REIMPLEMENTACE DATABÁZE KONFERENCÍ A PUBLIKACÍ I

Bc. Tomáš Vondra

Diplomová práce
Fakulta informačních technologií
České vysoké učení technické v Praze
Katedra softwarového inženýrství
Studijní program: Informatika
Specializace: Softwarové inženýrství
Vedoucí: doc. Ing. Petr Fišer, Ph.D.
6. května 2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vondra** Jméno: **Tomáš** Osobní číslo: **500579**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Reimplementace databáze konferencí a publikací I

Název diplomové práce anglicky:

Reimplementation of the database of conferences and publications I

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Petr Fišer, Ph.D. katedra číslicového návrhu FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.11.2024** Termín odevzdání diplomové práce: **09.05.2025**

podpis vedoucí(ho) ústavu/katedry

podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Bc. Vondra Tomáš

Podpis studenta

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vondra** Jméno: **Tomáš** Osobní číslo: **500579**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Reimplementace databáze konferencí a publikací I

Název diplomové práce anglicky:

Reimplementation of the database of conferences and publications I

Pokyny pro vypracování:

Přeprogramujte část stávající webové aplikace pro správu publikací a konferencí do jazyka podporovaného ICT FIT, tj. TypeScript. Původním jazykem je PHP (Nette Framework) a JavaScript. Jako databázi použijte PostgreSQL (původní je MySQL). Jedná se o přeprogramování front-endu i back-endu aplikace. Zaměřte se konkrétně na:
- vytvoření a administraci vývojového prostředí DevOps,
- správu publikací (seznam publikací, správa typů publikací, časopisů, autorů),
- správu tematických kategorií publikací,
- správu uživatelů a uživatelských skupin,
- návrh společné části front-endu (hlavní stránky, přehledné zobrazení seznamů konferencí a publikací, vyhledávání).

Dále vytvořte důkladnou dokumentaci vytvořené části práce (uživatelskou i programátorskou), dle požadavků stanovených vedoucím práce.
Výslednou aplikaci důkladně otestujte, proveďte funkční a uživatelské testy. Ze zpětné vazby uživatelů navrhněte témata pro další vylepšení aplikace.

Seznam doporučené literatury:

Odkaz na tuto práci: Vondra Tomáš. *Reimplementace databáze konferencí a publikací I.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2025.

Chtěl bych poděkovat především své rodině za neustálou podporu nejen při psaní této práce, ale i při celé době mého studia.
Dále děkuji doc. Ing. Petru Fišerovi, Ph.D., za profesionální vedení, cenné připomínky a vstřícný přístup, které mi významně pomohly při realizaci této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

Prohlašuji, že jsem v průběhu příprav a psaní závěrečné práce použil nástroje umělé inteligence. Vygenerovaný obsah jsem ověřil. Stvrzuji, že jsem si vědom, že za obsah závěrečné práce plně zodpovídám.

V Praze dne 6. května 2025

Abstrakt

Diplomová práce se zabývá komplexní reimplementací stávající webové aplikace PubConf, určené ke správě odborných publikací a konferencí na Fakultě informačních technologií ČVUT. Hlavním cílem bylo přepsat původní PHP/-Nette řešení do moderního fullstack prostředí postaveného na Next.js a TypeScriptu, s využitím ORM Prisma a komponentové knihovny Mantine. Součástí práce bylo navržení a implementace modulárního systému autentizace a autorizace uživatelů pomocí Microsoft EntraID (OpenID Connect) a knihovny Next-auth, dále vytvoření DevOps procesu s GitLab CI/CD a Docker kontejnerizací pro automatizované buildování, testování a nasazení aplikace. Kvalita a použitelnost řešení byla ověřena uživatelským testováním, jehož výsledky vedly k doladění UI a odstranění identifikovaných nedostatků.

Klíčová slova Správa publikací, konference, Next.js, TypeScript, Prisma, DevOps, Docker, GitLab CI/CD, autentizace, EntraID, Next-auth, uživatelské rozhraní

Abstract

This thesis addresses the comprehensive reimplementation of the existing PubConf web application, which is used to manage scholarly publications and conferences at the Faculty of Information Technology, Czech Technical University in Prague. The primary goal was to rewrite the original PHP/Nette solution into a modern full-stack environment built on Next.js and TypeScript, leveraging the Prisma ORM and the Mantine component library. The work also involved designing and implementing a modular user authentication and authorization system using Microsoft EntraID (OpenID Connect) and the Next-auth library, as well as establishing a DevOps pipeline with GitLab CI/CD and Docker containerization for automated building, testing, and deployment. The solution's quality and usability were validated through user testing, whose results led to UI refinements and the resolution of identified issues.

Keywords Publication administration, conferences, Next.js, TypeScript, Prisma, DevOps, Docker, GitLab CI/CD, authentization, EntraID, Next-auth, user interface

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam zkratek

ORM	Object–relational mapping
API	Application programming interface
DOM	Document Object Model

Kapitola 1
Úvod

Digitální správa informací je v současnosti nezbytnou součástí většiny odborných a akademických institucí. S rostoucím množstvím dostupných dat je kladen stále větší důraz na efektivní uchovávání, třídění a vyhledávání informací. Právě s tímto cílem vznikají různé softwarové nástroje, které uživatelům pomáhají s organizací a správou informací.

Na Fakultě informačních technologií ČVUT v Praze byla v rámci několika závěrečných prací vyvinuta webová aplikace PubConf. Tato aplikace se specializuje na správu odborných publikací a konferencí a slouží především zaměstnancům a studentům fakulty. Umožňuje ukládat informace o publikacích, konferencích, časopisech, autorech a dalších. Nabízí nástroje pro jejich vyhledávání, filtrování a editaci.

Tato práce navazuje na předchozí vývoj systému PubConf a klade si za cíl vytvořit jeho novou verzi na základě modernějších technologií. Jedná se o týmovou práci, v jejímž rámci se jednotliví členové soustředí na vybrané oblasti aplikace. Moje část práce se zaměřuje zejména na návrh a implementaci správy publikací, autorů a přidružených entit.

Součástí práce je také analýza použitých technologií, vytvoření systému DevOps, návrh architektury aplikace, implementace zodpovědných částí systému a testování s reálnými uživateli. Získaná zpětná vazba slouží k identifikaci a následné eliminaci problémových míst v uživatelském rozhraní. Výsledkem této práce je robustní a snadno rozšiřitelný systém, který reflektuje potřeby koncových uživatelů.

Kapitola 2
Cíle práce

Cílem této diplomové práce je převést původní webovou aplikaci „databáze konferencí a publikací“ (zkráceně PubConf), která byla implementována v jazyce PHP za použití frameworku Nette, do moderních technologií podporovaných ICT oddělením Fakulty informačních technologií ČVUT.

Hlavním požadavkem tohoto převodu je přepsání celé aplikace do jazyka TypeScript, a to na všech jejích aplikačních vrstvách.

Zadání této diplomové práce jsme si rozdělili s Bc. Davidem Kocmanem, který se ve své diplomové práci [1] zaměřil na část aplikace týkající se vytváření a importu publikací a na správu konferencí. Já jsem se soustředil především na oblast správy publikací a přidružených entit, dále na autentizaci a autorizaci uživatelů a na administraci systému jako celku.

Konkrétně jsem se věnoval následujícím úkolům a oblastem aplikace:

- Administrace vývojového prostředí a DevOps:
  - Založení a nastavení projektu
  - Zprovoznění automatizovaného sestavení a nasazení aplikace na server.
- Správa publikací:
  - Seznam publikací a vyhledávání
  - Správa kategorií, časopisů a autorů - jejich vyhledávání, editace a vytváření
- Správa uživatelů a uživatelských skupin
  - Správa uživatelských účtů - vytváření, mazání, editace
  - Správa uživatelských rolí
  - Autorizace a autentizace
- Hlavní stránka

Kapitola 3
Popis aplikace

Protože se tato práce zabývá převážně implementací, nejprve popíši samotnou aplikaci a její hlavní funkce.

Aplikace se zaměřuje na správu publikací, konferencí a souvisejících dat. Umožňuje jejich vyhledávání, úpravu, vytváření a mazání. Navíc spravuje uživatelské účty a jejich role s různými úrovněmi přístupu k datům aplikace. Tyto funkcionality podrobně rozeberu. Při popisu se opírám o minulé práce [2, 3].

Pro lepší pochopení domény aplikace přikládám zjednodušený doménový diagram 3.1. Pro nahlédnutí jsem také jako přílohu přidal uživatelskou příručku B.

3.1 Uživatelé a role

Aplikace spravuje vlastní databázi uživatelů. K tomu, aby měl uživatel do aplikace přístup, potřebuje uživatelský účet. Přihlášení je možné přes poskytovatele Shibboleth pro studenty a zaměstnance FIT ČVUT, nebo jménem a heslem pro externí uživatele.

Úroveň přístupu jednotlivých uživatelů se liší podle přiřazené role. Aplikace rozlišuje celkem tři role – „Reader“, „Submitter“ a „Admin“.

Role „Reader“ uživateli umožňuje číst a vyhledávat jakékoliv záznamy. Tuto roli má implicitně přidělenou.

Role „Submitter“ uživateli umožňuje navíc upravovat jakékoliv záznamy o publikacích a konferencích.

Uživatel s rolí „Admin“ je oprávněn kromě publikací a konferencí spravovat také uživatelské účty.

3.2 Správa konferencí

Uživatel může vyhledávat v seznamu konferencí pomocí textového vyhledávání a dalších filtrů.

Textové vyhledávání lze použít pro vyhledávání podle jména, zkratky a lokace konference. Dále lze použít filtry pro kategorie, oblíbené, živé, archivované a brzy se konající.

„Ročníky konferencí lze filtrovat podle klíčových slov, obou typů kategorií, stavu konference a jejího označení uživatelskou hvězdičkou a podle písmen anglické abecedy. Ročníky konferencí lze řadit podle názvu, zkratky, roku, místa a několika dat. Výsledky filtrování jsou stránkované. “ [3]

Dále může uživatel upravovat konference a jejich přiřazené ročníky. Ročník a konferenci může označit za neaktivní. Ročníky lze zařazovat do indexových databází a jednotlivé konference je možné mezi sebou spojovat.

3.3 Správa publikací

Uživatel může vyhledávat v seznamu publikací pomocí textového vyhledávání a dalších filtrů.

„Aplikácia ponúka v jednom formulári buď jednoduché vyhľadávanie podľa metadát, alebo aj pokročilejšie možnosti ako fulltextové vyhľadávanie alebo vyhľadávanie podľa typu publikácie, kategórii, autorov, tagov či anotácii.“ [2]

Informace evidované u publikace se liší podle typu publikace. Typy publikací odpovídají typům publikací definovaných nástrojem BibTeX. Kromě těchto informací může k publikaci každý uživatel přidat vlastní anotaci a štítek, které mohou být soukromé nebo veřejné.

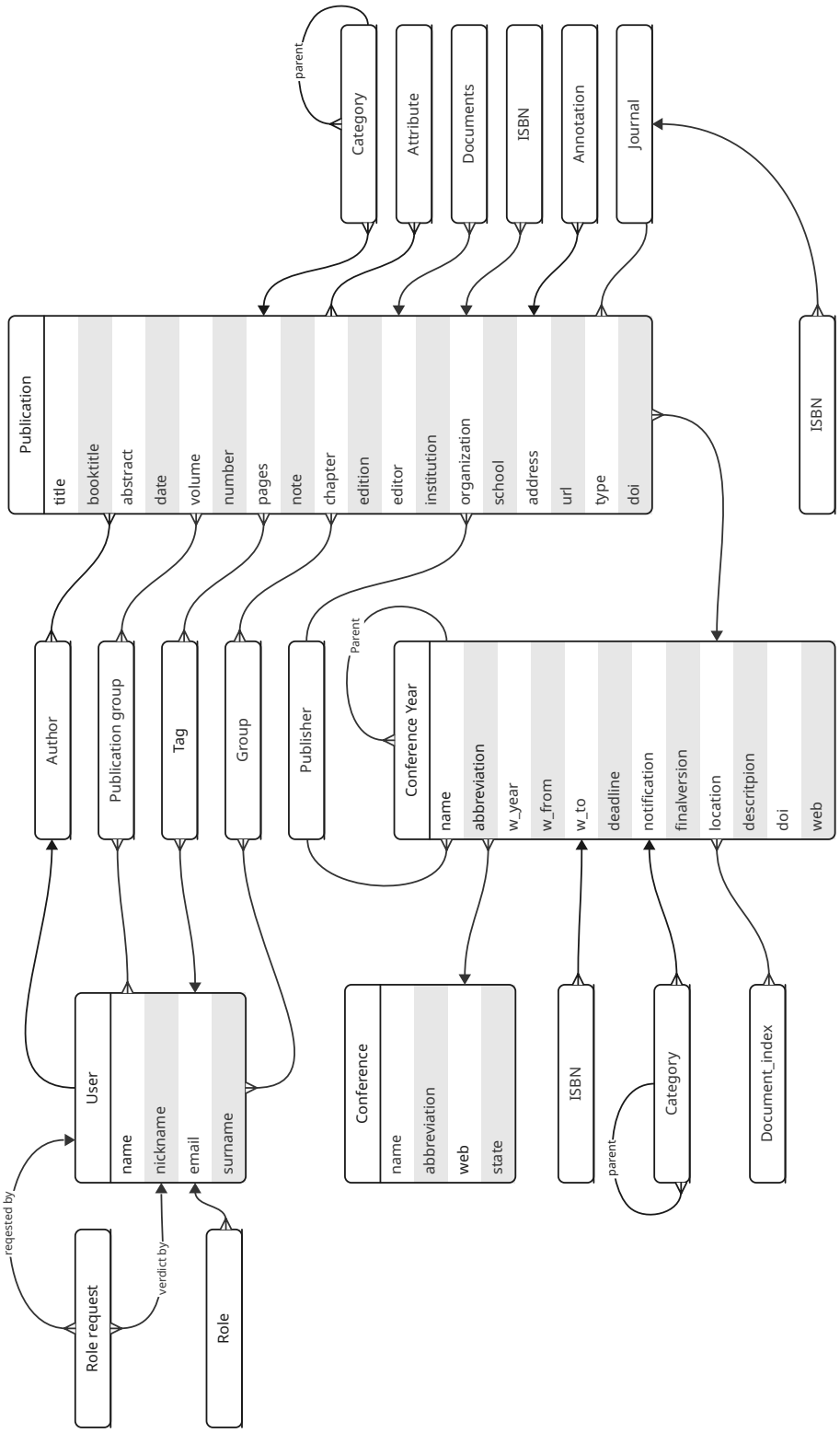
Dále publikace může být exportována do několika standardních formátů, jako je BibTeX a IEEE.

Systém samozřejmě umožňuje publikaci i vytvořit a editovat. Novou publikaci je možné importovat – pomocí BibTeX schématu a stažením dat ze Springeru – nebo vytvořit vyplněním formuláře. K publikaci může být připojen soubor s jejím obsahem.

Pokud je publikace typu `InProceedings`, lze k ní přiřadit i ročník konference. Toto je bod, který propojuje správu konferencí a správu publikací. Tato vazba indikuje, že daná publikace byla zveřejněna ve sborníku dané konference.

3.4 Ostatní

Uživatel může také editovat všechny ostatní záznamy, které lze k konferencím a publikacím přiřadit. Konkrétně jde o autory, časopisy, vydavatele, kategorie publikací a konferencí, uživatelské skupiny, indexové databáze a atributy.



Obrázek 3.1 Zjednodušený doménový model

Kapitola 4
Změny v návrhu

Jak bylo zmíněno v úvodu, práce navazuje na návrh a implementaci původní aplikace zpracovaných v diplomové práci [2]. U některých částí byly provedeny mírné změny návrhu, které zde blíže popíšu; další konkrétnější změny (například změny v databázi) budou zmíněny společně s jejich implementací.

