instalace je tedy vhodná spíše pro pokročilejší uživatele. Pro kontejnerizaci lze také využít oficiálního NixOS Docker image, nebo pomocí jednoduchého postupu vytvořit Amazon EC2 či Microsoft Azure instanci.

Alternativou k NixOS je systém Guix[25], který byl také vytvořen na základě Nix. Na rozdíl od NixOS nepoužívá jazyk Nix ale Scheme, konkrétně jeho implementaci Guile. Guix ale nedisponuje tolika balíčky jako NixOS a nemá tak velkou komunitu.

3.4 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 sekci 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 systémů NixOS více, které jsou mezi sebou propojené, tak je potřeba je konfigurovat společně.

Další technologií postavenou nad Nix je *NixOps*, která slouží jako nástroj pro IaC. Tento nástroj je úzce spjatý s NixOS a umožňuje deklarativně popsat infrastrukturu a konfiguraci jednotlivých strojů. Na základě vyhodnocení tohoto popisu NixOps vykoná nutné kroky nebo akce k tomu, aby byla infrastruktura v požadovaném stavu. V SQLite databázi se uchovává stav strojů (ve výchozím nastavení v souboru `~/.nixops/deployments.nixops`) a při opětovném spuštění se provádí jenom ty akce, které jsou potřeba [20].

K popisu infrastruktury se používá jazyk Nix, stejně jako u systému NixOS a definice balíčků. Podobně jako u deklarativního zápisu konfigurace systému NixOS, může být zápis infrastruktury strukturován do menších celků. Celá infrastruktura se popíše jako množina konfigurací systému NixOS. Příklad takovéto infrastruktury je ve výpisu 3.11. Kromě všech možností konfigurace systému NixOS, je možné u strojů nastavit speciální atributy. Ve výpisu 3.11 je to atribut `deployment.targetEnv` na řádku 3 a 16, kterým se určí cílové prostředí pro nasazení. NixOps podporuje tato cílová prostředí:

- VirtualBox VM
- NixOS
- Amazon EC2
- GCE
- Microsoft Azure
- Hetzner
- Digital Ocean
- Libvirtd (Qemu)
- NixOS kontejner[26]

[24] NixOS – https://nixos.org
[25] Guix – https://guix.gnu.org
[26] Možnost nasazení do NixOS kontejnerů není uvedena v dokumentaci a je zatím spíše experimentální.

Existují ale i další speciální atributy. Například `deployment.keys`, pomocí kterého je možné na stroji uložit důvěrná data (hesla, klíče) v dočasném souborovém sytému `/run/keys`. Důvěrná data nemohou být totiž nastavena v konfiguraci systému, protože při vyhodnocení výrazu konfigurace systému by byla uložena — stejně jako všechno ostatní — do `/nix/store`, bez jakéhokoliv zabezpečení.

```
1  {
2    webserver = {
3      deployment.targetEnv = "virtualbox";
4      deployment.keys.secret.text = "Some secret";
5      services.httpd.enable = true;
6      services.httpd.virtualHosts = {
7        "example.org" = {
8          documentRoot = "/data";
9      };
10     fileSystems."/data" = {
11       fsType = "nfs4";
12       device = "fileserver:/";
13     };
14   };
15
16   fileserver = {
17     deployment.targetEnv = "virtualbox";
18     services.nfs.server.enable = true;
19     services.nfs.server.exports = "...";
20   };
21 }
```

Výpis 3.11: 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 nástroje VirtualBox.

Uloží-li se popis infrastruktury ve výpisu 3.11 do souboru `infrastructure.nix`, je možné provést vytvoření a nasazení infrastruktury následujícími příkazy:

```
$ nixops create infrastructure.nix --deployment production
$ nixops deploy --deployment production
```

Při každé další změně infrastruktury už stačí spustit jenom příkaz `nixops deploy`. Při nasazení jsou nově potřebné stroje vytvořeny, stávající jsou případně aktualizovány a již nepotřebné jsou odstraněny. Pokud se po opětovném spuštění nasazení infrastruktury nic nezměnilo, tak se nic neprovádí, tzn. nasazení infrastruktury je *idempotentní*.

Protože se pro nasazení používá systém NixOS a balíčkovací systém Nix, nepřepisují se na cílových strojích žádné soubory a staré konfigurace jsou na strojích stále uloženy v adresáři `/nix/store`. Díky tomu je možné snadno provést rollback příkazem:

```
$ nixops rollback --deployment production
```

Příkaz `nixops` nabízí mnoho dalších operací, jako je zastavení/spuštění daného stroje, spuštění příkazu na všech strojích, nebo záloha/obnovení diskových zařízení. Podrobný popis je v dokumentaci k NixOps [20].

Podobně jako NixOps funguje i Terraform[27], který také spravuje infrastrukturu pomocí deklarativního zápisu. Tento nástroj podporuje velké množství cloudových poskytovatelů a je de facto standardem pro IaC. Chybí ale integrace s Nix a pro popis infrastruktury používá svůj vlastní DSL.

3.5 Hydra: systém kontinuálního sestavování založený na Nix

Nástroj pro průběžnou integraci Hydra používá pro popis akcí a jejich závislostí Nix. Díky tomu je prostředí pro sestavení vytvořeno automaticky a deterministicky. Navíc mohou být jednoduše sestaveny a testovány různé varianty software, protože balíčky jsou funkce. [9]

V nástroji Hydra se definují akce, které se mají provést při změně kódu v repozitáři. Podle konvence je tato definice uložena v kořenu repozitáře v souboru `release.nix`. Příklad definice akcí prováděných při průběžné integraci je ve výpisu 3.12.

```
1  let
2    pkgs = import <nixpkgs> {};
3
4    jobs = rec {
5
6      tarball =
7        pkgs.releaseTools.sourceTarball {
8          name = "hello-tarball";
9          src = <hello>;
10         buildInputs = (with pkgs; [ gettext texLive texinfo ]);
11       };
12
13     build =
14       { system ? builtins.currentSystem }:
15
16       let pkgs = import <nixpkgs> { inherit system; }; in
17       pkgs.releaseTools.nixBuild {
18         name = "hello";
19         src = jobs.tarball;
20         configureFlags = [ "--disable-silent-rules" ];
21       };
22   };
23 in
24   jobs
```

Výpis 3.12: Příklad specifikace akcí (jobs) v Hydra [21]. Akce mohou být propojené. Například na řádku 19 využívá akce `build` v atributu `src` výstup akce `tarball`.

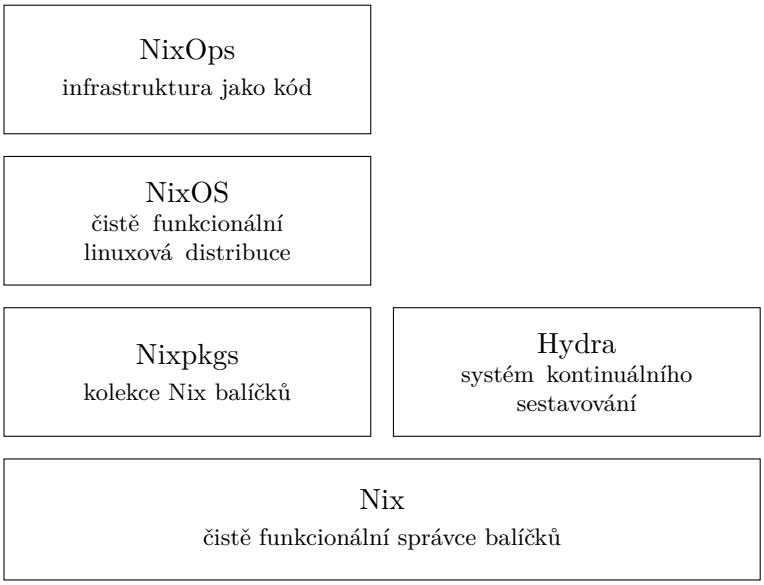
[27] Terraform – https://www.terraform.io

Tento nástroj byl vytvořen samotnými tvůrci Nix. Jako backend pro ukládání informací o vykonaných sestaveních a jejich stavu slouží PostgreSQL databáze. Nejedná se o rozšířený CI systém a bohužel není jednoduché tento nástroj zprovoznit. V době psaní této práce nefungovala manuální instalace uvedená v manuálu a jediná možnost, jak nástroj Hydra používat, bylo použít modul v konfiguraci systému NixOS.

Velmi podobným nástrojem je Hercules[28], který je nabízen jako SaaS. Jedná se o reimplementaci Hydra v jazyku Haskell tak, aby byla stále kompatibilní s backendem systému Hydra. Definice akcí se provádí velmi podobně jako v případě Hydra. Hercules umí navíc pro průběžné nasazení využívat Terraform a sestavování je provozováno na bázi agentů na infrastruktuře uživatele. Používání agentů má velkou výhodu v tom, že může být balíček sestavován a testován na různých platformách.

Dalšími systémy CI postavenými nad Nix jsou *micro-ci*[29] a *Scylla*[30]. Tyto dva nástroje fungují velmi jednoduše – vykonávají atributy v množině atributů. Příklad souboru `release.nix` by tak fungoval i pro ně. Na rozdíl od nástroje Hydra ale nemají uživatelské rozhraní a fungují pouze s repozitáři u služby GitHub.

V této kapitole byly popsány oficiální technologie kolem balíčkovacího systému Nix. Nejedná se o vyčerpávající dokumentaci, ale spíše o představení jednotlivých technologií a shrnutí jejich vlastností. Vztah mezi nimi je zobrazen na obrázku 3.5. V následujícím textu budou zkratkou Nix/NixOps souhrnně označovány technologie v levém sloupci, tedy technologie Nix, Nixpkgs, NixOS a NixOps. Následující kapitola se zabývá možnostmi použití těchto technologií v praxi pro průběžnou integraci a nasazení software.



Obrázek 3.5: Technologie postavené na Nix a jejich provázanost. Každá technologie staví na technologii v bloku pod ní. Obrázek byl převzat z NixOS wiki stránek[31] a upraven.

[28] Hercules – https://github.com/hercules-ci
[29] micro-ci – https://github.com/ocharles/micro-ci
[30] Scylla – https://github.com/Mic92/scylla
[31] NixOS Wiki: *Nix Ecosystem* – https://nixos.wiki/wiki/Nix_Ecosystem

Kapitola 4

Návrh použití Nix pro průběžnou integraci a nasazení software

Správce balíčků Nix zajišťuje reprodukovatelnost sestavení, atomičnost aktivování a přesně specifikované závislosti balíčků. Operační systém NixOS přidává možnost použít Nix pro konfigurační soubory a umožňuje deklarativně popsat a reprodukovat konfiguraci celého operačního systému. Pro deklarativní zápis infrastruktury a konfigurace jednotlivých strojů pomocí NixOS slouží NixOps. Technologie představené v předchozí kapitole jsou tedy ideálními nástroji pro spolehlivou průběžnou integraci a nasazení software při agilním vývoji. V následujících sekcích jsou shrnuty nedostatky Nix a technologií kolem něj a představeny možnosti jejich využití, pro jednotlivé fáze CI/CD vymezené v sekci 2.3.

4.1 Zhodnocení současného stavu CI/CD a Nix/NixOps

V kapitole 2 byly nastíněny problémy existujících řešení CI/CD používaných při agilním vývoji.

- Bez kontejnerizace není snadné reprodukovat prostředí CI/CD serveru a reprodukovatelně spustit proces CI/CD lokálně. Není možné tak spolehlivě ověřit úspěšnost některých fází, ještě předtím, než jsou spuštěny na CI/CD serveru. Pro kritické operace, jako je například nasazení aplikace na produkční server, je toto zásadní.
- 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). Tento problém se netýká přímo CI/CD řešení ale spíše nástrojů CMS, které jsou CI/CD serverem spouštěny.
- 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 v předchozím textu převažují pozitivní vlastnosti, má Nix a technologie s ním spojené i několik nevýhod. Některé z nich mohou být v budoucnu vyřešeny v dalších verzích, ale některé už z principu návrhu Nix řešitelné nejsou.

- Nix má mnoho příkazů, a ne všechny mají stejné rozhraní. Například příkaz `nix-env` má jako parametr název derivace, kdežto příkaz `nix-build` název souboru. Tento problém by měl vyřešit nový příkaz `nix`.
- Jazyk Nix je DSL pro balíčky, ale tento jazyk nepoužívá žádný koncept balíčků (balíčky jsou funkce). Argumenty balíčku jsou nejen závislosti, ale i jiné parametry pro vytváření variant balíčku, což není rozlišeno.
- Pro vytvoření balíčku nebo integraci projektu s Nix musí být vytvořen Nix výraz.
- Jazyk Nix je silně typovaný, ale zároveň i dynamicky typovaný. To není úplně dobrá kombinace, ale na druhou stranu to umožňuje mít kompaktnější zápis balíčků.
- Pro Nix zatím není plná podpora ve vývojových prostředích (IDE) a jsou dostupné pouze zvýrazňovače kódu (IntelliJ, Gedit, Nano, Vim, ...). Výjimkou je Emacs, kde se dá zprovoznit jednoduché „našeptávání“ atributů.
- Jazyk Nix nepodporuje speciální dokumentační komentáře. Atribut `meta` využívaný v definici balíčků pro metadata, není konstruktem jazyka ale jen konvence.
- Instalace nebo aktualizace balíčku je provedena změnou symlinku. To například znamená, že balíček nemůže spustit migraci dat nebo konfigurace (post install/update script).
- Pokud se změní některá nízkoúrovňová knihovna (například glibc), tak se musí takřka všechno znovu sestavit.
- Úložiště `/nix/store` není CAS (Content adressable 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 systému NixOS. Je potřeba si dát pozor, aby se klíče a hesla nezkopírovala do `/nix/store` při vyhodnocování výrazu.
- Při použití NixOps nelze vyjádřit závislosti služeb na jiných službách mezi různými stroji. Například pro webovou aplikaci může být požadováno, aby databázový server a na něm běžící služba MySQL byla dostupná dříve, než je spuštěna služba Apache na webovém serveru. Alternativou k NixOps může být nástroj *DisnixOS*, který tento problém řeší. Mezi službami na různých strojích umožňuje popsat závislosti a zajistit, z vnějšího pohledu, jejich atomické nasazení v celé infrastruktuře [25]. DisnixOS ale není tak rozšířený a není aktivně vyvíjený jako NixOps. Jeho popis a použití je mimo rozsah této práce.

Průběžná integrace byla jedna z počátečních cílů vývoje Nix. Hydra jako oficiální nástroj od tvůrců Nix ukazuje, jaké jsou možnosti Nix v oblasti CI. V jednom souboru jsou deklarativně definovány jednotlivé akce k vykonání, které mohou být navzájem i propojeny. Mezi akcemi se tak tvoří závislosti podobně jako mezi balíčky. Hydra pak už jenom pomocí Nix sestavuje definované akce, jako by to byly balíčky a prezentuje výsledky. Podobným způsobem fungují i nástroje micro-ci a Scylla.

Je možné nalézt několik málo projektů, které demonstrují použití Nix. Například projekt *nixtodo*[1] nebo *TodoMVC*[2]. Oba tyto příklady ale demonstrují použití Nix pouze pro

[1] nixtodo – https://github.com/basvandijk/nixtodo
[2] TodoMVC – https://github.com/nix-community/todomvc-nix

webovou aplikaci a pro začátečníka jsou velmi komplikované, neboť obsahují 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.

Neexistuje komplexní a jednotný přehled použití Nix pro aplikace různého druhu. Například webové, mobilní, distribuované atd. Dále, kvůli nutnosti definování každého balíčku pomocí Nix, je nutné mít příklady pro jednotlivé programovací jazyky a nástroje pro sestavení, případně obecný návod, jak jakýkoliv projekt, používající oficiálně nepodporované technologie, zprovoznit pomocí Nix. Nakonec chybí i demonstrace použití Nix pro všechny možnosti procesu CI/CD.

Jako CI server pro Nix balíčky se nejčastěji používá Hydra. Velkou nevýhodou CI systému Hydra, micro-ci a Scylla je jejich zaměření na Nix. Není možné pomocí nich provozovat CI i pro projekty, které Nix nepoužívají. S tím je spojená i zhoršená uživatelská přívětivost pro uživatele bez znalosti Nix. Nakonec je problém s nasazením aplikace – jedná se o CI systémy ne o CI/CD systémy. Ukázky použití Nix s klasickými CI/CD systémy se ale hledají velmi těžko.

4.2 Použití Nix pro vývoj

Nix má hned několik zajímavých vlastností pro vývoj. Za prvé díky explicitně definovaným závislostem je sestavení projektu otázkou jednoho příkazu. Dále zajišťuje Nix reprodukovatelnost, takže všichni vývojáři v týmu mají záruku stejného sestavení. Nakonec je možné i spustit subshell obsahující všechny závislosti definované v dané derivaci pomocí příkazu `nix-shell`. Při spuštění subshellu se nic neinstaluje do aktuálního profilu, ani se samotná derivace nesestavuje.

Pro použití Nix je zapotřebí zadefinovat projekt jako balíček. Pro tento balíček je pak možné spustit `nix-shell` nebo `nix-build`. Pro lokální vývoj je potřeba pracovat s lokálními zdrojovými soubory projektu, takže definice balíčku by neměla obsahovat odkaz na zdrojové soubory v podobě volání funkce `fetchurl`, jako tomu bylo v příkladě 3.6 v předchozí kapitole.

Pro používání příkazu `nix-build` s lokálními soubory, zprvu vypadá jako nejlepší změnit v derivaci hodnotu atributu `src` ze stažení vzdáleného zdroje (funkce `fetchurl`) na aktuální adresář, tedy `./`. Problém je, že toto způsobí kopii všech zdrojových souborů do `/nix/store`. Pokud pak projekt sestavíme pomocí `nix-build`, tak se vytvoří symlink `./result` odkazující na výstup derivace. To ale způsobí, že se aktuální adresář změnil (přibyl soubor `result`) a při příštím volání `nix-build` se všechny soubory opět nakopírují do `/nix/store`, přičemž se vytvoří nový, jiný symlink `./result`, protože aktuální sestavení je závislé na jiných zdrojových souborech. Opakované spouštění `nix-build` tak rekurzivně kopíruje všechny zdrojové soubory do `/nix/store`. Tento problém jsem vyřešil odfiltrováním souboru `./result` ze zdrojových souborů.

Při spuštění příkazu `nix-shell` se derivace nesestavuje, takže problém se souborem `result` nenastává. Jenže pokud by se použil předchozí způsob, budou se stále kopírovat všechny lokální soubory do `/nix/store` při každé změně jakéhokoliv souboru, a problém tak přetrvává. Tento problém jsem vyřešil změnou atributu `src` derivace na `null`. Aby se nemusel stále měnit atribut `src` mezi spouštěním příkazů `nix-build` a `nix-shell`, vytvoří se nový soubor `shell.nix` vedle souboru definice balíčku (`default.nix`). Příkaz `nix-shell` dokonce dává souboru `shell.nix` vyšší prioritu, než souboru `default.nix`, při spuštění bez dalších parametrů. Soubor `shell.nix` může vypadat například tak, jako ve výpisu 4.1.

```
1  { pkgs ? import <nixpkgs> {} }:
2  let
3    app = import ./default.nix;
4  in
5    app.overrideAttrs (oldAttrs: {
6      src = null;
7      nativeBuildInputs = oldAttrs.nativeBuildInputs ++ [ pkgs.vim ];
8    })
```

Výpis 4.1: Příklad souboru `shell.nix`, ve kterém je odstraněna definice zdrojových souborů. V tomto souboru je navíc možné definovat další závislosti potřebné jenom pro vývoj, jak je vidět na řádku 6, kde je přidán balíček `vim`.