4.1 Navigace

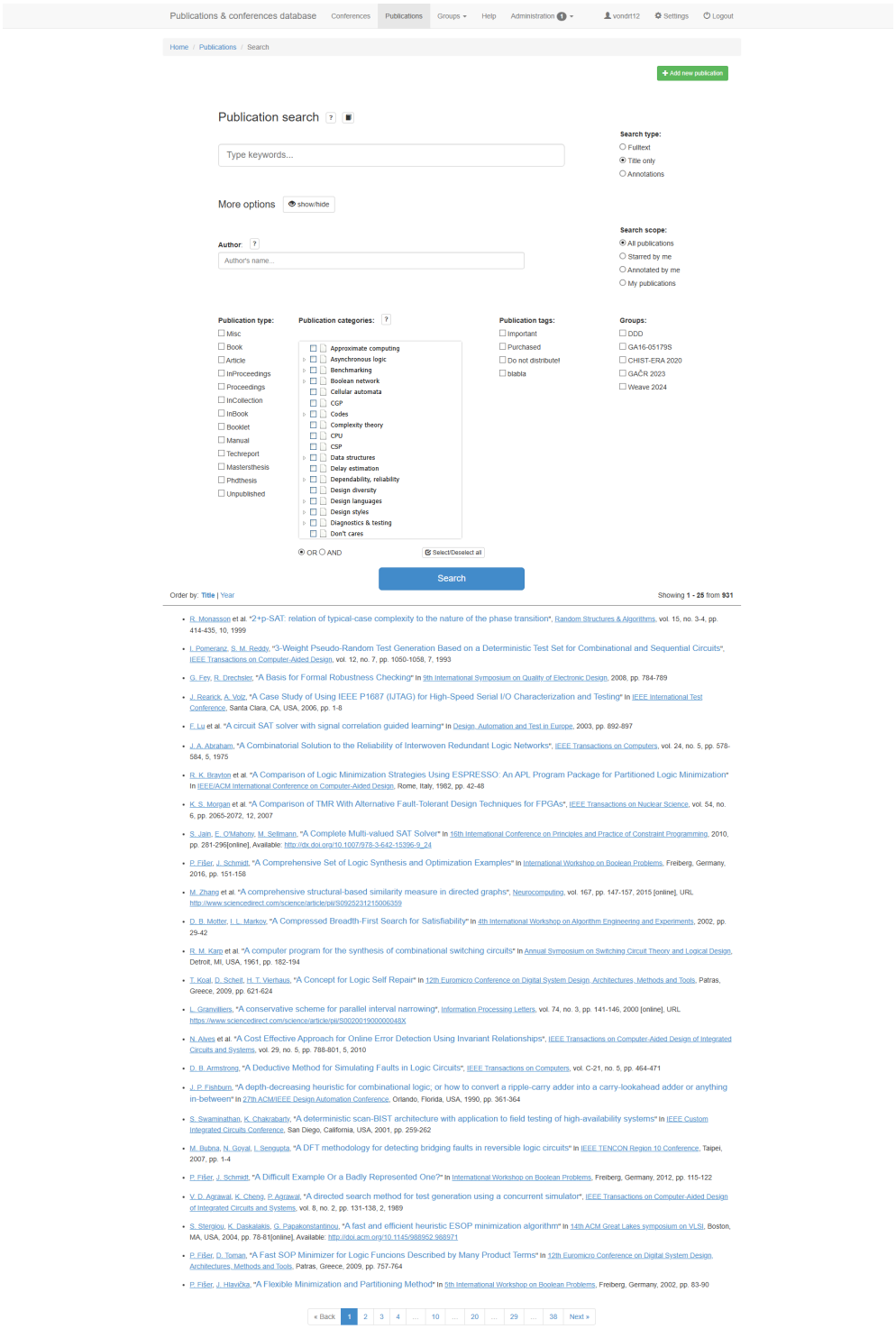
První výraznou změnu oproti původní aplikaci představuje nové rozložení navigace. Stejně jako v původní verzi PubConf se pod hlavní navigací nachází drobečková navigace, avšak odkazy na jednotlivé podstránky byly přesunuty do postranního menu. Díky tomuto rozcestníku se může uživatel snadněji pohybovat mezi stránkami a podstránkami aplikace.

4.2 Seznam publikací

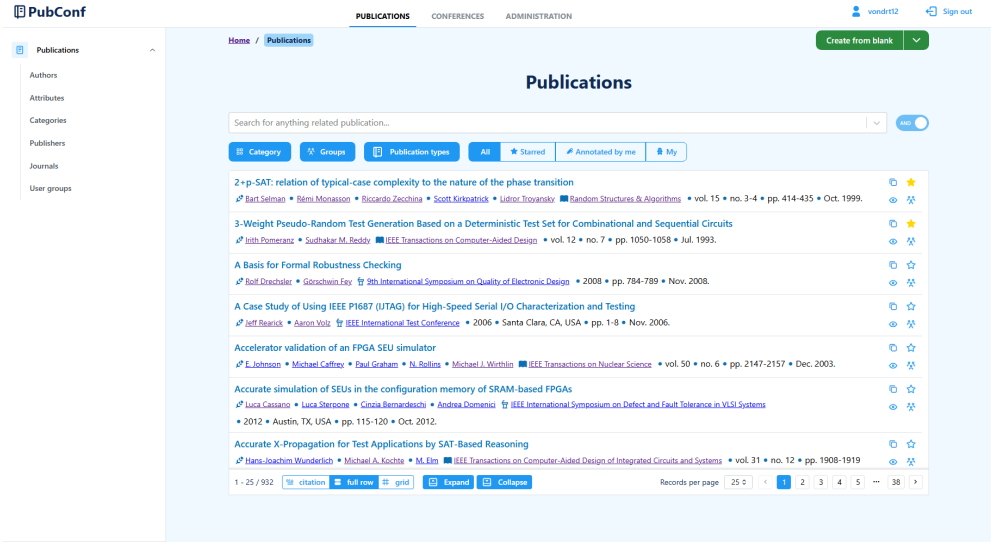
Hlavním interaktivním prvkem v kontextu publikací je vyhledávání a filtrování. Oproti původní aplikaci PubConf nabízí PubConf2 vyhledávání s živou nápovědou, která se dynamicky aktualizuje při změně vstupu od uživatele. Uživatel tak může okamžitě reagovat na výsledky hledání. Dále je ve vyhledávacím poli možné zadat více textových řetězců současně a tím dále omezit výsledky vyhledávání.

Celkové rozložení stránky s vyhledáváním publikací bylo upraveno (viz obrázky 4.1 a 4.2) a nový návrh filtrů je výrazně kompaktnější a uživatelsky přívětivější (viz obrázek 4.3).

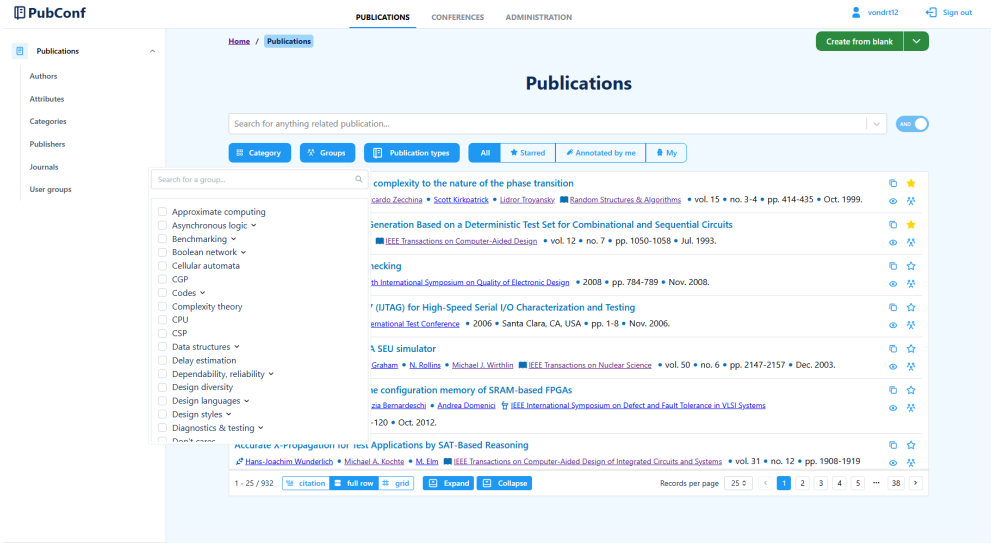
Z původního návrhu zůstala zachována prezentace vyhledaných publikací, avšak nově přibyly další dva pohledy, mezi nimiž může uživatel přepínat.



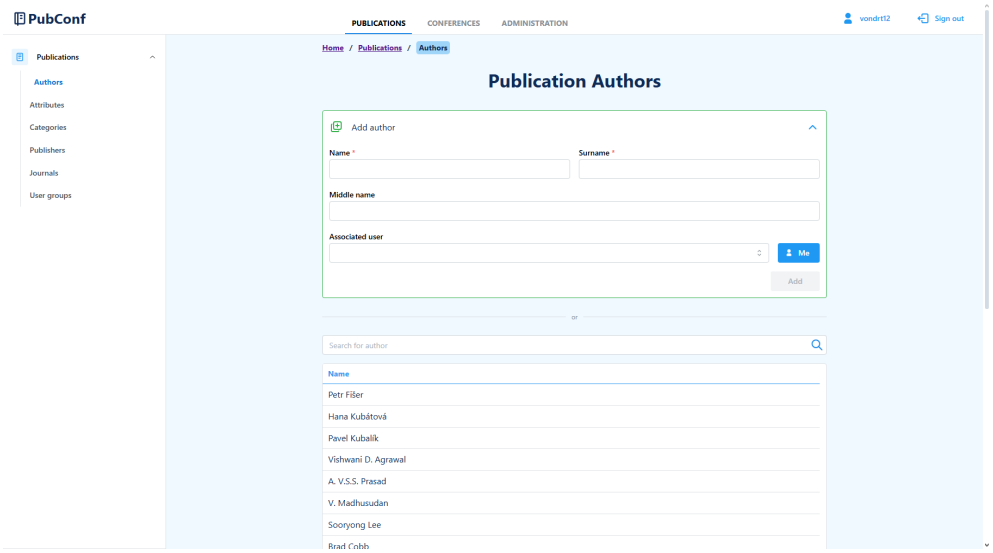
Obrázek 4.1 PubConf - vyhledávání publikací



Obrázek 4.2 PubConf2 - vyhledávání publikací



Obrázek 4.3 PubConf2 - vyhledávání publikací – otevřený filtr



Obrázek 4.4 PubConf2 - správa autorů – seznam

4.3 Správa publikací

Jednotlivé podsekce stránky představují správu pro autory, atributy, kategorie, vydavatele, časopisy a skupiny. Všechny tyto stránky využívají sjednocený návrh, který se z hlediska funkčnosti výrazně neliší od původní aplikace. Se všemi údaji je možné provádět operace vytváření, úpravy a mazání. V těchto údajích lze samozřejmě vyhledávat a prohlížet přidružené publikace.

Autoři mohou být kromě publikací navázáni také na právě jeden uživatelský účet. Tím se označuje, že daný autor reprezentuje konkrétního uživatele. Aby mohl uživatel publikovat pod svým jménem, musí nejprve tohoto autora vytvořit a provázat se svým účtem. Tato akce je dostupná v sekci správy autorů. V novém návrhu zde přibylo tlačítko, které při vytváření nebo úpravě autora automaticky přidá aktuálně přihlášeného uživatele (viz obrázky 4.4 a 4.5).

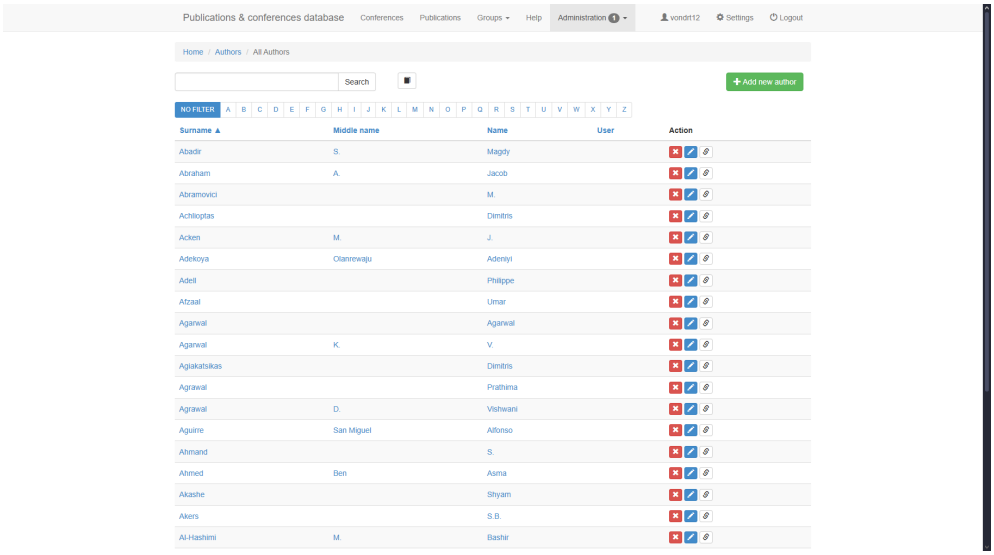
Ostatní podsekce – atributy, vydavatelé, kategorie a skupiny – se funkcionalitou od původní verze aplikace nijak zásadně neliší.

4.4 Administrace

Administrace zajišťuje správu uživatelů a jejich žádostí. Do této části aplikace mají přístup pouze účty s rolí „Admin“.

Ve správě uživatelů je dostupný přehled všech účtů, které v aplikaci existují, včetně jejich přiřazených rolí. Jednotlivé uživatele je možné vyhledat, upravovat jejich role a jméno, případně je zcela odebrat (viz obrázek 4.6).

Uživatelský účet může mít dvě různé formy, které se liší způsobem přihlášení. Uživatel se může přihlásit prostřednictvím školního e-mailu; v takovém



Obrázek 4.5 PubConf - správa autorů - seznam

případě mu bude účet automaticky vytvořen, nebo – pokud již existoval v původní verzi aplikace PubConf – s tímto účtem spárován. Alternativní způsob přihlášení je pomocí jména a hesla. Tento typ účtu může vytvořit pouze administrátor ve správě uživatelů a slouží především pro uživatele mimo ČVUT, tedy například externí spolupracovníky.

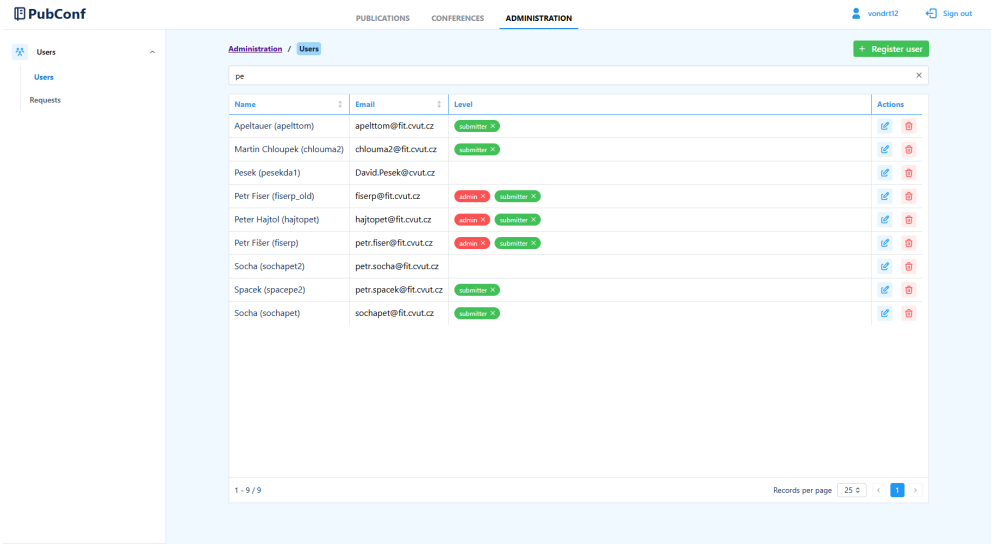
Administrátor může každému uživateli ručně přiřadit roli, zároveň si však uživatel může o roli sám požádat. Druhá část administrace slouží jako přehled těchto žádostí. Administrátor má možnost žádost schválit (čímž je uživateli přiřazena požadovaná role), nebo ji zamítnout (kap. 4.7). Nově se starší žádosti po dosažení maximálního počtu automaticky mažou (o tomto se více rozepíši později).