4.3 Použití Nix pro sestavení

Definice balíčku v Nix je stavebním kamenem pro další používání Nix v projektu. Už při vývoji se naráží na problém s lokálními soubory. Ve finální definici balíčku, používané CI/CD serverem, je zdrojem souborů vzdálený repozitář. Řešení filtrace souboru `result` ze zdrojových souborů uvedené v předchozí sekci nemusí být dostačující pro simulaci sestavení CI/CD serverem. Proto je dobré, při sestavení z lokálních souborů, odfiltrovat všechny soubory, které nebudou později ve vzdáleném repozitáři. U zdrojových souborů v definici balíčku jsem použil pomocnou funkci pro odfiltrování souborů, které jsou uvedené v souboru `.gitignore` a soubor `result`. Takové zdrojové soubory by měly být totožné s obsahem repozitáře. Mezi použitím lokálních souborů a vzdáleného repozitáře se pak dá snadno přepínat parametrem balíčku. Příklad je ve výpisu 4.2.

```
1  { stdenv, fetchurl, nix-gitignore, localFiles ? true }:
2
3  stdenv.mkDerivation rec {
4
5    src = (
6      if localFiles then
7        nix-gitignore.gitignoreSource [ "result" ] ./.
8      else
9        fetchurl {
10         url = "mirror://gnu/hello/${pname}-${version}.tar.gz";
11         sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
12       }
13   ...
```

Výpis 4.2: Definice zdrojových souborů v derivaci. Soubory mohou být získány stažením ze vzdáleného repozitáře nebo kopií lokálního adresáře. V případě lokálních souborů jsou odfiltrovány soubory nepatřící do repozitáře a soubor `result`.

Důležitým aspektem, který ovlivňuje možnost použití Nix v projektu, je *dodání závislostí*. Ačkoliv má Nix velmi velkou komunitu, neexistují žádné referenční balíčky pro všechny programovací jazyky a sestavovací nástroje. Jediný balíček, který je možné označit za referenční, je GNU hello, který používá Autotools. Některé jazyky a sestavovací nástroje

mají částečnou podporu jako je Python nebo NPM. V takovém případě existují pomocné funkce nebo nástroje, které usnadňují použití Nix pro takovéto projekty. Tyto funkce mají většinou tvar `build*[Module|Package]` a nástroje většinou tvar `*2nix`, kde znak `*` je nahrazen názvem jazyka nebo sestavovacího nástroje. Obecně, podle podpory dané technologie v Nixpkgs, je možné dodat k projektu závislosti třemi způsoby:

1. Závislosti jsou v repozitáři Nixpkgs. Tento způsob je z pohledu Nix nejlepší, ale je dostupný jen pro malý počet technologií. Například pro Haskell je dostupná většina závislostí.

2. Generované derivace závislostí. Seznam závislostí je zapsán strukturovaně v nějakém souboru a pomocí nástroje `*2nix` je možné derivace závislostí vygenerovat. Příkladem je sestavovací nástroj NPM pro jazyk JavaScript a nástroj `node2nix`. Použití příkazu `*2nix` lze automatizovat pomocí dvoustupňové derivace – první derivace spustí nástroj `*2nix` a druhá derivace využije vygenerované derivace z první derivace při sestavení aplikace.

3. Závislosti jako jedna derivace (FOD). Pokud ani jedna z možností uvedených výše nelze použít pro dodání závislostí, musí se závislosti získat externím nástrojem a do definice balíčku pak dodat jako jedna derivace (fixed output derivation). Takový přístup se musí použít například u jazyka PHP a sestavovacího nástroje composer. I když tento přístup funguje obecně pro jakýkoliv projekt, má jeden zásadní problém – musí se manuálně specifikovat hash výstupu derivace pro získání závislostí. Pokud navíc použitý externí nástroj nepodporuje reprodukovatelnost, může se jeho výstup v čase měnit a musí se tedy i opětovně specifikovat hash výstupu na správnou hodnotu.

Pro reprodukovatelnost je zapotřebí zafixovat definice použitých závislostí z repozitáře Nixpkgs uvedením jeho přesné verze. Je možné například vytvořit soubor `nixpkgs.nix`, s obsahem jako ve výpisu 4.3, a ve všech ostatních souborech neimportovat repozitář Nixpkgs ze systému (`import <nixpkgs>`) ale importovat tento soubor (`import ./nixpkgs.nix`).

```
import (
  fetchTarball https://github.com/NixOS/nixpkgs/archive/20.03.tar.gz
) {
  config = { allowUnfree = true; };
  overlays = [
    (self: super: {
        my-hello = super.callPackage ./my-hello.nix;
    })
  ];
}
```

Výpis 4.3: Zafixování verze repozitáře Nixpkgs a jeho nastavení, pro reprodukovatelné sestavení. Součástí může být i modifikace množiny balíčků pomocí overlays.

4.4 Použití Nix pro testování

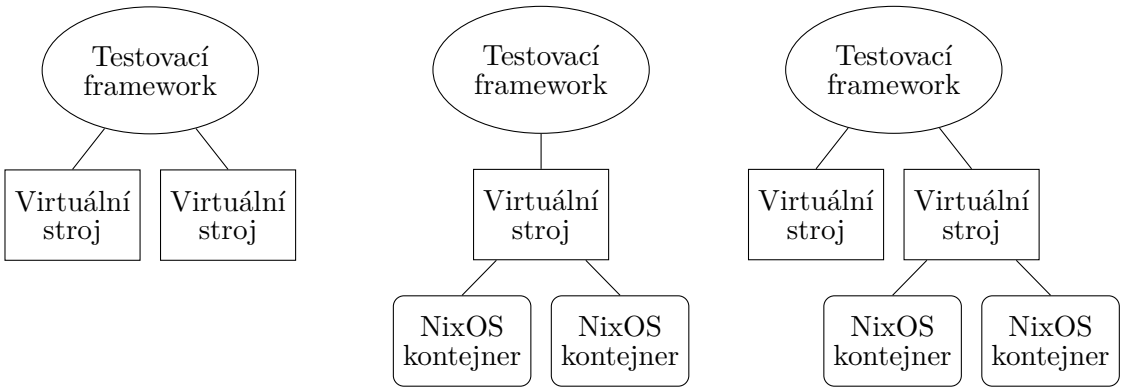
V rámci sestavení aplikace mohou být provedeny jednotkové testy. Po sestavení mohou přijít na řadu testy, které testují jednotlivé komponenty aplikace nebo i aplikaci jako celek. Jsou to například systémové testy, stress testy, coverage testy atd.

Většinu testů by mělo být možné provést posloupností shell příkazů. K tomu je v Nix možné využít derivaci vytvořenou speciální funkcí `runCommand`:

```
runCommand "tests" { nativeBuildInputs = [ app ]; } ''
  mkdir -p $out/test
  echo "Hello world" > expectedOutput
  app > givenOutput
  diff expectedOutput givenOutput > $out/test/result
'';
```

Může se klidně testovat i pomocí nějakého testovacího nástroje, stačí jej uvést jako závislost. Závislosti potřebné pro testování jsou tak oddělené od závislostí aplikace. Je potřeba mít ale na paměti, že testy spuštěné tímto způsobem nemohou být závislé na něčem, co není specifikované jako závislost pro testování. Testovací derivace se musí provést vždy když se změní její vstupy, takže nemůže být specifikována jako FOD a nemá tedy ani přístup k internetu.

Spuštění posloupnosti příkazů ale nemusí být dostačující pro systémové testy. Aplikace se může skládat z více komponent, kde každá komponenta může vyžadovat jiné prostředí. Repozitář Nixpkgs obsahuje malý testovací framework pro spuštění několika virtuálních strojů se systémem NixOS, které jsou navíc propojené mezi sebou virtuální sítí [19]. Protože jsou tyto stroje napojené i na testovací framework, je poté možné spustit testovací skript a testovat tak například dostupnost služeb na těchto systémech. Schéma testování infrastruktury se dvěma stroji je na obrázku 4.1a. Virtuální stroje vytvořené pomocí tohoto frameworku sdílí adresář `/nix/store` s hostitelským systémem, ze kterého rovnou načítají jádro systému bez vytváření diskového obrazu systému, takže jejich spuštění je rychlé a nejsou náročné na diskový prostor [24].



(a) Testování pomocí několika virtuálních strojů.

(b) Testování pomocí jednoho virtuálního stroje s několika NixOS kontejnery.

(c) Kombinace testování pomocí virtuálních strojů a NixOS kontejnerů.

Obrázek 4.1: Různé úrovně systémového testování pomocí NixOS testovacího frameworku.

Spuštění několika virtuálních strojů je ale velmi náročné na systémové prostředky. Druhou variantou je simulovat stroje pomocí NixOS kontejnerů v jednom virtuálním stroji. V takovém případě by se vytvořila testovací infrastruktura zobrazená na obrázku 4.1b. Pokud je testování spouštěno přímo na NixOS, nemuselo by být potřeba ani vytvářet virtuální stroj. NixOS kontejnery podobně jako virtuální stroje sdílí `/nix/store` a jejich vytvoření je také rychlé. Oproti virtuálním strojům ale nemají dedikované systémové prostředky, takže jejich použití může být efektivnější.