4.5 Uživatelské nastavení

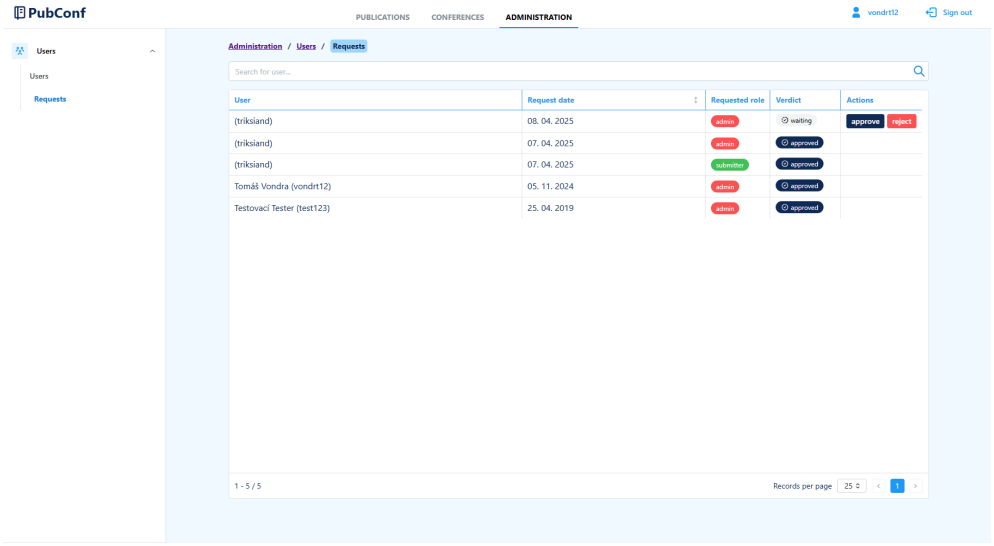
Na stránce uživatelského nastavení se nachází akce specifické pro přihlášeného uživatele. Zobrazují se zde základní informace o jeho účtu, jako je jméno, příjmení a e-mailová adresa. Uživatel zde může rovněž upravit některé prvky chování aplikace. Většina nastavení se týká zobrazení tabulek s filtrovanými daty (např. tabulka publikací).

V původní aplikaci si uživatel mohl nastavit například počet výsledků, které se v tabulce zobrazí najednou, a dále své jméno a příjmení. V nové verzi jsme do této sekce přidali možnost nastavit výchozí pohled tabulky a určovat, zda se mají řádky s dodatečnými informacemi zobrazovat automaticky, nebo být výchozím způsobem skryté.

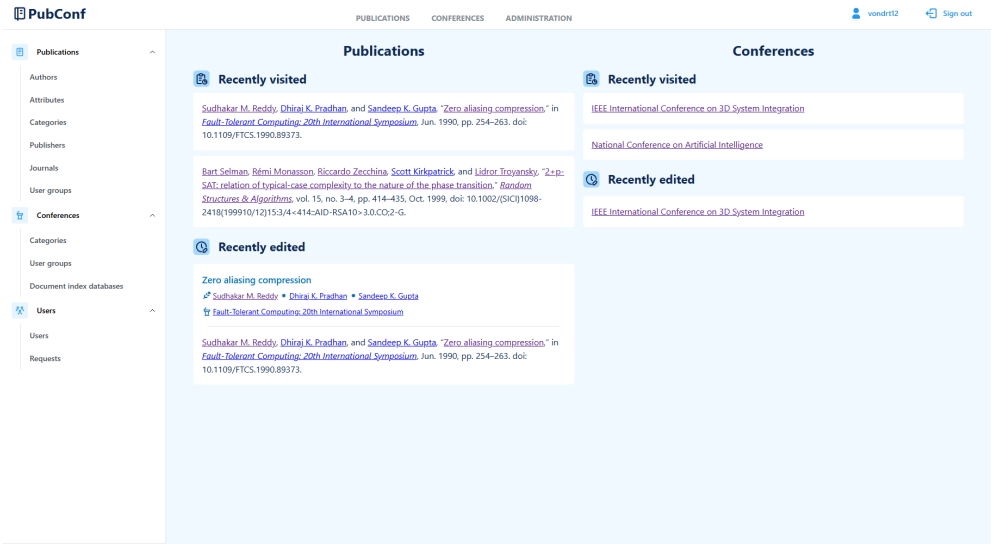
V staré aplikaci byly na této stránce navíc zobrazeny různé agregace dat,



Obrázek 4.6 PubConf2 - správa uživatelů



Obrázek 4.7 PubConf2 - administrace – uživatelské žádosti



Obrázek 4.8 PubConf2 - Domovská stránka

jako uživatelovy oblíbené publikace. Tyto záložky byly v nové verzi odstraněny, protože jejich obsah je nyní možné snadno dohledat pomocí hlavního vyhledávání nebo v příslušných podsekcích.

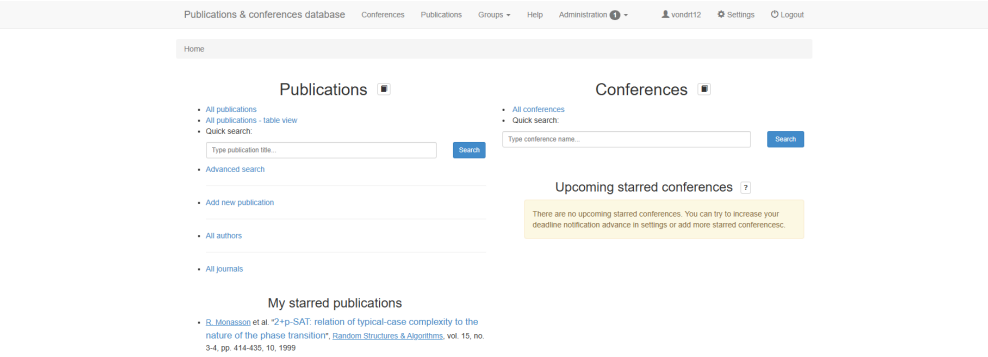
4.6 Domovská stránka

Domovská stránka prošla výraznou změnou návrhu. Původně obsahovala rozcestník, zjednodušené vyhledávací pole pro publikace a konference a také seznam oblíbených publikací a konferencí daného uživatele (viz obrázek 4.9).

V nové verzi jsem vyhledávání odstranil, protože z domovské stránky lze snadno přejít na stránku s plnohodnotným vyhledáváním, a není proto potřeba mít vyhledávání dostupné na více místech. Ze stejného důvodu jsem odstranil také odkazy na oblíbené publikace a konference – stejně jako jsem je odstranil z profilu uživatele.

Domovská stránka nyní dále slouží jako rozcestník na ostatní sekce aplikace. Z postranního menu je možné přejít například na správu publikací nebo konferencí.

Nově jsem přidal sekci s posledně editovanými nebo navštívenými publikacemi a konferencemi. Díky tomu se uživatel může rychle vrátit k nedávno zobrazenému obsahu, aniž by jej musel znovu vyhledávat (viz obrázek 4.8).



Obrázek 4.9 PubConf - Domovská stránka

Kapitola 5
Analýza technologií

V této části práce se krátce zabývám výběrem technologií, které jsme společně s Davidem Kocmanem použili k implementaci aplikace. Při jejich výběru jsme se řídili požadavky ICT oddělení Fakulty informačních technologií, které bude aplikaci v budoucnu spravovat.

Zaměřuji se na volbu vhodného frameworku pro backendovou a frontendovou část aplikace, dále na výběr databázového systému a souvisejících technologií [4]. Nakonec se podívám na DevOps a připojenou infrastrukturu.

Protože se v implementaci zabývám i autorizací a autentizací uživatelů, proberu i technologie, které nám toto v souladu s ICT podmínkami umožňují.

5.1 Backend a frontend

Hlavním požadavkem na backend a frontend byla jejich implementace v jazyce `TypeScript`[4], což výrazně omezilo naše možnosti. Jelikož obě části aplikace měly být implementovány ve stejném jazyce, nabízela se možnost použít `fullstack` framework.

Proberme výhody a nevýhody takového přístupu.

5.1.1 Fullstack framework

Hlavní výhodou `fullstack` frameworku oproti oddělenému backendu a frontendu je absence nutnosti provádět integraci mezi jednotlivými částmi. Díky tomu, že obě části sdílí společnou kódovou základnu (`codebase`), je snazší mezi nimi sdílet části kódu i datové typy. Odpadají také nároky na integrační testování a DevOps, neboť je potřeba spouštět pouze jeden server a sestavovat jedinou aplikaci místo dvou.

Toto spojení technologií frontendu a backendu přináší ještě další výhody. Například umožňuje použití `server-side rendering` (SSR), což je přístup, při kterém se HTML stránky sestaví na aplikačním serveru a spolu s potřeb-

nými styly se pošlou na klientský server (v našem případě prohlížeč), který je může rovnou zobrazit. Oproti tomu existuje přístup `client-side rendering` (CSR), který sestavuje stránku až na klientském zařízení, často za pomoci JavaScriptu. SSR tedy obvykle zajišťuje rychlejší načítání stránek. Pokud celou stránku není možné sestavit na straně serveru, sestaví se jen část stránky a na klientské straně pak dochází k tzv. `hydrataci` (viz [5, 6]).

Nevýhodou tohoto přístupu je silná provázanost mezi frontendem a backendem. Pokud bychom v průběhu vývoje zjistili, že nám daný přístup nebo framework nevyhovuje, přechod na jiný framework nebo rozdělení aplikace na samostatný backend a frontend by byl velmi obtížný. Také nám tato skutečnost může stížit práci při snaze udržet kód čistý a přehledný, neboť zde zaniká klasická hranice mezi backendem a frontendem a obě části musí podléhat jedné architektuře.

Další nevýhodou `fullstack` frameworků je omezený výběr kompatibilních technologií.

5.1.2 Výběr frameworku pro další práci

Pro další vývoj aplikace jsme se rozhodli zvolit `Next.js` jako hlavní framework, a to z důvodu našich předchozích zkušeností s tímto nástrojem z minulých projektů. `Next.js`, postavený nad knihovnou `React`, nabízí flexibilitu a rozsáhlou podporu pro SSR a moderní vývoj webových aplikací. Tento framework nám umožní efektivně vyvinout jak backendovou, tak frontendovou část aplikace a využít všechny výhody `fullstack` přístupu.

Mezi frameworky, které jsme zvažovali, byl i `Nuxt.js`, postavený nad `Vue.js`. Oba frameworky podporují `SSR`, ale přístup `Nuxt.js` je vývojářsky přívětivější a neklade na vývojáře téměř žádné speciální požadavky ohledně jeho implementace. Na druhou stranu `Next.js` vyžaduje explicitní rozdělení částí aplikace na ty, které běží pouze na serveru, a ty, které mohou běžet i na klientském zařízení. Nakonec, i když oba frameworky mají své silné stránky, výběr toho správného by měl vycházet z konkrétních požadavků projektu a schopností členů týmu. Proto jsme pro další vývoj zvolili `Next.js`

5.2 Komponentová knihovna

Komponentová knihovna nám pomůže při vykreslování uživatelského rozhraní. Poskytuje předpřipravené komponenty, jako jsou textbox, dropdown, select a další. Umožňuje tyto komponenty globálně upravovat a spravovat barevná schémata.

V naší aplikaci jsme použili knihovnu `Mantine`. Mantine se skvěle hodí pro vývoj s Next.js. Je bez další konfigurace připravená pro SSR a umožňuje kombinovat různé přístupy k aplikování stylů od globálních specifikací až po konkrétní úpravy na jednotlivých komponentách.

5.3 Databáze a přidružené technologie

Podmínky ICT oddělení omezují výběr databázového stroje na dva konkrétní: PostgreSQL a SQLite. SQLite jsme však vyřadili, protože se jedná o databázi uloženou v jediném souboru [7], což omezuje možnosti paralelního zápisu [8] a škálování [8]. Tyto vlastnosti z ní činí nevhodnou volbu pro středně velké a větší aplikace s více uživateli nebo náročnějším provozem.

Z těchto důvodů jsme zvolili relační databázi PostgreSQL, která nabízí robustní podporu pro souběžný přístup, pokročilé transakční mechanismy a lepší škálovatelnost, což lépe odpovídá našim potřebám [9, 10].

Po volbě databáze bylo potřeba vyřešit, jak bude naše aplikace s databází komunikovat. Pro tento účel jsme zvolili ORM knihovnu `Prisma`. Prisma je vhodná pro prostředí `Node.js` psané v jazyce `TypeScript` a její použití je přímo doporučováno tvůrci frameworku `Next.js` [11]. Umožňuje nám definovat a modifikovat databázové schéma, vytvářet migrace a, co je nejdůležitější, generovat databázového klienta přímo ze schématu. Tento klient slouží k typově bezpečné komunikaci s databází. Součástí generovaného klienta jsou i typy objektů, které odpovídají tabulkám definovaným v databázi. Konkrétnější použití knihovny představíme v dalších částech textu.

5.4 Autentizace a autorizace

Podle daných podmínek by se měli uživatelé aplikace mít možnost přihlásit pomocí systému Shibboleth ČVUT. Celá infrastruktura se ale v době psaní tohoto textu chystá na přechod na technologii `EntraID`. Proto jsem si u ICT oddělení vyžádal možnost tuto technologii také použít, čímž se vyhneme nutnosti v budoucnu technologii migrovat.

`EntraID` (dříve známý jako Azure Active Directory) je cloudová identitní a přístupová služba společnosti Microsoft, která umožňuje centrálně spravovat uživatelské identity a jejich přístupy k aplikacím a systémům. Podporuje standardní autentizační protokoly jako `OAuth 2.0` a `OpenID Connect`, díky čemuž je možné jej snadno integrovat s moderními webovými aplikacemi.

Uživatelé se do aplikace přihlašují pomocí jednotného přihlášení (SSO) přes své školní účty ČVUT, což zjednodušuje správu přístupů a zároveň zvyšuje bezpečnost. `EntraID` rovněž umožňuje zavést vícefaktorové ověřování a další bezpečnostní politiky, což přispívá ke splnění požadavků na ochranu identity a dat.

V naší aplikaci jsem autentizaci realizoval pomocí knihovny `Next-auth`, která poskytuje přímou podporu pro přihlášení přes `EntraID` pomocí protokolu `OpenID Connect`. Po úspěšném ověření získám od služby `EntraID` základní identifikační údaje o uživateli, které pak využívám pro správu přístupových oprávnění uvnitř aplikace [12].

5.5 DevOps

`DevOps` představuje soubor praktik a nástrojů, které propojují vývoj (Development) a provoz (Operations) softwaru s cílem zefektivnit celý životní cyklus aplikace – od vývoje přes testování až po nasazení a provoz. Důraz je kladen na automatizaci, opakovatelnost a rychlou iteraci, což umožňuje častější nasazování nových verzí s vyšší spolehlivostí [13].

Pro implementaci DevOps přístupu v naší aplikaci jsem zvolil službu GitLab, kterou jsme zároveň s kolegou využili pro správu repozitáře a verzování kódu. GitLab poskytuje nástroje pro vytvoření CI/CD (Continuous Integration / Continuous Delivery), které umožňuje průběžnou integraci změn do kódu a jejich automatizované testování a nasazování.

V souladu s požadavky ICT oddělení je nutné aplikaci distribuovat v kontejnerizované podobě. K tomu jsem zvolil standardizovaný a široce používaný nástroj Docker, pomocí kterého můžeme vytvořit předpis prostředí, ve kterém má naše aplikace běžet. Takto definované prostředí je následně spouštěno v takzvaném kontejneru, který je spravován samotnou službou Docker.

Díky tomu, že aplikace neběží přímo v prostředí cílového serveru, ale v izolovaném prostředí kontejneru, vyhnu se tím problémům s nekompatibilitou balíčků a verzí. Zároveň je možné takto zabalenou aplikaci nahrát na vzdálené úložiště a snadno ji znovu spustit kdekoliv, kde je Docker dostupný. Tato vlastnost mi významně pomáhá při organizaci celého CI/CD procesu.

Kapitola 6

Konfigurace a architektura

V této části se dostávám k jádru své práce. Zde se nachází technická dokumentace, která popisuje použití vybraných technologií a přibližuje architekturu a infrastrukturu celé aplikace.

Vývojářská dokumentace s informacemi potřebnými k dalšímu vývoji je obsažena v souboru README.md v kořenovém adresáři aplikace. Konkrétní implementační detaily jsou pak popsány pomocí komentářů přímo v kódu. Zde představím pouze ty nejdůležitější části, technologie a komponenty.

6.1 Infrastruktura a DevOps

V analýze technologií jsem zmiňoval kontejnerizaci a využití služby GitLab. V této části popisuji, jak tyto nástroje využívám v praxi. Abych mohl lépe vysvětlit strukturu DevOps, popíši nejprve celý životní cyklus procesu zavedení nové funkcionality do aplikace. Na tento popis pak navážu konkrétními technologiemi, které jsem použil.

Životní cyklus změny začíná úpravou zdrojového kódu programátorem. To může představovat přidání nové funkcionality, ale i drobnou úpravu, například změnu textu. Tuto změnu programátor zaeviduje (zaverzuje) pomocí nástroje Git a následně nahraje do vzdáleného repozitáře spravovaného službou GitLab. GitLab detekuje změnu a automaticky spustí námi definované procesy, tzv. pipelines. V těchto pipelines se aplikace připraví k nasazení na cílový server, otestuje a nahraje se do kontejnerového registru Docker Registry. Automatizace následně zahájí nasazení na cílový server, kde se společně s migračními skripty spustí připravený kontejner aplikace a nastaví se potřebné proměnné prostředí.

Rozeberu nyní tento proces krok po kroku.

6.1.1 Změny a verzování

Nástroj Git umožňuje vývojářům evidovat změny ve formě tzv. commitů a organizovat je do stromové struktury. Každý repozitář má počáteční commit, od něhož se odvíjejí všechny další změny. Tyto změny je možné organizovat do větví (branches), přičemž každá větev představuje nezávislý vývoj nové funkcionality aplikace. Větve lze následně spojovat (merge), čímž vzniká výsledná verze aplikace.

Speciální větví je hlavní větev (main branch), která představuje produkční verzi aplikace. Pokud GitLab zaznamená změnu v této větvi, spustí automatizaci prvního kroku nasazení.

6.1.2 Gitlab Pipelines

GitLab Pipelines jsou nástrojem pro automatizaci navázanou na verzovací systém Git. Vývojáři umožňují definovat kroky, které se mají provést, prostředí, ve kterém se tyto kroky mají realizovat, a také okamžik jejich spuštění. Prostředí má k dispozici kompletní obsah repozitáře, včetně zdrojových souborů aplikace.

Veškerá konfigurace je uložena ve speciálním souboru `.gitlab-ci.yml`, který je umístěn v kořenové složce aplikace. Jako příklad zde uvádím vlastní konfigurační soubor (viz výpisy 6.1, 6.2 a 6.3). Každý ze souborů představuje jeden krok v dříve popsaném životním cyklu aplikace. Následující část věnuji detailnímu rozboru těchto kroků a jejich fází.

```yaml
build_push:
  image: docker:24.0.5
  stage: build
  services:
    - name: docker:dind
      alias: thedockerhost

  variables:
    DOCKER_HOST: tcp://thedockerhost:2375/
    DOCKER_DRIVER: overlay2
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - echo "$DOCKER_REGISTRY_PASSWORD" | docker login --
        username $DOCKER_REGISTRY_USER --password-stdin
  script:
    - docker build -t $DOCKER_REGISTRY:latest .
    - docker push $DOCKER_REGISTRY:latest
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Výpis kódu 6.1 Ukázka konfigurace CI/CD, část 1.

```yaml
test:
  image: node:lts
  stage: test
  variables:
    # turn off automatic client generation on import
    PRISMA_GENERATE: "false"

  before_script:
    - bash .gitlab/vpn_connect.sh
    - yarn install

  script:
    - yarn run test
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Výpis kódu 6.2 Ukázka konfigurace CI/CD, část 2.

```yaml
 deploy:
  image: ubuntu:latest
  stage: deploy
  before_script:
    ##
    - source .gitlab/vpn_connect.sh
    - echo $DEPLOY_USER
    - echo $DEPLOY_SERVER
    - echo "$DEPLOY_USER@$DEPLOY_SERVER"
    - ssh $DEPLOY_USER@$DEPLOY_SERVER "echo
      $DOCKER_REGISTRY_PASSWORD | docker login --username
      $DOCKER_REGISTRY_USER --password-stdin"
  script:
    - ssh $DEPLOY_USER@$DEPLOY_SERVER "docker compose down
      db app"
    - scp ./docker-compose.yml $DEPLOY_USER@$DEPLOY_SERVER:
      ~/docker-compose.yml
    - scp ./.env.production $DEPLOY_USER@$DEPLOY_SERVER:~/.
      env
    - ssh $DEPLOY_USER@$DEPLOY_SERVER "docker rmi -f
      $DOCKER_REGISTRY"
    - ssh $DEPLOY_USER@$DEPLOY_SERVER "docker pull
      $DOCKER_REGISTRY:latest"
    - ssh $DEPLOY_USER@$DEPLOY_SERVER "AUTH_SECRET=
      $AUTH_SECRET AUTH_AZURE_AD_CLIENT_ID=
      $AUTH_AZURE_AD_CLIENT_ID AUTH_AZURE_AD_SECRET=
      $AUTH_AZURE_AD_SECRET AZURE_AD_TENANT_ID=
      $AZURE_AD_TENANT_ID DATABASE_PASSWORD=
      $DATABASE_PASSWORD docker compose up -d --force-
      recreate db app"
  rules:
```

```
19      - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Výpis kódu 6.3 Ukázka konfigurace CI/CD, část 3.

6.1.3 Build and Push

První krok se jmenuje **build_push**. V tomto kroku se aplikace sestaví, připraví se pro spuštění a nahraje se do Docker registry. Konfigurace se skládá z několika oddělených fází.

V oddílu image a variables definuji prostředí, ve kterém se zbytek fází vykonává. Pomocí oddílu rules pak specifikuji, za jakých podmínek se má tento krok spustit. V tomto případě má krok proběhnout pouze tehdy, pokud se nacházím v hlavní větvi.

Ve fázi before_script provádím úkony, které nastavují prostředí pro hlavní skript. Obsahem hlavního skriptu jsou příkazy k sestavení a kontejnarizaci aplikace a následnému nahrání výsledného obrazu do Docker registry.

6.1.4 Docker build

Než přejdu k další fázi v konfiguraci CI/CD, ukážu nejprve to, jak se aplikace sestavuje.

Jak jsem již zmínil, sestavení a kontejnerizaci provádím pomocí nástroje Docker. Docker mi umožňuje definovat proces sestavení aplikace pomocí konfiguračního souboru nazývaného Dockerfile. Tento soubor je strukturou velmi podobný konfiguraci GitLab pipelines a obsahuje posloupnost instrukcí, podle kterých se vytvoří výsledný obraz aplikace. Následující ukázka představuje konkrétní Dockerfile, který pro svou aplikaci používám (viz výpis 6.4).

```
1  # syntax=docker.io/docker/dockerfile:1
2
3  FROM node:22 AS base
4
5  \# 1. Install dependencies only when needed
6  FROM base AS deps
7  \# Check https://github.com/nodejs/docker-node/tree/
       b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine to
       understand why libc6-compat might be needed.
8  RUN corepack enable
9  RUN corepack prepare yarn@4.5.1 --activate
10
11 WORKDIR /app
12
13 \# Install dependencies based on the preferred package
       manager
14 COPY .yarnrc.yml ./
15 COPY .yarn ./.yarn
16 COPY package.json yarn.lock* ./
```

```
17 RUN yarn install
18
19
20 \# 2. Rebuild the source code only when needed
21 FROM base AS builder
22 WORKDIR /app
23 COPY --from=deps /app/node_modules ./node_modules
24 COPY . .
25 # This will do the trick, use the corresponding env file for
       each environment.
26 COPY .env.production.sample .env.production
27 RUN npx prisma generate
28 RUN npm run build
29
30 COPY start_script.sh ./
31 RUN chmod +x start_script.sh
32
33 # 3. Production image, copy all the files and run next
34 FROM base AS runner
35 WORKDIR /app
36
37 ENV NODE_ENV=production
38 ENV NODE_OPTIONS='--inspect'
39 RUN addgroup --system --gid 1001 nodejs
40 RUN adduser --system --uid 1001 nextjs
41
42 COPY prisma/ ./prisma/
43 # Automatically leverage output traces to reduce image size
44 # https://nextjs.org/docs/advanced-features/output-file-
      tracing
45 COPY --from=builder /app/.next/standalone ./
46 COPY --from=builder /app/.next/static ./.next/static
47 COPY --from=builder /app/public ./public/
48 COPY --from=builder /app/prisma ./prisma/
49 COPY --from=builder /app/start_script.sh ./
50
51 EXPOSE 3000
52
53 ENV PORT=3000
54 \# set hostname to localhost
55 ENV HOSTNAME="0.0.0.0"
56
57 ENTRYPOINT ["./start_script.sh"]
```

Výpis kódu 6.4 Ukázka konfigurace Docker

Aplikaci sestavuji postupně ve třech různých prostředích, která odděluji pomocí direktivy `FROM`. V prvním prostředí instaluji závislosti na projektu – tedy všechny potřebné balíčky a knihovny. V dalším prostředí sestavím aplikaci do výsledného balíčku a nakopíruji do něj potřebné soubory z repozitáře.

Poslední prostředí slouží jako produkční běhové prostředí, ve kterém aplikace skutečně poběží. Zde nastavuji proměnné potřebné pro správné fungování aplikace a kopíruji pouze ty soubory, které jsou nezbytné pro spuštění serveru, aby byla výsledná velikost kontejneru co nejmenší.

6.1.5 Testování

Konfigurace testovacího kroku (viz výpis 6.2) je poměrně přímočará. Klíčovým bodem je spuštění skriptu `vpn_connect.sh`, který zahajuje připojení k fakultní `VPN`. Toto připojení je nezbytné, protože testy vyžadují přístup k databázi, která – stejně jako produkční databáze – běží na fakultním serveru dostupném pouze z fakultní sítě.

Po navázání připojení následuje instalace potřebných balíčků a spuštění samotných testů.

Podrobnější konfiguraci testování popíšu v části věnované knihovně `Vitest` (kap. 6.5).

6.1.6 Deploy

Dále popisuji druhou část konfigurace CI/CD. Tato část se stará o nahrání vytvořeného kontejneru na cílový server a jeho následné spuštění.

Stejně jako v předchozím kroku, je i zde potřeba se připojit na fakultní `VPN`. Následně se připojím pomocí `SSH` k serveru, stáhnu kontejner nahraný v předchozím kroku do Docker registry a spustím ho.

Aby mohla aplikace správně fungovat, je potřeba nastavit tzv. `secrets`. Jedná se o proměnné, které by neměly být nikde veřejně vystavené. Tyto proměnné nesmí být součástí sestaveného obrazu aplikace, a proto je nenastavuji při sestavení, ale až v momentu samotného spuštění kontejneru. Díky tomu nejsou známy ani aplikaci - v sestavené aplikaci nejsou nikde napevno dosazené, ani cílovému serveru, ale pouze běžícímu kontejneru. Správu těchto tajných proměnných zajišťuje GitLab, který pro ně poskytuje bezpečné úložiště a zajistí jejich dosazení při nasazení. Konkrétně se jedná o tyto proměnné:

1. **DOCKER_REGISTRY_PASSWORD** - heslo pro přihlášení do `Docker registry`.
2. **AUTH_SECRET** - šifrovací klíč používaný autentizační knihovnou.
3. **AUTH_AZURE_AD_CLIENT_ID, AUTH_AZURE_AD_SECRET** a **AZURE_AD_TENANT_ID** - údaje potřebné pro konfiguraci autentizace vůči Microsoft EntraID.
4. **DATABASE_PASSWORD** - heslo k databázovému účtu.

6.1.7 Spuštění

Po předchozím kroku přejdeme ke spuštění aplikace. K tomu používám nástroj `Docker Compose`, který slouží k orchestraci Docker kontejnerů. `Docker Compose` umožňuje jedním příkazem spustit vícero kontejnerů najednou, nastavit jejich závislosti a parametry.

V mém případě používám `Docker Compose` ke spuštění kontejneru s aplikací a současně i kontejneru s databází. Před samotným spuštěním aplikace je však potřeba provést migrace databáze. Z toho důvodu musí být databázový kontejner spuštěn dříve než aplikační, což `Docker Compose` umožňuje pomocí definice závislostí mezi službami. Po zajištění, že databáze běží, se automaticky spustí skripty, které provedou databázové migrace a následně se spustí samotná aplikace.

6.2 Cílový server

V této sekci krátce doplním informace o konfiguraci aplikace přímo na cílovém serveru.

Server, na který jsem aplikaci nasadil a na kterém momentálně běží, jsme dostali přidělený naším vedoucím práce. Na tomto serveru však již běží **Apache** server s poměrně rozsáhlou konfigurací pro více různých stránek. Jelikož jsem nemohl vystavit aplikaci přímo na portu 80 kvůli již běžícímu Apache serveru a současně jsem nemohl využít přímo Apache server kvůli tomu, že je aplikace kontejnerizovaná, bylo potřeba zvolit jinou strategii. Nakonec jsem Apache nakonfiguroval tak, aby pro aplikaci fungoval pouze jako proxy.

Celé nastavení Apache proxy pro náš server vypadá následovně 6.5. Klíčová jsou zde pravidla "RewriteRule". Tyto pravidla využívají Apache rewrite engine a zajišťují, aby se každý požadavek, který přijde na port :80 s podadresou /PubConf2, přeposlal přímo na adresu http://127.0.0.1:3000/PubConf2, na které běží naše aplikace. Stejně tak naopak odpovědi naší aplikace se posílají přes port :80 zpět.