V případě potřeby se mohou předcházející přístupy kombinovat, jak je vidět na obrázku 4.1c. Odpovídající zápis pro testovací framework, který vytvoří testovací infrastrukturu na obrázku 4.1c, je ve výpisu 4.4

```
import ./make-test-python.nix {
  nodes = {
    client = { config, pkgs, ... }: {
      # NixOS konfigurace
    };
    webserver = { config, pkgs, ... }: {
        containers.apache = {
          autoStart = true;
          config = { config, pkgs, ... }: {
            # NixOS konfigurace
          };
        };
        containers.proxy = {
          autoStart = true;
          config = { config, pkgs, ... }: {
            # NixOS konfigurace
          };
        };
    };
  };
  testScript = ''
    start_all()
    webserver.wait_for_unit("containers@proxy.service")
    webserver.wait_for_unit("containers@apache.service")
    client.wait_for_unit("default.target")
    client.succeed("wget https://webserver")
  '';
}
```

Výpis 4.4: Testování pomocí NixOS testovacího frameworku. Testovací infrastruktura sestává z virtuálního stroje `client` a virtuálního stroje `webserver` obsahující dva NixOS kontejnery `apache` a `proxy`. Nad stroji bude proveden testovací skript, který vyčká na spuštění služeb a poté příkazem `wget` ověří dostupnost webového serveru na stroji `webserver`.

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.

Mít ale další stejnou produkční infrastrukturu pro testování může být velmi nákladné. Je proto možné použít kompromisní řešení: nasadit část infrastruktury na jeden systém NixOS 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. Protože je vše reprodukovatelné (aplikace, prostředí, infrastruktura), je tímto způsobem možné otestovat nasazení aplikace ještě před samotným nasazením.

4.5 Použití Nix pro release

Ve fázi release se vytváří různé výstupy, které mohou být využity k distribuci aplikace k zákazníkům nebo pro nasazení na server. Nix může v tomto směru ulehčit mnoho práce. Za prvé může být sestavení 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 nebo .rpm. Vytvoření těchto instalačních souborů není snadné, protože závislosti v Nix fungují úplně jiným způsobem. Například sestavení .deb balíčku způsobí spuštění linuxové distribuce Debian v QEMU a uvnitř se teprve sestaví .deb balíček.

Díky podpoře cross-compilation je také velmi snadné sestavovat aplikaci pro jiné platformy. Pro použití aplikace v kontejnerech je možné jednoduše vygenerovat Docker image nebo OCI kontejner s aplikací. Příklad je ve výpisu 4.5. Kromě kontejnerů je možné generovat i celý operační systém NixOS. Stačí vyhodnotit speciálně upravenou konfiguraci systému a vytvoří se bootovatelné ISO nebo obraz virtuálního stroje. Repozitář nixos-generators[3] obsahuje kolekci všech generovatelných formátů systému NixOS.

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

  build = pkgs.app;

  buildRaspberryPi = pkgsCross.raspberryPi.app;

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

Výpis 4.5: Příklad sestavení pro RaspberryPi pomocí cross-compilation a vytvoření Docker image s aplikací.

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

[3] nixos-generators – https://github.com/nix-community/nixos-generators

4.6 Použití Nix pro nasazení

Díky kompletně specifikovaným závislostem je nasazení software pomocí Nix velmi jednoduché. Stačí na cílový stroj dodat closure balíčku. Nix dokonce nabízí samostatný příkaz pro kopii closure:

```
$ nix-copy-closure --to alice@itchy.example.org $(type -p firefox)
```

Nebo je možné exportovat closure do souboru a pak tento soubor importovat [17]:

```
$ nix-store --export $(nix-store -qR $(type -p firefox)) > closure
$ nix-store --import < closure
```

Pokud se pro nasazení použije closure výstupu derivace, je to podobné jako binární distribuce balíčku. Při použití closure `.drv` souboru se jedná o source code distribuci balíčku. V druhém případě se jako další krok musí na cílovém stroji ještě balíček sestavit pomocí příkazu `nix-build`.

Lepší variantou pro nasazení, než je použití closure, je použití konfigurace NixOS. Společně s nasazením aplikace může být tak i upravena konfigurace 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 aktivovat ji.

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. Jak bylo zmíněno v sekci 4.1, není zatím koncepčně vyřešená manipulace a uložení hesel a ostatních klíčů v konfiguračních souborech NixOS. Pokud se pro konfiguraci strojů použije NixOps, dá se tento problém vyřešit pomocí dočasného souborového systému vytvořeného při nasazení (atribut `deployment.keys`). Namísto NixOps je možné společně s NixOS samozřejmě použít i jiné nástroje, například Terraform. Nicméně popis infrastruktury pak není ve stejném formátu jako popis konfigurace strojů a balíčků.

Nix nestanovuje postupy nebo pravidla pro jeho použití. Možností je několik a je na uživateli jakým způsobem bude Nix pro své potřeby používat. V této kapitole byly navrženy možnosti řešení pro jednotlivé fáze CI/CD. Výsledky implementace tohoto návrhu a jejich zhodnocení prezentuje následující kapitola.

Kapitola 5

Implementace sady příkladů

V předchozí kapitole bylo navrženo použití Nix pro průběžnou integraci a nasazení software. Následující sekce shrnují výsledky implementace tohoto návrhu. Byla vytvořena sada příkladů demonstrující komplexní použití Nix tak, aby bylo možné je jednoduše použít a integrovat do existujících řešení CI/CD. Příklady mají za cíl být co nejjednodušší, a přitom demonstrovat nejdůležitější funkce Nix. Pro vývojáře by neměl být problém najít v příkladech svou doménu zájmu a příklad snadno modifikovat a adaptovat na svůj projekt.

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

5.1 Dostupné technologie a druhy aplikací v příkladech

Celkem jsem vytvořil 15 různých příkladů, které pokrývají 7 programovacích jazyků a 9 různých sestavovacích nástrojů, jmenovitě se jedná o následující programovací jazyky a sestavovací nástroje:

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

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. Ale 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 4.3. Přehled dostupných způsobů integrace v sadě příkladů, pro jednotlivé nástroje pro správu závislostí, je v tabulce 5.1.

Příklady pokrývají nejenom různé programovací jazyky a nástroje pro sestavení, ale také různé druhy aplikací. Součástí příkladů je například webová aplikace ve frameworku Laravel, distribuovaná aplikace pomocí Apache Spark, vícevrstvá aplikace vytvořená v ja-

Nástroj pro správu závislostí	**Závislosti v Nixpkgs**	**Generované derivace závislostí**	**Závislosti jako jedna derivace (FOD)**
Autotools	✓		
Go modules			✓
Cabal	✓	✓	
Maven			✓
Gradle			✓
NPM		✓	✓
Composer			✓
Pip	✓		

Tabulka 5.1: Dostupné příklady úrovně integrace s Nix pro jednotlivé nástroje pro správu závislostí.

zyku Python a mobilní aplikace pro platformu Android. Kompilované aplikace obsahují i možnost cross-compile a je tedy možné je použít i ve vestavěných systémech.

V sadě příkladů jsou důležité `.nix` soubory, a ne zdrojové soubory aplikací a jejich implementace. Samotné aplikace v příkladech jsou buď známé open-source projekty, nebo velmi jednoduché aplikace vytvořené jen za účelem demonstrace dané technologie v příkladech. Tyto demonstrativní příklady například pouze vypíší na standardní výstup „Hello world“. Byl ale kladen důraz na to, aby jejich sestavení bylo stejné jako u plnohodnotných aplikací. Každý příklad proto obsahuje alespoň jednu závislost, aby sestavení nebylo triviální.

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. Opakující se kód je však možné vyčlenit do samostatných souborů a vytvořit tak malou pomocnou knihovnu. Kromě zdrojových souborů, obsahuje každý příklad pět až devět `.nix` souborů a volitelně i soubor `cicd.sh` pro integraci s Nix a pro CI/CD:

- `nixpkgs.nix` – zafixování verze a konfigurace Nixpkgs a přidání balíčku aplikace jako overlay. Pokud by se v budoucnu razantně změnila struktura Nixpkgs nebo by URL použitá k zafixování verze Nixpkgs nebyla dostupná, je součástí repozitáře s příklady i záložní kopie repozitáře Nixpkgs.
- `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` – výběr balíčku aplikace z `nixpkgs.nix`.
- `shell.nix` – podobný soubor jako `default.nix`, který ale mění atributy výsledného balíčku, jako je například atribut `src` a přidává nástroje potřebné pouze pro vývoj.
- `ci.nix` – sada dostupných akcí pro CI. Obsahuje i demonstraci všech možností Nix pro danou technologii. Například různé druhy testování (posloupnost příkazů, virtuální stroje, NixOS kontejnery, ...) nebo různé druhy výstupů (instalační soubory, cross-compilation, generování systému NixOS, ...) atd.

- `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 (nezávislý na NixOps).
- `cd-*.nix` – fyzický popis infrastruktury pro NixOps. 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.
- `cicd.sh` – spuštění CI pipeline a nasazení pomocí NixOps.

Použité soubory nebo jejich význam může být samozřejmě upraven dle potřeb. Například pro desktopovou aplikaci nejsou zapotřebí soubory pro nasazení aplikace. Pokud se používá více infrastruktur zároveň (testovací, staging, produkční, ...), mohou být vytvořeny další odpovídající `.nix` soubory nebo přidán argument do souboru `cd.nix`.

Zajímavým důsledkem použití programovacího jazyka pro popis infrastruktury je vysoká flexibilita. Kupříkladu infrastruktura pro distribuovanou aplikaci, lze jednoduše škálovat jedním parametrem funkce a tvořit tak infrastrukturu variabilně. U webové aplikace je zase možné parametry ovládat rozdělení aplikace na různé stroje. V produkčním prostředí může být pro webovou aplikaci využit webový server, databázový server nebo i souborový server. Pro testovací účely je ale možné jedním parametrem upravit konfiguraci tak, aby se vše nasadilo na jeden nebo dva samostatné servery.

Ty aplikace, které se nasazují na server jako služby (webové, distribuované, vícevrstvé), jsou vytvořeny dle doporučení *Twelve Factor App* [26] pro vytváření cloudových aplikací. Následující výčet shrnuje, jak je jednotlivých bodů těchto doporučení v příkladech dosaženo:

1. Zdrojový kód – existuje jenom jeden repozitář aplikace a několik různých instancí nasazení aplikace (produkční, testovací, lokální, ...) odvozené od souboru `cd.nix`.
2. Závislosti – kompletně a explicitně definované závislosti jsou zajištěny použitím balíčkovacího systému Nix.
3. Konfigurace – konfigurace je oddělená od zdrojových kódu (je generovaná v modulu). Klíče a hesla jsou uložená v dočasném souborovém systému a aplikaci jsou předána pomocí proměnných prostředí.
4. Podpůrné služby – služby na jednotlivých strojích jsou referencovány pomocí URL a mohou být tak jednoduše vyměněny za jiné.
5. Sestavení, vydání, spuštění – Nix store je read-only, takže nasazenou aplikaci nelze upravovat. Každá verze či varianta aplikace má unikátní hash a lze se jednoduše vrátit k předchozí verzi.
6. Procesy – služby jsou primárně bezstavové a jakýkoliv stav je ukládán do podpůrných služeb. Například u webové aplikace to je souborový server nebo databázový server.
7. Vazba s portem – každá aplikace je plně samostatná a v modulu definuje vše, co potřebuje k běhu.
8. Souběh – procesy a služby jsou spravovány správcem procesů (systemd). Výpočetní uzly u distribuované aplikace mohou být spuštěny a škálovány lokálně jako samostatné procesy, nebo mohou být rozděleny na různé stroje.

9. Zahoditelnost – tento bod není řešen pomocí Nix ani návrhem příkladů, měl by být řešen použitím vhodného frameworku pro danou doménu, či správným vývojem aplikace.

10. Podobnost Vývoj/Produkce – díky použití Nix a NixOS je prostředí reprodukovatelné a není tak, až na hardwarové rozdíly a konfiguraci, žádný rozdíl mezí vývojovým a produkčním prostředí.

11. Logy – všechny výstupy služeb jsou centralizovány správcem procesů (systemd), odkud mohou být odesílány do jiných úložišť nebo služeb.

12. Admin procesy – jednorázové procesy jsou spouštěny ve stejném prostředí a stejným způsobem jako procesy aplikace (pomocí správce procesů). Například migrační skript databáze u webové aplikace, je spuštěn jednorázově po spuštění aplikace.

5.2 Integrace CI/CD systémů s Nix

Jak bylo zmíněno výše, je u každého příkladu soubor `ci.nix`, který definuje dostupné akce pro CI server. Tento soubor má jednotné rozhraní u všech druhů aplikací, od mobilní aplikace pro Android přes webovou aplikaci napsanou v PHP až po desktopovou aplikaci v jazyku C. Soubor `ci.nix` u každého projektu definuje dostupné akce v množině atributů. Pro vykonání dané akce, například akce `build`, stačí spustit příkaz:

```
$ nix-build ci.nix -A build
```

CI systém může fungovat tak, že bude jenom provádět všechny dostupné akce v souboru `ci.nix`. Ostatně CI systémy s podporou Nix (Hydra, micro-ci, Scylla) přesně takto fungují. Populární nástroje pro CI/CD představené v sekci 2.4 jsou víceúčelové a můžou fungovat stejně. Stačí, aby měly dostupný příkaz `nix-build` a úložiště objektů `/nix/store`. K tomu může být použit například Docker kontejner s instalací Nix.

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 vytvoření instalačních souborů může být provedeno až po úspěšném vykonání všech testů. Příklad takové definice je ve výpisu 5.1a a odpovídající vytvořené závislosti a výstup fází je na obrázku 5.1. V samotném CI systému se už pak vůbec nemusí specifikovat fáze integrace a nasazení a spustí se už jenom akce `pipeline` a uloží její výstup, který je na obrázku 5.1b.

Pokud je to potřeba, je možné atribut `pipeline` definovat tak, aby obsahoval pouze nejdůležitější akce a dlouho trvající testy nebo instalační soubory pak mohou být v jiném atributu, například `long-tests`. Potom se v CI systému nastaví, aby se atribut `pipeline` vykonal při každé aktualizaci repozitáře a atribut `long-tests` třeba jenom jednou za hodinu neboli two-staged build.

Z obrázku 5.1a je patrné, že poslední fáze `phase-release` má závislost na všech předcházejících fázích a tranzitivně i na všech akcích. Oproti tomu první fáze `phase-build` není závislá na žádné předcházející fázi. Pomocná funkce `mkPipelinePrev` ve výpisu 5.1b umožňuje přiřadit závislosti i této první fázi. Pomocí této funkce tak může být snadno vytvořena integration pipeline. Stačí jako závislosti první fáze pipeline uvést poslední fáze jiných pipeline.

```
pipeline = mkPipeline [
  (phase "build" [
    build
  ])
  (phase "test" [
    test1
    test2
  ])
  (phase "release" [
    debPackage
    rpmPackage
  ])
];
```

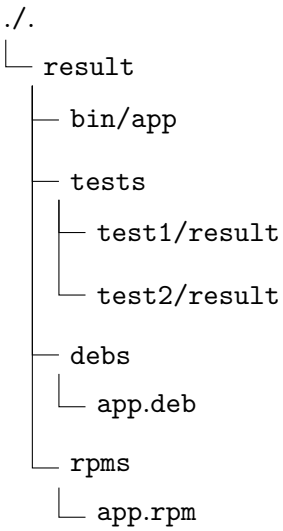
(a) Definice pipeline v souboru `ci.nix`.

```
mkPipeline = mkPipelinePrev null;
mkPipelinePrev = prev: phases:
  foldl mkDependency prev phases;

mkDependency = prev: next:
  next.overrideAttrs (
    oldAttrs: { prev = prev; }
  );
phase = phaseName: jobs:
  pkgs.symlinkJoin {
    name = "phase-${phaseName}";
    paths = [ jobs ];
  };
```

(b) Pomocné funkce pro vytvoření CI pipeline.

Výpis 5.1: Ukázka definice CI pipeline pomocí Nix. Z pohledu Nix je fáze nebo celá pipeline jenom další derivace (balíček). Akce v každé fázi mohou být vykonány paralelně a jednotlivé fáze jsou pak vykonány vždy jedině sekvenčně.

(a) Závislosti fází v `/nix/store` odpovídající výpisu 5.1a. Každá fáze má závislost na předcházející (červená šipka).



(b) Výstup pipeline, který byl vytvořen sloučením výstupů všech fází (zde neuvedený implementační detail).

Obrázek 5.1: Pipeline v `/nix/store` a její výstup v aktuálním adresáři.

Pokud se aplikace skládá z velkého počtu komponent, mohou být závislosti mezi několika pipeline složité. Jedna komponenta může záviset na 0 až N jiných komponentách a mohou vznikat i cyklické závislosti. To ale není potřeba nějak řešit pomocí CI systému, protože se na to použije celý aparát, který má Nix k dispozici pro vyhodnocování závislostí. Všechny potřebné závislosti (komponenty) se sestaví, ty, co se nezměnily, tak se nebudou sestavovat znovu a vše, co půjde, se bude sestavovat paralelně.

Součástí CI/CD pipeline může být i fáze nasazení. Tato fáze je ale na rozdíl od ostatních fází stavová. Při nasazení může být pozměněna i infrastruktura a k tomu je zapotřebí znát její aktuální stav. Z tohoto důvodu není nasazení součástí pipeline v souboru `ci.nix` a neprovádí se pomocí Nix. K nasazení slouží skript `cicd.sh`, který nejprve vykoná atribut `pipeline` v souboru `ci.nix` příkazem uvedeným výše a poté pomocí NixOps provede nasazení. V CI/CD systému tak stačí jenom spustit skript `cicd.sh`.

Integrace procesu CI definovaném v souboru `ci.nix` s nástroji Hydra, micro-ci nebo Scylla by neměl být problém, protože soubor `ci.nix` je s nimi kompatibilní. Jen je nutné mít na paměti, že tyto nástroje vykonávají všechny atributy v souboru, a to nemusí být vždy žádoucí. V případě potřeby mohou být tak některé atributy „schovány“ konstruktem `let ... in`. V následujících odstavcích je krátce shrnuto použití v klasických CI/CD systémech, které byly zmíněné v sekci 2.4. Konfigurační soubory pro tyto CI/CD systémy je možné nalézt v repozitáři s příklady.

Jenkins. V CI/CD systému Jenkins se musí prostředí nastavit manuálně. V případě Nix je potřeba jej nainstalovat jedním příkazem a poté vše funguje dle očekávání. Adresář `/nix/store` funguje jako cache, takže se vždy sestavuje jenom to, co se změnilo. Jediný problém může nastat při provozování Jenkins v Docker kontejneru. Oficiální Jenkins Docker image totiž neobsahuje podporu pro virtualizaci uvnitř kontejneru, a není pak tedy možné v rámci pipeline testovat pomocí NixOS testovacího frameworku.

CircleCI. Při použití CircleCI probíhá proces CI/CD ve virtuálním stroji nebo v Docker kontejneru. Nix lze nainstalovat jedním příkazem, nebo je možné použít oficiální Nix Docker image. Pro každé sestavení je vytvořen nový virtuální stroj nebo Docker kontejner, takže adresář `/nix/store` není zachován mezi sestaveními a po každém spuštění procesu CI/CD se musí spousta derivací znovu stáhnout z cache. Dalším problémem je, že CircleCI nepodporuje vnořené virtuální stroje, které jsou potřeba pro testování pomocí NixOS testovacího.