```
1 DirectoryIndex disabled
2 RewriteEngine On
3 RewriteRule ^$ http://127.0.0.1:3000/PubConf2 [P,L]
4 RewriteCond %{REQUEST_FILENAME} !-f
5 RewriteCond %{REQUEST_FILENAME} !-d
6 RewriteRule ^(.*)$ http://127.0.0.1:3000/PubConf2/$1 [P,L,NE
    ]
7 \# unset duplicate header.
8 RequestHeader unset X-Forwarded-Host
```

Výpis kódu 6.5 Ukázka konfigurace Docker

6.3 Next.js

V této sekci se zabývám koncepty vývoje ve frameworku `Next.js` a popíši, jak jsem je využil v rámci své práce.

6.3.1 Konfigurace

Konfigurace frameworku Next.js se nachází v kořenovém adresáři projektu v souboru `next.config.mjs`. Tento soubor slouží k nastavení různých aspektů běhu aplikace — například optimalizace build procesu, definice trvalých přesměrování, konfigurace basePath, nebo rozšíření podpory pro další nástroje jako `ESLint` či `Sass`.

V našem projektu konfiguruji následující oblasti:

- **ESLint a Sass:** Nastavuji podporu pro lintování kódu a Sass stylů.
- **Base path:** Definuji `basePath`, což je klíčové pro správné routování na cílovém serveru. Naše aplikace je hostována pod cestou `/PubConf2`, a proto musí veškeré interní odkazy používat stejný základ.
- **Redirects:** Nastavuji trvalá přesměrování — například z kořenové cesty `/` na `/home`, protože uživatelé obvykle očekávají, že domovská stránka bude dostupná na kořeni.

Další možné konfigurační možnosti, které Next.js nabízí, zahrnují nastavení prostředí pro `image optimization`, `režim kompilace` (např. `strict mode`), nebo vlastní Webpack konfiguraci [14].

6.3.2 Proměnné prostředí

Ke konfiguraci Next.js se také váží takzvané proměnné prostředí. Tyto proměnné se používají pro definování konstantních hodnot, které potom můžeme v kódu referencovat. Next.js rozlišuje dva typy proměnných - dosazených při sestavení a dosazených při spuštění.

Proměnné dosazené při spuštění lze referencovat pouze v kódu, který běží na straně serveru.

Proměnné dosazené při sestavení lze referencovat i v klientské části kódu, ale jejich hodnoty jsou při sestavení napevno dosazené v kódu a jsou tedy zjistitelné ze zdrojových souborů v prohlížeči. Definujeme je pomocí prefixu `NEXT_PUBLIC_`.

V naší aplikaci definujeme následující proměnné (některé z nich jsme již zmiňovali v sekci o DevOps):

1. **AUTH_SECRET** - šifrovací klíč používaný autentizační knihovnou.
2. **SPRINGER_URL** - adresa Springer API.

3. **DATABASE_URL** - řetězec pomocí kterého se připojujeme k naší databázi. ve formátu `postgresql://[user[:password]@][netloc][:port][/dbname][?param1=value1&...]`.
4. **BASE_PATH** - základní cesta aplikace.
5. **NEXT_PUBLIC_BASE_PATH** - základní cesta aplikace vystavená na klienta.
6. **FILE_STORAGE** - cesta, do které se ukládají soubory připojené k publikacím.
7. **BASE_URL** - základ URL na které aplikace běží.
8. **AUTH_URL** - základní adresa API autentizace, využívaná knihovnou `Next-Auth`.
9. **NEXT_PUBLIC_FILE_STORAGE_URL** - proměnná `FILE_STORAGE` vystavená na klienta.
10. **AUTH_AZURE_AD_CLIENT_ID** - ID naší aplikace přiřazené poskytovatelem `Microsoft EntraID`.
11. **AUTH_AZURE_AD_SECRET** - klíč vygenerovaný poskytovatelem `Microsoft EntraID` pro naší aplikaci.
12. **AZURE_AD_TENANT_ID** - ID FIT ČVUT přiřazené poskytovatelem `Microsoft EntraID`.
13. **MAX_REQUESTS** - maximální počet uživatelských požadavků než se začne mazat nejstarší.

6.3.3 Navigace

Jelikož je aplikace z velké části vykreslována na serveru, navigace mezi stránkami probíhá pomocí klasických HTTP požadavků, tedy převážně prostřednictvím HTML odkazů. Tento přístup kontrastuje s koncepcí jednostránkových aplikací (SPA), kde většina navigace probíhá na klientovi pomocí JavaScriptu.

Framework `Next.js` generuje pro každou stránku samostatný HTTP `GET` endpoint, jehož umístění odpovídá složkové struktuře projektu [15]. Každý soubor v adresáři `app/` definuje samostatnou stránku nebo komponentu, přičemž název složky či souboru určuje cestu v rámci URL. Tento přístup umožňuje statickou i dynamickou tvorbu stránek, a to jak na straně serveru, tak i klienta [15].

Díky tomuto přístupu je také jednoduché udržet strukturu projektu přehlednou a organizovanou, neboť to funkcionalita přímo vynucuje.

Co je na tomto přístupu nevýhodné, je skutečnost, že Next.js nedokáže automaticky generovat odkazy na základě názvů stránek. Odkazy tedy musíme

zadávat jako prosté textové řetězce s plnou URL adresou (relativní vůči kořenové doméně). Tento přístup je náchylný k překlepům a jiným chybám. Navíc pokud dojde ke změně adresy stránky, je nutné ručně upravit každý výskyt odkazu v aplikaci.

routeGetter Problém s generováním odkazů řeším zavedením tzv. `routeGetter` objektů. Tyto objekty reprezentují jednotlivé segmenty aplikace a obsahují funkce, které vrací plnohodnotné adresy na konkrétní stránky daného segmentu (budeme jim říkat *gettery*). Objekty lze do sebe libovolně vnořovat tak, aby odpovídaly složkové struktuře aplikace.

Příklad `routeGetter` objektu pro správu uživatelů je uveden v listingu 6.6.

V příkladu si také můžeme povšimnout použití metody `withBasePath`. Tato metoda přijímá jako první parametr textový řetězec `basePath` a jako druhý parametr `routeGetter` objekt. Vrací nový `routeGetter`, jehož každá metoda (`getter`) je obalena funkcí, která k výsledku připojí zadaný `basePath`. Díky tomu je možné jednoduše vytvářet hierarchii `routeGetter` objektů odpovídající struktuře URL adres.

Tímto přístupem jsem centralizoval definici odkazů v celé aplikaci.

```
export const Users = withBasePath('/users', {
    getRoot() {
      return '';
    },
    detail: {
        getProfile(id: string) {
          return `/${id}/profile`;
        },
        getAnnotations(id: string) {
          return `/${id}/annotations`;
        },
        getPublications(id: string) {
          return `/${id}/publications`;
        },
        getTags(id: string) {
          return `/${id}/tags`;
        },
    },
});
```

Výpis kódu 6.6 routeGetter - správa uživatelů

6.3.4 Serverové a klientské části kódu

Jedním z klíčových konceptů `Next.js` je rozdělení kódu na část běžící na serveru a část běžící na klientovi. Výchozím chováním je vykonávání komponent

na straně serveru. Pokud je potřeba, aby se komponenta vykreslovala nebo chovala jako interaktivní prvek v prohlížeči, je nutné ji explicitně označit pomocí direktivy `"use client"` [16].

Tímto způsobem vývojář přesně definuje, které části kódu poběží na serveru a které na klientovi. Tento přístup přispívá k lepší bezpečnosti a výkonnosti aplikace, neboť citlivé části logiky zůstávají na serveru a na klienta se přenáší pouze nutné minimum [16].

My se v projektu snažíme, aby co největší část aplikace běžela pouze na serveru. Jako klientské komponenty vystavujeme pouze ty, které vyžadují interaktivitu.

Klientské komponenty by navíc neměly samy stahovat data z databáze. Pokud tato data potřebují, měly by jim být předána rodičovskou komponentou vykreslovanou na serveru.

6.3.5 Server Actions a Middleware

Server actions Next.js zavádí nový koncept tzv. `Server Actions`, které umožňují zpracovávat formuláře a jiné akce přímo na serveru, bez potřeby samostatného API endpointu. Jedná se o funkce vykonávané přímo při zpracování požadavku serverem, přičemž jejich volání může být integrováno například do HTML formulářů pomocí atributu `action` [17].

Middleware Další důležitou součástí Next.js jsou `middleware` – funkce, které se spouštějí při každém příchozím požadavku ještě předtím, než se zpracuje konkrétní stránka. Middleware lze použít například pro přesměrování, autentizaci nebo přidání vlastních HTTP hlaviček. Umísťují se do souboru `middleware.ts` na kořenové úrovni projektu a běží na serveru v tzv. Edge runtime [18].

O použití serverových akcí a middleware se více zmiňuji v sekci o architektuře projektu.

6.4 Prisma

Jak jsem již zmiňoval v analýze technologií, knihovna `Prisma` poskytuje ORM (Object-relational mapping) rozhraní, které zprostředkovává komunikaci mezi naší aplikací a databází. Představíme si základní koncepty této knihovny.

Schema Centrální součástí Prismy je soubor `schema.prisma`, ve kterém se definují datové modely aplikace. Tyto modely reprezentují databázové tabulky a jejich vztahy, a zároveň slouží jako zdroj pro generování typově bezpečného klienta, pomocí něhož aplikace přistupuje k databázi.

Migrace Změny v datovém modelu aplikace se spravují pomocí systému migrací. Při každé úpravě struktury datového modelu Prisma umožňuje automaticky vytvořit migrační skript, který reflektuje provedené změny. Tento skript se následně spouští při nasazení aplikace na server, což zajišťuje, že struktura databáze odpovídá aktuálnímu datovému modelu.

Prisma klient Dotazy nad databází provádíme pomocí `prisma` objektu, vygenerovaného pomocí příkazu `prisma generate`. Tento objekt je potřeba instancovat pomocí třídy `Prisma Client`.

Prisma klient se stará o připojení k databázi a spravuje k tomu vlastní vláknový fond. Aby nedocházelo k vytvoření dalšího spojení při každém požadavku na server, je nutné, aby byl tento objekt instancován jako tzv. `singleton` (viz. [19]).

Middleware Prisma podporuje vlastní systém middleware v podobě `Prisma extensions`. Ten funguje podobně jako middleware v Next.js. Jedná se o funkce, které se vykonají při každém požadavku na databázi ještě před samotným zpracováním požadavku. Díky tomu lze například upravit chování dotazů nebo přidat logování či autentizaci na úrovni databázových operací [20].

Těchto rozšíření může být několik a aktivují se připojením k Prisma klientovi pomocí metody `$extends` při jeho instancování. Tyto rozšíření se dají řetězit a definovat tak pořadí, v jakém se dotaz v rozšířeních zpracovává. Jako ukázku zde přiložím část konfigurace našeho Prisma klienta 6.7

```
1  /**
2  It is necessary to cast this to PrismaClient as it
       reduces the amount of type instances significantly.
       When you remove this, the autocompletion for prisma
       object is way too slow.
3  **/
4  const prismaInstance = new PrismaClient()
5    .$extends(submitterExtension)
6    .$extends(rightsRequestExtension) as PrismaClient;
7  const globalForPrisma = globalThis as unknown as {
       prisma: PrismaClient };
8  export const prisma = globalForPrisma.prisma ||
       prismaInstance;
```

Výpis kódu 6.7 konfigurace prisma klienta - middleware

6.5 Vitest

`Vitest` je moderní testovací knihovna pro JavaScript a TypeScript, která klade důraz na rychlost, jednoduchost a dobrou integraci s vývojovým prostředím. Kromě základní podpory pro psaní unit testů nabízí také snapshot testování,

mocking a běh v režimu `watch`. Díky plné podpoře TypeScriptu a velmi rychlé odezvě je vhodná jak pro frontendový, tak backendový vývoj.

V této práci jsem `Vitest` využil především k psaní unit testů pro serverové akce. Tyto akce zpravidla sestávají převážně z volání databáze prostřednictvím knihovny `Prisma`, a proto je nezbytné, aby měly testy přístup k reálné testovací databázi. V následující části popisuji konfiguraci, kterou jsem pro tento účel zavedl.

vitest.config.js Globální konfigurace testovacího prostředí se nachází v souboru `vitest.config.js`. V tomto souboru definuji prostředí, ve kterém testy běží, a nastavím potřebné proměnné jak pro lokální vývoj, tak pro prostředí GitLab CI. Dále zde specifikuji soubory, které se spouštějí globálně před všemi testy (viz výpis 6.8), a soubory, které se spouštějí před každým testovacím souborem zvlášť (viz výpis 6.9).

```
    const conf = config({ path: '.env.test' });
    expand(conf)
    const adminClient = new PrismaClient({
        datasources: {
            db: {
                // tell Prisma to use your test database URL
                url: process.env.ADMIN_DATABASE_URL,
            },
        },
    });
    await adminClient.$connect();
    adminClient.$executeRawUnsafe(
        `SELECT 'CREATE DATABASE ${process.env.DATABASE_NAME}' WHERE NOT EXISTS (
            SELECT FROM pg_database WHERE datname = '${process.env.DATABASE_NAME}'
            )\\gexec`
    )
    await adminClient.$disconnect();
    //migrate public schema
    execSync(`DATABASE_URL=${process.env.TEST_DATABASE_URL} npx prisma migrate deploy`);
    //create snapshot
    execSync(
        `DATABASE_URL=${process.env.TEST_DATABASE_URL} npx prisma migrate diff \
    --from-empty \
    --to-schema-datamodel=prisma/schema.prisma \
    --script > test/fixtures/schema.sql`,
        { stdio: 'inherit' }
    );
```

Výpis kódu 6.8 Vitest globální příprava

```
    const mockedUser = {
        submitter_id: BigInt(99),
        id: '0',
        roles: ['ADMIN'],
        email: 'a@b',
        username: 'u',
        emailVerified: null,
    };

    vi.mock('@/auth/auth', () => ({
        auth: vi.fn(async () => ({ user: mockedUser })),
    }));

    vi.mock("@/lib/prisma/client", async ()=> {
        const {setupIsolatedDatabase} = await vi.
            importActual<typeof import("@/vitest.
            isolatedDatabase.ts")>("@/vitest.isolatedDatabase
            ")
        const {client} = await setupIsolatedDatabase();
        return {prisma: client};
})
```

Výpis kódu 6.9 Vitest individuální příprava

Globální příprava Každý testovací `Vitest` worker musí běžet v izolovaném databázovém schématu, aby si paralelně spuštěné testy nepřepisovaly data. Proto globální konfigurace nejprve načte proměnné prostředí ze souboru `.env.test`, který definuje přístupové údaje k databázi. V lokálním prostředí odpovídá tato konfigurace výchozí testovací databázi, zatímco v GitLab CI jsou hodnoty přepsány proměnnými z nastavení repozitáře.

Pokud databáze ještě neexistuje, je vytvořena, a následně je na ni aplikována migrace pomocí příkazu `prisma migrate deploy`. Po úspěšné migraci se pomocí `prisma migrate diff` vytvoří SQL snímek struktury databáze. Tento snímek je následně použit k vytvoření izolovaných schémat pro jednotlivé testovací workery. Tento postup je nezbytný, protože `Prisma` nedovoluje provádět paralelní migrace na různých schématech v jedné databázi.

Individuální příprava Před spuštěním každého testu se pomocí individuálního setupu vytvoří izolované schéma databáze a připraví se potřebné mock objekty. Kritickým krokem je zde nahrazení výchozího `Prisma` klienta klientem, který se připojuje k právě vytvořenému testovacímu schématu. Díky tomu testy pracují vždy s čistými daty a jsou navzájem izolované.

Úklid Po dokončení testů je nutné testovací databázi „uklidit“ a odstranit všechna vytvořená schémata, aby se zamezilo jejich hromadění. `Vitest` k tomu

poskytuje funkci `teardown`, kterou využívám k odstranění testovací databáze po ukončení testovací relace. Tento krok zaručuje, že po skončení testování nezůstávají v databázi žádné zbytkové artefakty.

Testy Samotné soubory s unit testy se nacházejí ve složce `__tests__` v kořenovém adresáři projektu. Pro ilustraci přikládám testování serverové akce `filterPublishers` (viz výpis 6.10).

Na začátku testu je nutné namockovat závislosti, které nejsou předmětem testování — například funkci `auth`. Následují sekce `beforeEach` a `afterAll`. Sekce `beforeEach` se spouští před každým testem a slouží k přípravě dat, se kterými test pracuje. Naopak `afterAll` se vykoná po dokončení všech testů a umožňuje tzv. „úklid“ — tedy odstranění dočasných dat nebo uvolnění prostředků.

Při lokálním vývoji lze testy spustit příkazem `yarn run test`.

```
describe('filterPublishers (integration)', () => {
  beforeEach(async () => {
    // wipe the table
    await prisma.publisher.deleteMany();
    // seed with known data
    await prisma.publisher.createMany({
      data: [
        { name: 'Alpha Press', address: 'London' },
        { name: 'Beta Books', address: 'Berlin' },
        { name: 'Gamma Publishing', address: 'Athens' },
        { name: 'Delta House', address: 'Denver' },
      ],
    });
  });

  afterAll(async () => {
    await prisma.$disconnect();
  });

  test('returns all when search is empty', async () => {
    const { data, totalCount } = await filterPublishers('',
        10, 1);
    // we seeded 4 publishers
    expect(totalCount).toBe(4);
    // and pageSize=10 so you get all 4
    expect(data.map((p) => p.name).sort()).toEqual([
      'Alpha Press',
      'Beta Books',
      'Delta House',
      'Gamma Publishing',
    ]);
   });
  });
```

Výpis kódu 6.10 unit testy - filterPublishers

6.6 React

Frontendová část frameworku Next.js je postavena na knihovně *React*, která umožňuje vytvářet moderní uživatelská rozhraní pomocí komponentového přístupu. Základní myšlenkou Reactu je, že uživatelské rozhraní lze popsat jako funkci nad stavem aplikace. Jakmile se stav změní, React provede efektivní přepočet a aktualizuje pouze ty části DOM stromu, které změnou skutečně prošly. To zajišťuje vysoký výkon i přehlednou strukturu kódu.

Představíme si dva návrhové vzory, které v aplikaci hojně používáme.

Provider pattern Důležitou součástí vývoje v Reactu je také tzv. *Provider pattern*. Tento koncept slouží k předávání sdílených dat (například informace o přihlášeném uživateli) napříč komponentami. Pomocí `Provider` komponenty lze zpřístupnit daný kontext všem vnořeným komponentám, aniž by bylo potřeba data explicitně předávat přes argumenty komponent v každé úrovni.

Hook pattern S konceptem poskytovatelů (`Provider`) úzce souvisí také *React hooky*. Hooky umožňují komponentám pracovat se stavem, životním cyklem a dalšími funkcemi Reactu. Nejčastěji používané hooky jsou například `useState` pro práci se stavem, `useEffect` pro reakce na změny dat nebo `useContext` pro přístup ke kontextu poskytnutému pomocí `Provideru`.

Použitím vlastních hooků můžeme navíc definovat opakovaně použitelnou logiku, která je snadno sdílitelná napříč komponentami. Tento přístup podporuje modularitu a přehlednost aplikace.

6.7 Mantine

`Mantine` je komponentová knihovna pro React. Nabízí široké spektrum často používaných komponent a umožňuje jejich globální úpravy prostřednictvím `Theme` souboru. V tomto souboru lze definovat vlastní barevné palety a proměnné pro kaskádové styly.

Konfigurace Moje vlastní konfigurace 6.11 je poměrně krátká. Definuji jen několik barev, které v aplikaci používáme, a nastavuji základní barvy pro texty.

```
1     export const theme = createTheme({
2   components: {
3     Skeleton: Skeleton.extend({
4       classNames: {
5         root: classes.darkSkeleton,
```

```
      },
    }),
    Text: Text.extend({
      styles: (th) => ({
        root: {
          color: th.colors.darkText[0],
        },
      }),
    }),
    Title: Title.extend({
      styles: (th) => ({
        root: {
          color: th.colors.darkText[0],
        },
      }),
    }),
  },
  colors: {
    lightBlue: [
      '#F0F9FF',
      '#cfecff',
        //...
    ],
    category: colorsTuple(DEFAULT_THEME.colors.orange[6]),
    tag: colorsTuple(DEFAULT_THEME.colors.red[6]),
    group: colorsTuple(DEFAULT_THEME.colors.green[9]),
    author: colorsTuple(DEFAULT_THEME.colors.grape[4]),
    buttonGreen: [
      '#effaf2',
      '#dff2e3',
      //...
    ],
    buttonRed: colorsTuple('#bc2727'),
    title: colorsTuple(DEFAULT_THEME.colors.blue[4]),
    darkText: colorsTuple('#102A56'),
    lightText: colorsTuple('#0D6EB4'),
  },
  primaryColor: 'lightBlue',
  shadows: {
    sharp: 'rgba(50, 50, 93, 0.25) 0px 2px 5px -1px, rgba(0,
        0, 0, 0.3) 0px 1px 3px -1px;',
    sharpBig: 'rgba(50, 50, 93, 0.25) 0px 6px 12px -2px,
       rgba(0, 0, 0, 0.3) 0px 3px 7px -3px;',
  },
});
```

Výpis kódu 6.11 Mantine theme - theme.ts

CSS moduly Mantine podporuje jak dynamické, tak statické stylování komponent. Hlavním způsobem stylování jsou tzv. `CSS moduly`. Tyto moduly umožňují importovat styly přímo do JavaScriptových souborů a přiřazovat komponentám třídy v nich definované. Výhodou tohoto přístupu je, že se styly vztahují pouze na komponenty v rámci konkrétního souboru, čímž se minimalizuje riziko konfliktu názvů tříd.

My navíc používáme rozšíření klasického CSS — `Sass` [21]. `Sass` (Syntactically Awesome Stylesheets) přináší množství užitečných funkcí, které klasické CSS nepodporuje, například proměnné, zanořování, dědičnost stylů pomocí `@extend`, mixiny nebo logiku pomocí podmínek a cyklů. Díky těmto vlastnostem umožňuje psát strukturovanější, přehlednější a lépe udržovatelný kód.

Kód psaný v `.scss` nebo `.sass` souborech se následně překládá do běžného CSS, které je možné načítat jako CSS moduly.

Style API Další možností je stylování komponent pomocí předdefinovaných atributů (např. `h` pro výšku, `w` pro šířku atd.), které jsou součástí API každé Mantine komponenty. Pokud potřebujeme upravit styl komponenty detailněji, lze využít atribut `style`, do kterého můžeme přímo zadat JavaScriptový objekt s CSS vlastnostmi.

Dynamické styly V praxi často potřebujeme komponentě přiřadit určitou CSS třídu pouze při splnění určité podmínky. V Reactu se tento problém běžně řeší formátováním textového řetězce, což se však při větším počtu podmínek stává nepřehledným. Proto jsem vytvořil pomocnou funkci `pc` (zkratka pro „process classes“), která přijímá objekt, jehož klíčem je název třídy a hodnotou logická hodnota. Funkce pak z tohoto objektu vytvoří textový řetězec obsahující pouze třídy, jejichž hodnota je `true`.

```
    <nav
        className={pc({
          [classes.navbar]: true,
          [classes.transition]: true,
          [classes.collapsed]: !expanded,
        })}
      >
        <ScrollArea className={pc({ [classes.links]: true, [
            classes.collapsed]: !expanded })}>
          <div>{links}</div>
        </ScrollArea>
        <div className={classes.footer}></div>
      </nav>
```

Výpis kódu 6.12 Příklad použití funkce pc

6.8 Architektura

V této sekci rozebereme architekturu našeho projektu a popíšeme jeho strukturu.

Architektura, kterou jsme při vývoji zvolili, vychází z principů definovaných frameworkem Next.js. Její základní myšlenky lze shrnout následovně:

- Struktura projektu odráží strukturu URL adres a je dále rozdělena na části kódu běžící na serveru a na klientu.
- Společné části stránek (např. hlavička, navigace) jsou umístěny ve speciálních layout souborech.
- Manipulace s daty probíhá výhradně na serveru prostřednictvím server akcí.
- Data z databáze by se měla načítat primárně v serverových komponentách.
- Jednotlivé části uživatelského rozhraní jsou složené ze znovupoužitelných React komponent.
- Pokud je stejné chování použito ve víc různých komponentách, je abstrahováno pomocí React hook vzoru.
- Pokud více vnořených komponent sdílí společná data, jsou poskytována pomocí React provider vzoru.

Dobrou představu o tomto uspořádání si lze udělat ze složkové struktury projektu. Složky na nejvyšší úrovni znázorňuje obrázek 6.1.

```
/ - Hlavní adresář aplikace
├── app - Obsahuje jednotlivé stránky a routy (Next.js App Router)
├── auth - Přihlašování a NextAuth konfigurace
├── components - Znovupoužitelné UI komponenty
├── hooks - Vlastní React hooky
├── lib - Server akce, utility a konstanty
└── prisma - ORM klient, schéma databáze a migrace
```

Obrázek 6.1 Struktura složek projektu

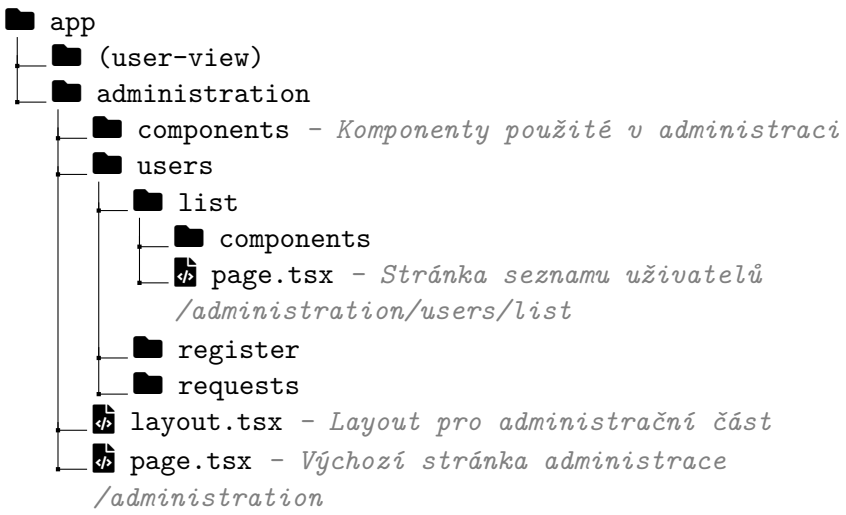
app Tato složka obsahuje veškeré stránky aplikace.

Každá podsložka představuje jednu část URL a pokud obsahuje soubor `page.tsx`, je považována za plnohodnotnou stránku. Některé složky navíc obsahují podadresář `components`, který sdružuje React komponenty určené výhradně pro danou stránku nebo její podstránky (viz obrázek 6.2).

auth Ve složce `auth` se nachází konfigurace a pomocné funkce související s autentizací a autorizací pomocí NextAuth. Podrobněji se této problematice věnuji v kapitole o implementaci.

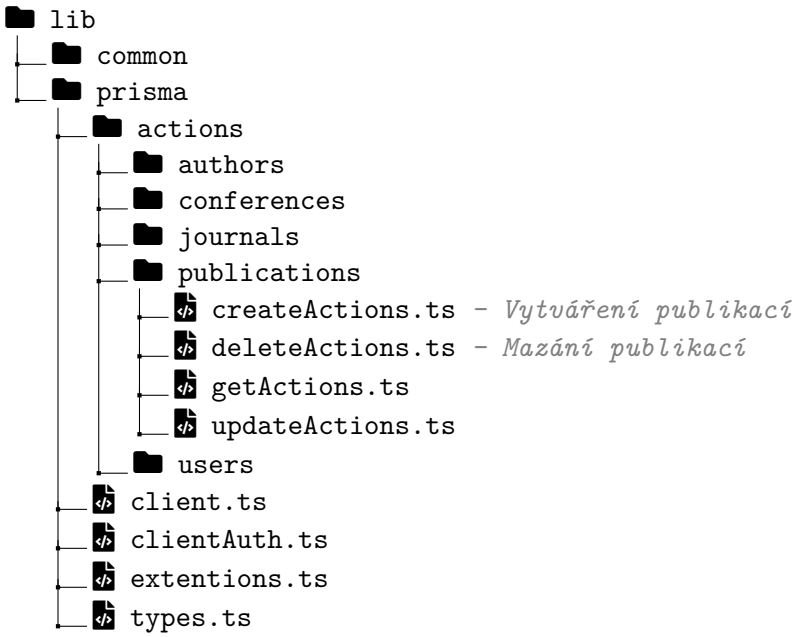
lib Adresář `lib` obsahuje námi definované typy, konstanty, pomocné funkce a především server akce.

Společně s komponentami tvoří server akce jeden z nejdůležitějších stavebních kamenů celé architektury. Jelikož instance našeho databázového klienta Prisma je dostupná pouze na straně serveru[1], je manipulace s daty možná výhradně přes tyto akce. Tímto způsobem koncentrujeme doménovou logiku na jedno místo a můžeme ji snadno organizovat podle typu akce i podle domény aplikace.



Obrázek 6.2 Struktura složky `app/administration`

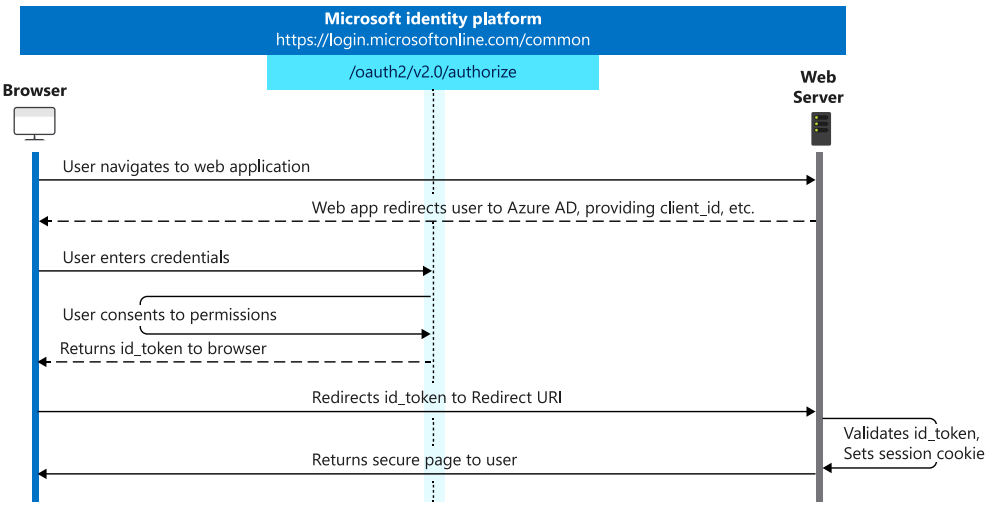
[1]Server-side komponenty mohou teoreticky přistupovat přímo k Prisma klientovi, považuji to však za antipattern. Takové řešení vede k rozdrobení doménové logiky do komponent, což výrazně ztěžuje její správu a testování.



Obrázek 6.3 Struktura adresáře `lib/`

6.9 Autentizace a autorizace

Výklad o implementaci aplikace začnu autentizací, protože ta ovlivňuje všechny další části aplikace. K její implementaci jsem použil knihovnu Next-auth, která zajišťuje celý proces `OAuth 2.0 authorization code flow` (viz diagram 6.4).



Obrázek 6.4 Schéma průběhu autorizace a přihlášení dle Microsoft Entra ID [22]

Po úspěšném přihlášení knihovna vystaví tzv. JWT token, který obsahuje

informace o uživateli. Tento token je uložen v HTTP cookie a je spolu s obsahem stránky odeslán na klientský server. V prohlížeči je cookie uchována až do odhlášení uživatele. Při každém dalším požadavku na server je cookie automaticky odesílána a její obsah použit k autorizaci uživatele.

6.9.1 Konfigurace

Nejprve je však potřeba knihovnu správně nakonfigurovat a určit poskytovatele identity. Next-auth poskytuje mnoho modulů s připravenými konfiguracemi pro různé poskytovatele.

V mém případě je poskytovatelem Microsoft Entra ID. Zde není třeba žádné speciální nastavení, postačuje pouze specifikovat hodnoty `clientId` a `clientSecret`. Tyto hodnoty, které jsem získal od ICT oddělení, slouží k autentizaci naší aplikace v rámci komunikace s Entra ID v OAuth flow. Entra ID si díky nim ověří, že naše aplikace je důvěryhodná, a může jí poskytnout údaje o uživateli.

Dále je potřeba zajistit, aby nepřihlášený uživatel byl přesměrován na přihlašovací stránku. K tomu slouží Next.js middleware. Jelikož nechci, aby jakákoliv stránka byla přístupná bez přihlášení, v middlewaru kontroluji každý příchozí požadavek. Ověřuji, zda požadavek obsahuje validní JWT token, a pokud tomu tak není, uživatel je automaticky přesměrován na přihlašovací stránku.

Nakonec je potřeba specifikovat údaje, které se budou posílat, uložené v JWT tokenu. Tyto údaje pak lze použít například k určení typu uživatele a jeho oprávnění. Na základě těchto informací lze blokovat provedení různých serverových akcí nebo upravit zobrazení uživatelského rozhraní. Já jsem konkrétně přidal do tokenu navíc parametr `role`, který představuje uživatelovu roli a práva a klíč do tabulky Submitter.

V tomto bodě je autentizace plně nakonfigurována. Následně bylo potřeba vytvořit endpointy, které knihovna Next-auth používá pro komunikaci s poskytovatelem identity.

6.9.2 Backend API

Při běžné konfiguraci stačí přidat soubor typu Next.js route handler do cesty `/api/auth/[...nextauth]`, který pokrývá všechny požadavky směřující na URL `/api/auth`.

V našem případě však máme nastavenou základní cestu (`basePath`) a aplikace je zároveň nasazena za Apache proxy. To způsobuje, že požadavky na autentizaci přicházejí na nesprávnou URL adresu. Tento problém jsem vyřešil přepisem adresy pomocí vlastní funkce `rewriteRequest`, která upraví příchozí požadavek do tvaru odpovídajícího očekáváním knihovny Next-auth (viz listing 6.13).