Travis CI. Služba Travis CI má přímou podporu pro Nix a není tedy potřeba řešit jeho instalaci. Proces CI/CD ale probíhá ve virtuálním stroji, takže stejně jako u CircleCI není adresář `/nix/store` zachován mezi sestaveními. Stejně tak i Travis CI nepodporuje vnořené virtuální stroje, takže testovaní pomocí NixOS testovacího frameworku není možné.

Gitlab CI/CD. Při využití infrastruktury Gitlab CI/CD probíhá proces CI/CD v Docker kontejneru. Podobně jako u CircleCI stačí použít oficiální Nix Docker image. Nastává pak bohužel i úplně stejný problém jako u CircleCI a Travis CI s adresářem `/nix/store` a vytvářením virtuálních strojů. Na rozdíl od nich je ale možné zprovoznit vlastní agenty a tyto problémy tak odstranit.

5.3 Zveřejnění příkladů jako open-source a zhodnocení

V sekci 4.1 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. Repozitář s příklady je dostupný jako open-source pod

licencí GPL na autorově účtu u služby GitHub[1]. Součástí repozitáře s příklady je i obecná šablona pro nové projekty s příkladem konfigurace CI/CD systémů.

Zhodnocení použití Nix pro vývoj

Definování projektu jako Nix balíček přispívá k dobrým programátorským praktikám. Proces sestavení je uložen na jednom místě v popisu derivace, kde jsou zároveň zmapovány všechny jeho závislosti a zdokumentovány všechny proměnné prostředí potřebné pro sestavení projektu. Při zadefinování modulu je i vytvořeno jakési rozhraní pro konfiguraci a běh aplikace.

I když je křivka učení Nix strmá, může být ve výsledku kratší než při použití jiných technologií. Vývojář může pro všechny projekty použít jenom jeden sestavovací nástroj (Nix), který abstrahuje různé způsoby dodávání závislostí a sestavení pro různé projekty. Pro nového člena týmu tak může být příprava prostředí pro vývoj mnohem jednodušší. Při vývoji webové aplikace nemusí například řešit instalaci a nastavení technologií Node.js, PHP nebo Java a ani instalaci sestavovacích nástrojů jako je NPM, Composer nebo Gradle. Pokud se v projektu používá Nix/NixOps, tak je sestavení balíčku, konfigurace systému a popis infrastruktury v jednom jazyku. Jsou tak nahrazeny různé technologie, jejichž ovládání a popis konfigurace (DSL) se musí vývojář naučit. Například struktura Terraform konfigurace pro popis infrastruktury, zápis Ansible příkazů pro konfiguraci strojů, zápis Dockerfile pro vytvoření prostředí a formát sestavovacího nástroje pro popis aplikace.

Pro vytvoření jednotného prostředí mezi vývojáři se často používají virtuální stroje (například Vagrant) nebo kontejnery (například Docker). Oproti použití virtuálních strojů pro vývoj je Nix samozřejmě rychlejší a používá méně paměti a prostoru na disku. Srovnání kontejnerů a Nix, z pohledu výkonnosti při vývoji, je srovnatelný. Oproti kontejnerům je ale Nix úspornější, co se diskového místa týče a prostředí je reprodukovatelné oproti použití například Dockerfile. Pokud se navíc Nix používá na NixOS, tak je úspora ještě výraznější, protože některé závislosti potřebné pro vývoj mohou být už v systému dostupné a sdílí se tak se systémem. Použití Nix může být výhodnější i kvůli jednoduššímu ladění aplikace. Není potřeba složitě získávat stav systému, log aplikace nebo soubory z izolovaného souborového systému, jako je tomu v případě kontejnerů. Na druhou stranu někdy je výhodné aplikaci, například kvůli experimentování, při vývoji izolovat. Ale i v takovém případě může být Nix snadno použit v kombinaci s kontejnery nebo virtuálními stroji. [16]

Zhodnocení použití Nix pro sestavení

Pokud vývojář úspěšně sestaví svůj projekt na své stanici se zapnutým sandboxem, může si být jistý, že projekt bude úspěšně sestaven i na CI serveru. Vývojář tak může integrovat své změny velmi často, a to přispívá k agilnímu vývoji. Bez jistoty úspěšného sestavení by se mohl bát, že bude muset řešit kde je problém, aby nenechal na CI serveru neúspěšné sestavení, které by mohlo brzdit ostatní členy týmu ve vývoji. Díky Nix tak může vložit své změny do repozitáře bez stresu i před koncem pracovní doby.

Pro snadné používání Nix pro průběžnou integraci a nasazení je potřeba, aby získání závislostí bylo na první (závislosti v Nixpkgs) nebo druhé úrovni (generované derivace závislostí). Zatím ale tomu tak vždy není, jak je možné vidět v tabulce 5.1 a závislosti musí být dodány externím nástrojem (třetí úroveň – závislosti jako jedna derivace). Sdílení závislostí mezi různými projekty pak není možné a externí nástroj ani nemusí podporovat reprodu-

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

kovatelnost. Nicméně příklady se kontinuálně sestavují a za uplynulý měsíc nenastal ani jednou problém s nereprodukovatelností, takže i třetí úroveň může být v mnoha případech dostačující.

Někdy může být integrace s Nix velmi složitá. Sada příkladů pokrývá 7 programovacích jazyků, 9 různých sestavovacích nástrojů a několik druhů aplikací. Se všemi technologiemi v příkladech se podařilo integraci s Nix provést. Nicméně nejedná se o precedens a pro projekty používající jiné technologie mohou nastat problémy se získáním závislostí a použití Nix pak nemusí být pro takovéto projekty tak zajímavé. Většina populárních programovacích jazyků a sestavovacích nástrojů ale dnes už umožňuje získat závislosti reprodukovatelně a jednalo by se tedy spíše o výjimku. Do budoucna by bylo určitě dobré sadu příkladů ještě rozšířit.

Zhodnocení použití Nix pro testování

Základní testování v podobě spuštění posloupnosti shell příkazů je možné provést v jakémkoliv CI/CD systému. Zajímavým přínosem Nix je posunutí systémových a akceptačních testů na jinou úroveň. Je možné testovat konfiguraci systému nebo dokonce celou infrastrukturu automaticky. V Nixpkgs k tomu slouží jednoduchý testovací framework, který umožňuje spustit několik virtuálních strojů a spustit nad nimi posloupnost příkazů. Pro rychlejší testování a úsporu systémových prostředků lze spustit jenom jeden virtuální stroj a stroje simulovat pomocí NixOS kontejnerů nebo tyto přístupy kombinovat. Spouštění virtuálních strojů nebo NixOS kontejnerů je pomocí tohoto frameworku velmi efektivní, protože se mezi hostitelským systémem, virtuálními stroji a kontejnery sdílí adresář `/nix/store`.

Testování pomocí NixOS testovacího frameworku má ale jednu nevýhodu. I když Nix může být provozován na jakémkoliv Linuxovém systému nebo na macOS, systémové testování pomocí tohoto testovacího frameworku může být prováděno jen na jedné platformě – na systému NixOS. Na jiných platformách nemůže být Nix použit k deklarativní správě konfigurace systému. [24]

Použití Nix pro testování přispívá k větší důvěře k otestovanosti aplikace. Sestavení balíčku je reprodukovatelné, takže to, co se testuje, je to samé jako to, co se nasadí na produkční server. Reprodukovatelné je i prostředí a aplikace tak může být testována ve stejném prostředí jako je prostředí produkčního serveru. Pomocí NixOps může být reprodukovatelně vytvořena i testovací infrastruktura, která je kopií produkční infrastruktury. Použití Nix tak odstraňuje rozdíly mezi testovacím a produkčním prostředím. Jediné, co se mezi nimi může lišit, je stav a hardwarová konfigurace. Spolehlivě a reprodukovatelně sestavit celou infrastrukturu, prostředí a balíčky umí jenom Nix/NixOps, jiné nástroje to nedokážou.

Zhodnocení použití Nix pro release

Vytváření různých variant balíčku je velmi snadné, protože balíček je funkce. Vytváření instalačních souborů .rpm, respektive .deb ale už tak jednoduché není. Musí se spustit virtuální stroj se systémem Debian, respektive Fedora a spustit sestavení balíčku uvnitř. Nejenže je to pomalé ale i náročné na diskový prostor, protože obraz systému musí být uložen v `/nix/store`.

Pro sestavení aplikace na různé platformy se často používají v CI/CD systémech agenti, kteří provedou sestavení na dané platformě a vrátí výsledek. V Nix se dá docílit stejného výsledku pomocí cross-compile. Každá definice balíčku obsahuje rekurzivně i definice všech závislostí a pro provedení cross-compile stačí změnit jen jeden atribut. Nicméně ne všechny

aplikace dostupné v Nixpkgs jsou vytvořené tak, aby je bylo možné zkompilovat na jinou platformu, než na kterou cílí, takže ne vždy cross-compile funguje správně.

K nasazení aplikace se často používají kontejnery. Opět, díky kompletně specifikovaným závislostem, může být pomocí Nix vytvořena Docker image nebo OCI kontejner, který obsahuje jenom aplikaci a run-time závislosti. Docker image tak může být násobně menší než při použití Apline Linux [14]. Pro někoho může být také zajímavé generování ISO obrazu nebo jiných formátů systému. Lze tak vytvořit live distribuci s již předinstalovaným software a nastaveným systémem, nebo vytvořit obraz virtuálního stroje s nainstalovanou aplikací, který se pak jen nahraje ke cloudovému poskytovateli.