```
1    function rewriteRequest(request: NextRequest) {
```

```
  const { protocol, host, pathname } = request.nextUrl;

  const { headers } = request;
  // Host rewrite adopted from next-auth/packages/core/src/lib/utils/env.ts:createActionURL
  const detectedHost = headers.get('x-forwarded-host') ?? host;
  const detectedProtocol = headers.get('x-forwarded-proto') ?? protocol;
  const _protocol = detectedProtocol.endsWith(':') ? detectedProtocol : `${detectedProtocol}:`;
  const url = new URL(
    `${_protocol}//${detectedHost}${process.env.BASE_PATH ?? ''}${pathname}${request.nextUrl.search}`
  );

  return new NextRequest(url, request);
}
```

Výpis kódu 6.13 Next-auth backend API - rewriteRequest

6.9.3 Databáze a migrace

Next-auth kromě kombinace JWT a cookies využívá k ukládání informací o uživateli také databázi. Vyžaduje proto existenci specifických tabulek a vazeb, které musí být přítomny ve schématu. Z tohoto důvodu jsem musel upravit existující databázové schéma a provést migraci dat.

Aby mohl Next-auth s naší databází komunikovat, je třeba mu poskytnout tzv. `adaptér`. Tento objekt definuje rozhraní, pomocí kterého může knihovna přistupovat k databázi prostřednictvím `Prisma` klienta.

Migrace V původní aplikaci byla reprezentace uživatele rozdělena do tří různých tabulek (viz obr. 6.5). Next-auth však požaduje schéma uvedené na obrázku 6.6. Tabulky `Auth_shibboleth` a `Auth_login_password` byly sjednoceny do jedné tabulky `User`. Různé typy přihlášení nyní rozlišuji pomocí atributu `type` v tabulce `Account`.

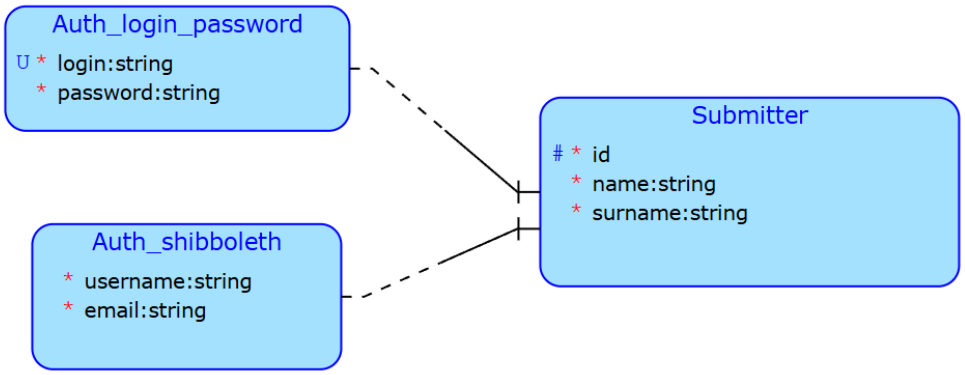
Nejprve jsem transformoval tabulku `Auth_shibboleth` na `User`. Protože původní tabulka měla primární klíč typu BigInt a nová tabulka vyžadovala textový identifikátor, bylo nutné vygenerovat nové hodnoty klíčů.

Migrace proběhla následujícím způsobem:

1. Přejmenování tabulky `auth_shibboleth` na `User`.
2. Přidání nového sloupce `userId` s hodnotami typu CUID.
3. Nastavení sloupce `userId` jako primárního klíče.
4. Smazání původního sloupce `id` a přejmenování `userId` na `id`.
5. Doplnění dalších atributů, jako `username`, do tabulky `User`.

Následně jsem vytvořil tabulku `Account` a přesunul do ní data z tabulky `Auth_login_password`:

1. Přidání tabulky `Account`.
2. Pro každý záznam v `Auth_login_password` vytvoření odpovídajících řádků v `User` a `Account`.
   - Pokud již existoval uživatel se stejným e-mailem, nevytvářel se nový záznam v `User`, ale pouze nový účet v `Account`.
3. Smazání tabulky `Auth_login_password`.



Obrázek 6.5 PubConf - původní model autentizace

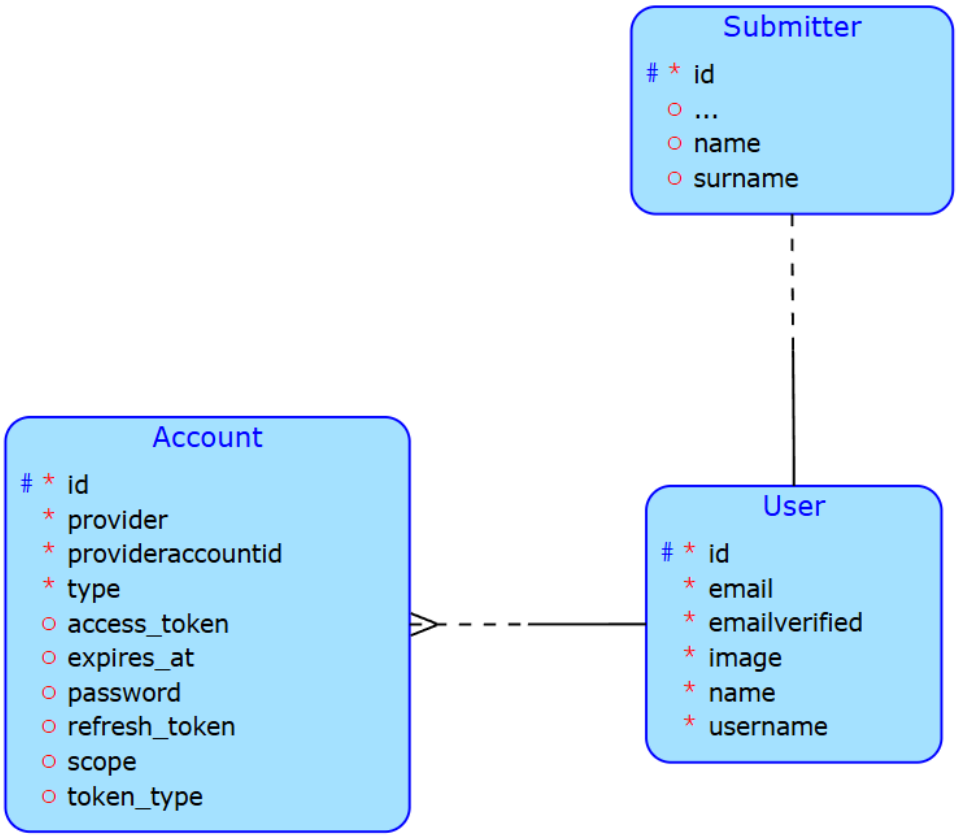
Nakonec jsem upravil tabulku `User_role`. V původní aplikaci existovalo pět různých rolí, reprezentovaných jako textový řetězec.

V nové verzi jsem zachoval dvě hlavní role: `Submitter` a `Admin`. Role `Submitter` umožňuje spravovat publikace a konference, zatímco `Admin` má navíc oprávnění spravovat uživatele.

Nově zaregistrovaný uživatel nemá přiřazenu žádnou roli a má tedy pouze práva pro čtení. Sloučil jsem některé starší role do jedné a změnil datový typ sloupce `role` na výčtový typ (`Enum`). To umožňuje bezpečnější a přehlednější práci s rolemi a zároveň zabraňuje uložení neplatných hodnot do databáze.

Adaptér NextAuth spravuje, podobně jako v případě konfigurace poskytovatelů, velké množství adaptérů pro populární databázové klienty – včetně adaptéru pro Prismu.

Pomocí adaptéru knihovna NextAuth načítá data z databáze a nahrává je do JWT tokenu. V základním nastavení adaptér načítá pouze data z tabulky `User`. V našem případě však potřebujeme, aby byly v JWT tokenu dostupné i údaje z tabulek `Submitter` a `User_role`. Z tohoto důvodu jsem knihovní adaptér musel mírně rozšířit.



Obrázek 6.6 PubConf2 - model autentizace po migraci na Next-auth

Jako základ mého adaptéru `PrismaAdapterProxy` jsem použil oficiální knihovní adaptér, ke kterému jsem přidal vlastní definice funkcí `getUserByEmail`, `getUser`, `getUserByAccount` a `createUser`.

První tři z uvedených funkcí slouží k načtení uživatele z databáze na základě různých identifikátorů. Nově vrací nejen data z tabulky `User`, ale také přidružené informace z tabulek `Submitter` a `User_role`.

Funkce `createUser`, která se volá při prvním přihlášení uživatele, vytváří záznam v tabulce `User` a zároveň automaticky přidává i odpovídající záznam v tabulce `Submitter`, protože každý uživatel musí být reprezentován v obou tabulkách.

Protože prostředí, ve kterém adaptér běží, nemůže používat prisma rozšíření, nepoužívá můj adaptér stejnou instanci prisma klienta jako zbytek aplikace. Pro potřeby adaptéru jsem vytvořil druhou instanci `prismaAuth`.

6.9.4 Použití v kódu

Jak jsem již zmínil, role uložená v JWT tokenu mi umožňuje dynamicky upravovat obsah stránky na základě uživatelských oprávnění. Například mohu skrýt formulář pro úpravu autorů nebo celou sekci pro správu uživatelů. Stejným způsobem zabezpečuji také serverové akce. Pokud by uživatel vyvolal akci, ke které nemá oprávnění, server vyhodí výjimku.

Způsob, jak získat informaci o uživateli, se liší v závislosti na prostředí, ve kterém se komponenta nachází.

Pokud je daná část vykreslována na serveru, lze token získat přímo z hlaviček HTTP požadavku. K tomu slouží pomocná funkce `auth()`, která token z požadavku extrahuje a dekóduje.

Pokud se nacházím v klientské komponentě, není možné token získat přímo z požadavku. V tomto případě ho lze načíst ze speciální provider komponenty `SessionProvider` umístěné v kořenovém souboru `layout.tsx`, do které byl token vložen při serverovém vykreslení stránky.

Zabezpečení server akcí Přestože Next.js middleware zpracovává volání serverových akcí, nemohu ho využít pro hromadné zabezpečení. Pokud bych například při neoprávněném přístupu vrátil výjimku, server by na volání nevrátil žádnou odpověď a klient by čekal až do vypršení časového limitu.

Abych mohl zabezpečení serverových akcí centralizovat i bez použití middleware, implementoval jsem obalovací funkce `withAuthUserCanEdit` a `withSubmitterCanEdit`. Pokud bych chtěl, aby byla nějaká serverová akce přístupná pouze uživatelům s oprávněním editace, obalím její definici voláním jedné z těchto funkcí.

Na obrázku 6.14 lze nahlédnout, jak vypadá takové použití.

```
1 export const addPublisher = withAuthUserCanEdit(
2     async (name: string) =>
3         prisma.publisher.create({
```

```
            data: {
                name,
            },
        })
);
```

Výpis kódu 6.14 Funkce chráněná pomocí withAuthUserCanEdit

Obě wrapper funkce fungují tak, že před samotným voláním vložené funkce zkontrolují pomocí funkce `auth()` roli přihlášeného uživatele. Pokud uživatel nemá potřebná práva, vyhodí výjimku. V opačném případě pokračují voláním původní funkce.

V případě, že vložená funkce potřebuje znát údaje o přihlášeném uživateli, wrapper jí tyto údaje předá jako jeden z argumentů. `withAuthUserCanEdit` a `withSubmitterCanEdit` se liší pouze tím, jakou informaci o uživateli předávají.

Implementace těchto wrapper funkcí je poměrně jednoduchá a přímočará. Na ukázku přikládám kód funkce `withAuthUserCanEdit` 6.15.