Zhodnocení použití Nix pro nasazení

Díky vlastnostem Nix je nasazení software velmi spolehlivé. Závislosti nasazované aplikace jsou přesně definované a mohou být nasazené společně s aplikací. Při instalaci aplikace (kopírování closure do `/nix/store`) se neovlivňuje stav systému. Balíčky mohou být v systému dostupné v různých verzích zároveň a správná verze se pak jenom aktivuje.

Aktivace balíčku je ale provedena změnou symlinku, takže balíček nemůže například spustit migraci dat nebo konfigurace (post install/update script). Nicméně konfigurace může být specifikována také jako závislost a migrační skript pro konfiguraci pak není potřeba. Problém s migrací dat se dá obecně vyřešit obalujícím skriptem, který před spuštěním aplikace zkontroluje, zdali se má nebo nemá spustit migrační skript. Případně i samotná aplikace může být vytvořena tak, aby při každém spuštění kontrolovala, v jaké verzi má k dispozici data a v jaké verzi je spuštěna.

Pomocí deklarativní konfigurace systému NixOS se dá zajistit reprodukovatelné prostředí pro aplikaci. To je zcela zásadní pro důvěru k úspěšnému nasazení a správnému fungování aplikace. Pokud byla stejná konfigurace použita i pro systémové testování, nebyl rozdíl mezi testovacím prostředím a produkčním prostředím. Problémy při nasazení způsobené jiným prostředím tak jsou plně eliminovány. Nasazení aplikace na stroj se systémem NixOS je velmi snadné. Stačí povolit modul aplikace, který obsahuje veškerou konfiguraci systému, která je potřeba pro danou aplikaci.

Pro nasazení aplikace 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.

Nástroj NixOps proces nasazení na stroje se systémem NixOS značně zjednodušuje. Celá infrastruktura je popsaná deklarativně a flexibilně. Infrastruktura je funkce a jedním parametrem se může ovlivnit například počet strojů. NixOps má podporu nejznámějších cloudových poskytovatelů a nasazením do virtuálních strojů umožňuje i otestovat nasazení celé infrastruktury. Je tak možné otestovat úspěšnost nasazení aplikace, ještě předtím, než bude nasazení provedeno CI/CD serverem na produkční infrastruktuře.

Ať už se pro nasazení aplikace použije closure nebo NixOS modul, není nasazení destruktivní. Nic se na cílovém stroji nepřepisuje a tím pádem se dá vždy snadno a bezpečně vrátit ke starší funkční verzi. Pomocí NixOps se dají vrátit uceleně i změny provedené na všech strojích v infrastruktuře. Velkou výhodou Nixops oproti jiným nástrojům pro IaC

je, že se při rollbacku změny infrastruktury, zároveň provede i změna konfigurace systému a aktivuje se odpovídající verze aplikace se správnou verzí její konfigurace. Změna celé infrastruktury nebo konfigurace systému není sice atomická, ale vše se provede najednou a jedním příkazem. To, jaká konfigurace aplikace bude aktivována je závislé na verzi aplikace ale i na prostředí, kde běží (testovací server, produkční server atd.). U NixOps je vše zpracováno a vyhodnoceno přímo ze zdrojových souborů, takže není potřeba udržovat obsáhlou dokumentaci.

Funkční softwarovou aplikaci lze rozložit na čtyři komponenty: spustitelný kód, konfiguraci, hostitelské prostředí a data [15]. První tři komponenty lze při nasazení spravovat pomocí Nix, NixOS a NixOps. Data nebo stav aplikace ale nelze při nasazení definovat deklarativně, a proto je nelze jednoduše spravovat pomocí Nix. Pokud by bylo třeba používat Nix i pro správu dat, bylo by zapotřebí vždy data zazálohovat a po nasazení zase obnovit. Takto funguje nástroj Dysnomia [23], který je úzce spjatý s nástrojem DisnixOS [25].

Podobně lze narazit na problém počátečního nastavení stavu aplikace. I když jsou počáteční data většinou statická a dají se například do databáze vložit jednoduchým skriptem, musí to být provedeno pouze jednou. V takovém případě je potřeba někde zaznamenat, že již došlo k provedení tohoto skriptu a při dalších nasazeních aplikace ho už nevykonávat. Druhou možností je při prvním nasazení aplikace spustit i nasazení počátečního stavu a při dalších nasazeních aplikace už počáteční stav nenasazovat. K tomu může sloužit parametr infrastruktury, který bude fungovat jako přepínač pro nasazení počátečního stavu.

Zhodnocení integrace Nix s klasickými CI/CD systémy

Nix je možné používat společně s kterýmkoliv CI/CD systémem zmíněném v sekci 2.4. Díky vlastnostem Nix si může vývojář zreprodukovat jakoukoliv akci prováděnou CI/CD serverem lokálně. Po provedených změnách tak může předem ověřit, zda sestavení na CI/CD serveru bude úspěšné. V případě problému nějaké akce provedené na CI/CD serveru, může vývojář tuto akci lokálně ladit. Nepotřebuje SSH přístup na CI/CD server nebo Docker kontejner.

Další výhoda definování balíčků a jednotlivých fází CI/CD pomocí Nix je nezávislost na použitém CI/CD systému. Každé řešení představené v sekci 2.4 má svůj vlastní formát specifikace fází, a tudíž není snadné přesunout proces průběžné integrace a nasazení z jednoho řešení na jiné. Díky zápisu v Nix je to ale jen o zpřístupnění Nix v daném systému a pak vykonání jednoho příkazu. Jak bylo ukázáno v sekci 4.4, je možné pomocí Nix spouštět i shell příkazy. Pipeline aplikace tak může obsahovat kombinaci imperativně a deklarativně definovaných akcí.

Při použití systémů jako je CircleCI, Travis CI nebo Gitlab CI/CD, které používají pro vykonání pipeline virtuální stroje nebo jiným způsobem zajišťují vždy čisté prostředí, je potřeba počítat s režií v podobě přípravy standardního prostředí. Mezi sestaveními totiž není dostupný adresář `/nix/store` a po každém spuštění procesu CI/CD se derivace dostupné ve veřejné binární cache musí znovu stáhnout a ty, které v cache dostupné nejsou, se musí znovu sestavit. Jako příklad je možné uvést sestavení balíčku GNU Hello, který je závislý pouze na standardním prostředí a u kterého se nic nesestavuje, protože je vše dostupné ve veřejné binární cache. V průměru sestavení, na výše zmíněných CI/CD systémech trvalo 44 sekund, jak je vidět v tabulce 5.2. Jedná se o minimální čas, po kterém započne provádění pipeline.

Pro jiné projekty může být příprava prostředí mnohem složitější. S přičtením času potřebného pro vykonání testů, může být pak odezva od CI/CD velmi dlouhá. Záleží na typu

Projekt	**Jenkins**	**CircleCI**	**Travis CI**	**Gitlab CI/CD**
GNU Hello	N/A	0:32	0:45	0:55
příklad Laravel aplikace	N/A	4:56	3:36	5:43

Tabulka 5.2: Doba sestavení v klasických CI/CD systémech při použití jako Saas. Jedná se o průměrný čas ze tří stejných nezávislých sestavení. U CI/CD systému Jenkins je doba závislá na stroji, kde sestavení probíhá a na stavu adresáře `/nix/store` a nedá se tak porovnat s ostatními.

projektu a na době případného sestavování závislostí. Například pouhé sestavení (bez testování a dalších akcí) webové aplikace ze sady příkladů, kde se musí dodat závislosti pomocí NPM a Composer, může trvat i více jak 5 minut, jak je vidět v tabulce 5.2. To může být problém například u agilní metodiky XP, u které je potřeba mít zpětnou vazbu od CI/CD serveru co nejdříve (maximálně do 10 minut [22]).

V rámci firmy je tento problém možné zmírnit ukládáním výsledků sestavení do privátní binární cache. Již sestavené derivace se tak nebudou muset sestavovat znovu a jenom se stáhnou z cache. Pro rychlejší odezvu může být pipeline také rozdělena a vytvořen tak two-staged build. Z důvodu nemožnosti systémového testování pomocí NixOS testovacího frameworku a neefektivnosti při sestavování pomocí Nix je ale vhodné použít Jenkins nebo Gitlab CI/CD s vlastními agenty namísto CircleCI a Travis CI.

U CI/CD systémů, kde je perzistentní adresář `/nix/store` mezi sestaveními, je potřeba provést přípravu prostředí pro vykonání pipeline pouze při prvním spuštění procesu CI/CD. Po každém dalším spuštění se již vykonané derivace znovu provádět nebo stahovat z veřejné binární cache nebudou, dokud se nezmění jejich vstupy. Na druhou stranu, pokud se v repozitáři provádí velké množství změn, tak se budou na CI/CD serveru do `/nix/store` ukládat stále nové a nové objekty. Průběžná integrace pomocí Nix tak postupem času může mít velké nároky na diskový prostor. Je tedy potřeba velmi často spouštět garbage collector, mnohem častěji než na stanici u vývojáře.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat možnosti uplatnění funkcionálního balíčkovacího systému Nix a jeho ekosystému (NixOS, NixOps) pro CI/CD při agilním vývoji. Aby mohl vývojář Nix použít, musí pro svůj projekt vytvořit popis balíčku. Navíc je potřeba, aby si vývojář dal na nějaké věci pozor a vyhnul se častým chybám. Důležitým aspektem je i dovednost začlenit do Nix netypické a neoficiálně podporované postupy sestavování software.

V této práci byly popsány principy balíčkovacího systému Nix a jeho ekosystému a byly představeny silné a slabé stránky těchto technologií. Tento popis může sloužit i jako návod nebo průvodce pro začátečníky, kteří s Nix začínají. Dále bylo navrženo použití Nix v jednotlivých fázích CI/CD při agilním vývoji a vytvořena sada příkladů demonstrující použití a možnosti Nix. Tato sada příkladů pokrývá 7 programovacích jazyků, 9 různých sestavovacích nástrojů a různé druhy aplikací. Pro vývojáře by neměl být problém najít v příkladech svou doménu zájmu a příklad snadno modifikovat a adaptovat na svůj projekt. Příklady byly zveřejněny jako open-source na webové stránce `https://github.com/vlktomas/nix-examples` a představeny komunitě okolo Nix.

Použití Nix při agilním vývoji přináší řadu výhod. Pro vývoj nabízí rychlé a na diskový prostor nenáročné prostředí, sestavení balíčku je reprodukovatelné, k systémovému testování slouží deklarativní popis infrastruktury propojených strojů a pro spolehlivé nasazení jsou zásadní kompletně a přesně specifikované závislosti a možnost jednoduchého rollbacku. Bylo ukázáno, že se dá Nix použít i pro definici CI/CD pipeline a oprostit tak celý proces průběžné integrace a nasazení od závislosti na konkrétním CI/CD systému. Na druhou stranu v některých CI/CD systémech (CircleCI, Travis CI) může použití Nix zvýšit odezvu CI serveru a nemusí být možné využít všechny možnosti testování, které Nix nabízí.

V práci bych chtěl pokračovat a rozšířit předkládanou sadu příkladů o další programovací jazyky, respektive sestavovací nástroje. V některých CI/CD systémech je možné nastavit úložiště, které je persistentní mezi sestaveními. To by mohlo použití Nix s těmito systémy velmi zefektivnit, ale je zapotřebí tyto možnosti ještě prozkoumat. Do budoucna by bylo zajímavé vylepšit nástroj NixOps o možnost nasazování stavu aplikací a o možnost deklarace závislosti mezi službami na různých strojích.

Použití Nix není jednoduché a je potřeba nastudovat velmi mnoho materiálů. Nicméně výhody, které nabízí, nesporně převažují toto počáteční úsilí. U Nix bude potřeba udělat ještě velkou část práce v oficiální podpoře sestavovacích nástrojů, v oblasti cross-compilation, 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.

Literatura

[1] Adams, B. a Mcintosh, S. Modern Release Engineering in a Nutshell – Why Researchers Should Care. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Osaka: IEEE, Březen 2016, sv. 5, s. 78–90. DOI: 10.1109/SANER.2016.108. ISBN 978-1-5090-1855-0.

[2] Bruno, L., Christensen, G. et al. Nix Pills. *NixOS Linux*, 10. března 2020 [cit. 15. 3. 2020]. Dostupné z: https://nixos.org/nixos/nix-pills/.

[3] Dolstra, E. *The purely functional software deployment model*. Utrecht, NL, 2006. Disertační práce. Utrecht University. ISBN 90-393-4130-3.

[4] Dolstra, E. Secure Sharing between Untrusted Users in a Transparent Source/Binary Deployment Model. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York: Association for Computing Machinery, 2005, s. 154–163. ASE '05. DOI: 10.1145/1101908.1101933. ISBN 1-58113-993-4.

[5] Dolstra, E. Purely Functional Configuration Management with Nix and NixOS. *InfoQ: Software Development News, Videos & Books*, 8. června 2014 [cit. 15. 1. 2020]. Dostupné z: https://infoq.com/articles/configuration-management-with-nix/.

[6] Dolstra, E., Jonge, M. de a Visser, E. Nix: A Safe and Policy-Free System for Software Deployment. In: *Proceedings of the 18th USENIX Conference on System Administration*. USA: USENIX Association, Leden 2004, s. 79–92. LISA '04.

[7] Dolstra, E., Löh, A. a Pierron, N. Nixos: A Purely Functional Linux Distribution. *Journal of Functional Programming*. USA: Cambridge University Press. listopad 2010, sv. 20, 5–6, s. 577–615. DOI: 10.1017/S0956796810000195. ISSN 0956-7968.

[8] Dolstra, E., Vermaas, R. a Levy, S. Charon: Declarative Provisioning and Deployment. In: *Proceedings of the 1st International Workshop on Release Engineering*. IEEE, 2013, s. 17–20. RELENG '13. DOI: 10.1109/RELENG.2013.6607691. ISBN 978-1-4673-6441-6.

[9] Dolstra, E. a Visser, E. The Nix Build Farm: A declarative approach to continuous integration. In: Citeseer. *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*. 2008.

[10] Dolstra, E., Visser, E. a Jonge, M. de. Imposing a Memory Management Discipline on Software Deployment. In: *Proceedings of the 26th International*

Conference on Software Engineering. USA: IEEE, 2004, s. 583–592. ICSE '04. DOI: 10.1109/ICSE.2004.1317480. ISBN 978-0-7695-2163-3.

[11] Duvall, P., Matyas, S. a Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1. vyd. Addison-Wesley Professional, 2007. Addison-Wesley Signature Series (Fowler). ISBN 978-0-321-33638-5.

[12] Felter, W., Ferreira, A., Rajamony, R. et al. An updated performance comparison of virtual machines and Linux containers. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Březen 2015, s. 171–172. DOI: 10.1109/ISPASS.2015.7095802. ISBN 978-1-4799-1957-4.

[13] Fowler, M. Continuous Integration. *Martinfowler.com*, 1. května 2006 [cit. 15. 1. 2020]. Dostupné z: https://www.martinfowler.com/articles/continuousIntegration.html.

[14] Hodique, Y. Using nix to build docker images. *Yann Hodique*, 17. dubna 2016 [cit. 8. 5. 2020]. Dostupné z: https://yann.hodique.info/blog/using-nix-to-build-docker-images/.

[15] Humble, J. a Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1. vyd. New Jersey: Addison-Wesley Professional, 2010. Addison-Wesley Signature Series (Fowler). ISBN 978-0-321-60191-9.

[16] Kaaretkoski, M. *Isolating software development environments*. Tampere, FI, 2018. 56 s. Diplomová práce. Tampere University of Technology.

[17] NixOS organization. Nix Package Manager Guide. *NixOS Linux*, 4. září 2019 [cit. 15. 1. 2020]. Dostupné z: https://nixos.org/nix/manual/.

[18] NixOS organization. Nixpkgs Users and Contributors Guide. *NixOS Linux*, 10. září 2019 [cit. 15. 1. 2020]. Dostupné z: https://nixos.org/nixpkgs/manual/.

[19] NixOS organization. NixOS Manual. *NixOS Linux*, 10. září 2019 [cit. 15. 1. 2020]. Dostupné z: https://nixos.org/nixos/manual/.

[20] NixOS organization. NixOps User's Guide. *NixOS Linux*, 17. dubna 2019 [cit. 15. 1. 2020]. Dostupné z: https://nixos.org/nixops/manual/.

[21] NixOS organization. Hydra User's Guide. *NixOS Linux*, 16. dubna 2020 [cit. 28. 4. 2020]. Dostupné z: https://nixos.org/hydra/manual/.

[22] Stellman, A. a Greene, J. *Learning agile: Understanding scrum, XP, lean, and kanban*. 1. vyd. Sebastopol: O'Reilly, 2015. ISBN 978-1-449-33192-4.

[23] van der Burg, S. A Generic Approach for Deploying and Upgrading Mutable Software Components. In: Dig, D. a Wahler, M., ed. *Fourth Workshop on Hot Topics in Software Upgrades (HotSWUp)*. IEEE Computer Society, June 2012, s. 26–30. HotSWUp '12. DOI: 10.1109/HotSWUp.2012.6226613. ISBN 978-1-4673-1764-1.

[24] VAN DER BURG, S. a DOLSTRA, E. Automating System Tests Using Declarative Virtual Machines. In: *21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, November 2010, s. 181–190. ISSRE '10. DOI: 10.1109/ISSRE.2010.34. ISBN 978-0-7695-4255-3.

[25] VAN DER BURG, S. a DOLSTRA, E. Disnix: A Toolset for Distributed Deployment. *Science of Computer Programming (SCP)*. USA: Elsevier. January 2014, sv. 79, s. 52–69. DOI: 10.1016/j.scico.2012.03.006. ISSN 0167-6423.

[26] WIGGINS, A. *The Twelve-Factor App*. 2017 [cit. 28. 4. 2020]. Dostupné z: https://12factor.net.

Příloha A

Obsah paměťového média

Přiložené paměťové médium obsahuje zdrojové soubory a další materiály vytvořené v rámci diplomové práce. Následující struktura uvádí nejdůležitější adresáře a soubory.

- `/`
  - `excel-at-fit` – materiály ke konferenci Excel@FIT
    - `2020-ExcelFIT-NixCICD` – zdrojové soubory článku pro LaTeX
    - `2020-ExcelFIT-NixCICD.pdf` – článek
  - `masters-thesis` – zdrojové soubory této technické zprávy pro LaTeX
  - `nix-examples` – sada příkladů pro Nix[1]
    - `desktop` – příklady pro desktopové aplikace
    - `distributed` – příklady pro distribuované aplikace
    - `mobile` – příklady pro mobilní aplikace
    - `multitier` – příklady pro vícevrstvé aplikace
    - `template` – šablona a návod pro nové příklady
    - `web` – příklady pro webové aplikace
    - `nixpkgs-20.03.tar.gz` – záložní kopie Nixpkgs
    - `README.md` – základní informace a návod na použití
  - `masters-thesis.pdf` – technická zpráva ve formátu PDF
  - `masters-thesis_print.pdf` – technická zpráva ve formátu PDF určena pro tisk

[1] Dostupné online na adrese https://github.com/vlktomas/nix-examples