```
export const withAuthUserCanEdit =
  <T extends any[], TReturn>(
    action: (...args: T) => Promise<TReturn>
  ): WithUserReturn<T, TReturn, ExtendedSessionUser> =>
  async (...args: WithUserInferArgs<T, ExtendedSessionUser>)
      => {
    const user = await getUserAuthOrThrow();
    user.canEditOrThrow();
    const anyAction = action as (...args: any[]) => Promise<
       TReturn>;
    if (action.length === args.length + 1) {
      return anyAction(...args, user);
    }
    return anyAction(...args);
  };
```

Výpis kódu 6.15 Autentizace – wrapper funkce `withAuthUserCanEdit`

6.10 Middleware

V této sekci se podíváme, jak v aplikaci využívám middleware. Velkou výhodou tohoto přístupu je možnost centralizovat opakující se kód na jedno místo. Například ověřování, zda má daný uživatel oprávnění upravovat určitou entitu, nemusíme opakovaně psát ve všech funkcích, které tento typ operace provádějí. Místo toho vložíme tuto logiku do middleware, který se zavolá automaticky pokaždé, když je daná akce požadována.

Tento přístup výrazně zjednodušuje vývoj i údržbu aplikace. Na druhou stranu má však i svá úskalí. Jelikož samotná volající funkce o existenci middle-

ware neví, může být obtížné odhalit příčinu problému, pokud právě middleware způsobí neočekávané chování nebo výjimku. Takové chování může vést ke zmatení, obzvlášť pokud není existence middleware dostatečně zdokumentována nebo známá celému týmu.

Z těchto důvodů považuji za důležité zmínit použití middleware i v této práci. Vědomí o jejich přítomnosti je zásadní pro správné pochopení chování systému, a především pro usnadnění jeho ladění a dalšího rozvoje.

V aplikaci používám dva druhy middlewaru, Next.js middleware a Prisma extensions.

Využití Next.js middlewaru už jsem zmiňoval u autentizace. Používám ho primárně na ověření, jestli je uživatel přihlášený.

Prisma extensions představují rozšíření vygenerovaného ORM klienta. Umožňují například upravovat parametry volaných Prisma funkcí ještě předtím, než jsou převedeny na databázové dotazy, nebo rozšiřovat funkcionalitu některých modelů o vlastní metody.

V rámci aplikace jsem implementoval dvě vlastní *Prisma extensions*.

6.10.1 Prisma extension

Většina tabulek v databázi obsahuje sloupec, který zaznamenává, jaký uživatel daný záznam vytvořil. Získání přihlášeného uživatele vyžaduje volání speciální funkce, přičemž tento údaj často potřebujeme předat na více místech v rámci jednoho dotazu. Proto jsem se rozhodl tuto logiku abstrahovat pomocí jednoho z Prisma rozšíření.

Toto rozšíření přerušuje volání metody `create` pro tabulky, které mají zmíněnou vazbu na uživatele. Do záznamu automaticky doplní informace o právě přihlášeném uživateli a následně pokračuje ve vykonání původní operace. Jelikož jedno volání může obsahovat i více vnořených volání (například pokud se při vytváření nové publikace zároveň vytvoří nový autor a rovnou se k publikaci přiřadí), musí rozšíření fungovat rekurzivně i pro vnořené části dotazu. Zde přikládám výstřižek kódu s komentáři 6.16

```
export const submitterExtension = Prisma.defineExtension((
    prisma) =>
  prisma.$extends({
    name: 'submitterExtension',
    query: {
      // extension is called on all models defined in schema
          .prisma
      $allModels: {
        // extension is called on all operations (create,
            update, delete)
        $allOperations: withNestedOperations({
          // toto se provede pro črodiovské hlavní volání.
```

```
          async $rootOperation({ query, args, operation,
              model }) {
            if (operation === 'create') {
              // name of a field in query argument object
              const mappedToSubmitterField =
                  registerSubmitterMap[model];
              if (!mappedToSubmitterField) return query(args
                  );
              const session = await auth(); // get signed in
                   user
              if (!session?.user) return query(args);
              // inserts the connect object into query args.
                   This argument will connect signed in user
                  to this model.
              return query({
                ...args,
                data: {
                  ...(args.data ? args.data : {}),
                  [mappedToSubmitterField]: {
                    connect: {
                      id: session.user.submitter_id,
                    },
                  },
                },
              });
            }
            return query(args);
          },
          //this is called for all nested operations
          async $allNestedOperations({ query, args,
              operation, model }) {
            ... //its identicall with the parent code
          },
        }),
      },
    },
  })
);
```

Výpis kódu 6.16 Prisma extension - připojení uživatele

Další rozšíření slouží pro mazání starých záznamů po přidání nové uživatelské žádosti. Pokud by tabulka s žádostmi měla přidáním nového záznamu překročit určitý počet žádostí, odstraní rozšíření nejstarší vyřízenou žádost. Díky tomu, že jsem tuto funkcionalitu odsunul do middlewaru, nemusíme se o ni starat, pokud bychom přidávali žádosti z několika míst kódu najednou. Pro ilustraci znovu přikládám výstřižek kódu.

```
export const rightsRequestExtension = Prisma.defineExtension
    ((prisma) =>
  prisma.$extends({
    name: 'rightsRequestExtension',
    query: {
      rights_request: {
        create: async ({ query, args }) => {
          const result = query(args);
          const totalCount = await prisma.rights_request.
              count();
          if (totalCount > maxRequests) {
            await prisma.rights_request.deleteMany({
              where: {
              //deletes all requests that are not in "
                  maxRequests" and are not in state "waiting
                  ".
                rights_request_id: {
                  notIn: (
                    await prisma.rights_request.findMany({
                      orderBy: {
                        request_datetime: 'desc',
                      },
                      take: maxRequests,
                      select: {
                        rights_request_id: true,
                      },
                    })
                  ).map((record) => record.rights_request_id
                      ),
                },
                verdict: {
                  not: 'waiting',
                },
              },
            });
          }
          return result;
        },
      },
    },
  })
);
```

Výpis kódu 6.17 Prisma extension - uživatelské žádosti

Kapitola 7
Stránky a funkcionality

V této části budu pokračovat v technické dokumentaci, ale zaměřím se na konkrétní vybrané stránky a funkcionality, které byly zmíněny už v popisu aplikace. Proberu jejich implementaci a nejdůležitější součásti.

7.1 Seznam publikací

Začněme nejdůležitější stránkou – seznamem publikací. Na této stránce může uživatel pomocí filtrů a vyhledávacího pole vyhledat konkrétní publikaci a následně se navigovat na její detail, případně na detail autora, časopisu a dalších souvisejících entit.

Hlavní část stránky se skládá z vyhledávacího pole, sady filtrů a tabulky s výsledky. Stav a logiku této části obsluhuje React provider komponenta `PublicationFilterProvider`, jejíž funkcionalita je zpřístupněna pomocí hooku `usePublicationFilter`. Veškerá logika pro filtrování, vyhledávání, stránkování a řazení je centralizována v serverové akci `filterPublications`, kterou provider volá s různými parametry v závislosti na aktivních filtrech.

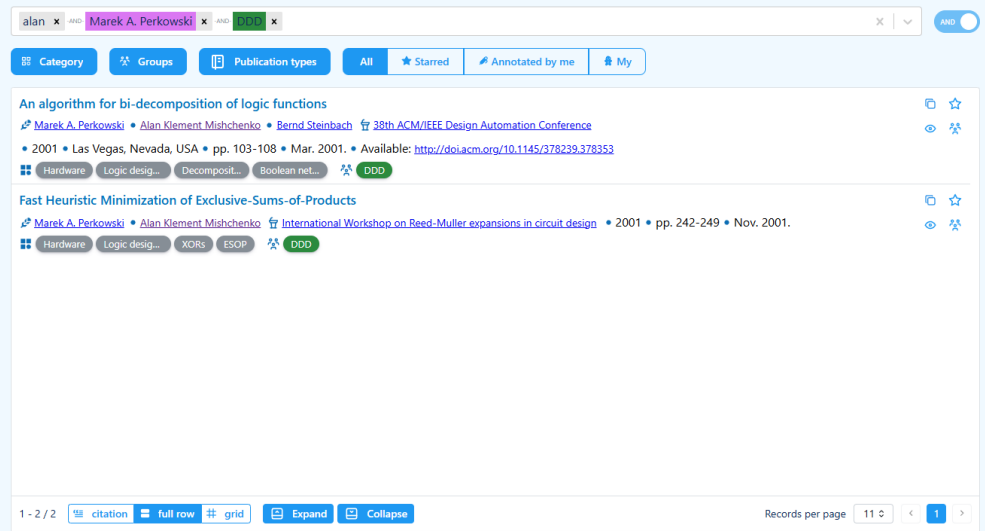
Stav filtrů si `Provider` uchovává pomocí hooku `useOptionallyPersistedState`, který ukládá hodnoty do úložiště prohlížeče. Díky tomu zůstávají zvolené filtry aktivní i po přechodu na jinou stránku nebo při obnově aplikace.

Protože však v některých částech systému (například ve správě autorů) toto chování není žádoucí, je možné jej vypnout pomocí parametru `persist`. Pokud je tato volba deaktivována, hook se chová jako běžný `useState`.

Původně byla tato funkcionalita implementována pouze pro seznam publikací. Vzhledem k tomu, že část aplikace, kterou implementoval David Kocman ve své práci [1], obsahuje obdobný seznam s vyhledáváním, abstrahoval ji později do znovupoužitelné generické komponenty. Díky tomu je nyní možné vytvořit další obdobné seznamy s podporou vyhledávání, filtrování a řazení pouze pomocí serverové akce, která dodržuje předepsané rozhraní.

7.1.1 Vyhledávací pole

V původní aplikaci sloužilo vyhledávací pole výhradně k vyhledávání podle názvu publikace. Já jsem jeho funkčnost rozšířil tak, aby bylo možné vyhledávat i podle autora, skupiny, značky, kategorie a roku vydání. Uživatel má také možnost zadat více klíčových slov současně. Na obrázku 7.1 je příklad vyhledávání podle klíčového slova „alan“, autora „Marek A. Perkowski“ a skupiny „DDD“.



Obrázek 7.1 PubConf – vyhledávání publikací s více filtry

Jako základní komponentu pro vyhledávání jsem použil knihovnu `React Select` [23]. Ačkoliv naše UI knihovna Mantine nabízí vlastní alternativu, její API nebylo pro mé potřeby dostatečně flexibilní.

7.1.2 Filtry

Pod vyhledávacím polem se nachází sada filtrů. Ty umožňují vyhledávání podle dalších atributů, konkrétně kategorií, skupin a typu publikace. Dále lze filtrovat pouze oblíbené publikace, publikace anotované uživatelem nebo ty, které daný uživatel vytvořil.

Smyslem těchto filtrů je zobrazit uživateli všechny dostupné možnosti, podle kterých může výsledky omezit. Navíc, oproti původní aplikaci, podporují v nabízených možnostech vyhledávat, což je užitečné zejména u rozsáhlých kategorií (např. více než 300 položek).

Všechny filtry jsou postaveny na společné komponentě `FilterMenuButton`, která zajišťuje zobrazení seznamu a stav vyhledávacího pole. Tato komponenta je připojena na celkový stav filtrů přes hook `usePublicationFilter`.

7.1.3 Tabulka

Funkcionality tabulky jsem částečně rozšířil. Nově umožňuje přepínání mezi různými pohledy.

Řádky tabulky lze zobrazit v kompaktním režimu bez dodatečných informací. Tyto informace je možné zobrazit jednotlivě pro každý řádek, hromadně pro aktuálně zobrazené řádky pomocí tlačítka „Expand“ ve spodní části tabulky, nebo globálně v uživatelském nastavení.

Nově jsem přidal také možnost přiřadit publikaci do jedné nebo více skupin přímo z tabulky.

Jako základ pro implementaci jsem využil komponentu `Mantine DataTable` [24], která poskytuje základní funkcionality jako stránkování, řazení a další.

7.2 Správa publikací

V této sekci se budu zabývat stránkami souvisejícími se správou publikací. Mezi ně patří správa autorů, časopisů, kategorií, vydavatelů, atributů a skupin.

Všechny tyto stránky používají stejné rozložení a komponenty. Představím, jaké mají funkce, vlastnosti a implementaci.

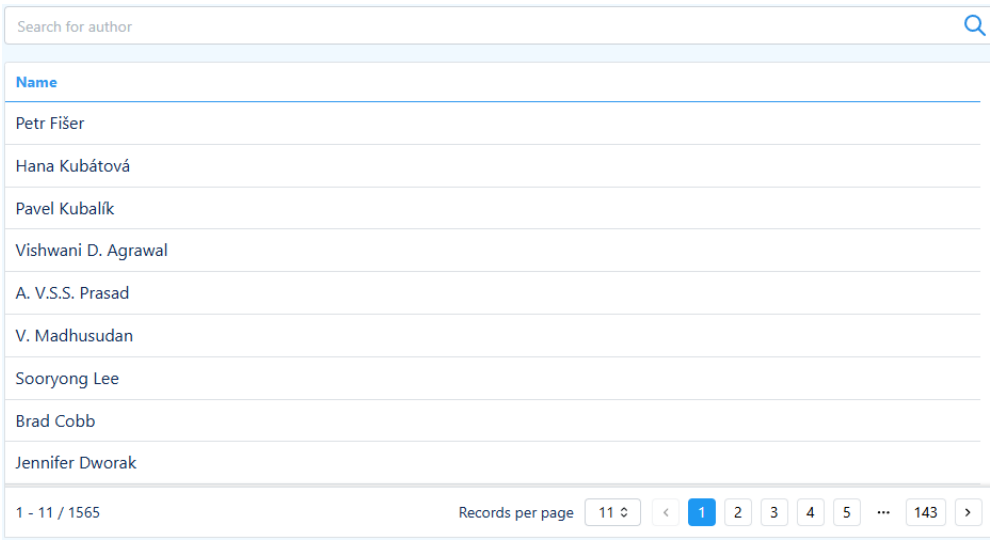
7.2.1 Seznam a filtr

Každá sekce správy publikací začíná stránkou s vyhledáváním a seznamem 7.2. Vyhledávání na těchto stránkách je podstatně jednodušší než na stránce se seznamem publikací, ale princip použité komponenty je podobný. Tuto komponentu jsem nazval `Picker` (např. `AuthorPicker`, `CategoryPicker` apod.), neboť slouží k výběru daného atributu. Proklikem na řádek Pickeru se uživatel dostane na stránku s detailem vybraného záznamu.

Picker Každý tento Picker obaluje generickou komponentu `PaginatedDataGridWithSearch`. Ta přijímá funkci (renderer), která definuje, jak se vykreslí jednotlivý řádek v seznamu výsledků, a také serverovou akci, která obdobně jako u seznamu publikací definuje logiku filtrování a vrací relevantní data.

Celý stav a logika komponenty `PaginatedDataGridWithSearch` je abstrahovaná pomocí hooku `useLoadingWithPaginatedSearch`. Samotná komponenta obsahuje jednoduché vstupní pole pro vyhledávání a tabulku implementovanou pomocí komponenty `Mantine DataTable`.

Hlavním rozdílem mezi `PaginatedDataGridWithSearch` a komponentou použitou pro seznam publikací je, že první jmenovaná využívá podstatně jednodušší vnitřní komponenty. Dále neumožňuje snadné rozšíření filtrovací akce o další parametry a nepoužívá provider komponentu pro propagaci svého stavu. Také neumožňuje perzistenci stavu filtrů. Celkově se jedná o jednodušší a uzavřenější verzi hlavního seznamu.



Obrázek 7.2 PubConf - správa publikací - vyhledávání

CategoryPicker Protože kategorie mají stromovou strukturu, nebylo možné pro jejich zobrazení použít komponentu `PaginatedDataGridWithSearch`, jako tomu je u ostatních entit.

Původně jsem pro komponentu `CategoryPicker` použil jako základ komponentu `Tree` z knihovny `Mantine`. Ta poskytuje jednoduché API a základní funkce, jako je checkbox nebo stav rozbalení. Komponenta by však měla mít i podporu pro *drag & drop*, aby uživateli umožnila měnit pořadí a hierarchii kategorií. Komponenta `Tree` navíc nepodporuje virtualizaci, a při větším počtu položek tak narážela na výkonnostní limity.

Z těchto důvodů jsem se nakonec rozhodl pro knihovnu `react-arborist`. Ta poskytuje virtualizované vykreslování, umožňuje operace jako přeskupení, mazání nebo editaci a navíc umožňuje efektivní vyhledávání v hierarchii.

Nevýhodou této knihovny je, že pro správné fungování virtualizace musí mít komponenta pevně danou výšku. Tím ztrácí možnosti plné responsivity a na menších monitorech může docházet k přetékání obsahu.

Abych přetékání zabránil, využil jsem hook `useResizeObserver`, který umožňuje sledovat rozměry vykresleného elementu. Pomocí tohoto hooku sleduji velikost rodičovského elementu komponenty `CategoryPicker` a na základě těchto údajů dynamicky měním její parametry.

Formuláře Na této stránce se také nachází formulář pro přidání záznamu. Jako základ pro tyto formuláře používám komponentu `BaseExpandableBlock`, díky níž je možné formulář zabalit, aby nezabíral zbytečně místo.

7.2.2 Detail

V detailu záznamu lze provádět akce jako úpravu a mazání. Také je zde možné vidět údaje s atributem spojené (například ISBN u časopisu). Ve spodní části stránky se vždy nachází komponenta se seznamem publikací, která ukazuje pouze publikace spojené s daným záznamem. Na pravé straně je pak příslušný Picker. Protože je rozložení pro všechny tyto detaily stejné, abstrahoval jsem ho do samostatné komponenty `DetailLayout`

Formulář na úpravu používá, podobně jako pro vytváření, `BaseExpandableBlock` a formulář pro mazání komponentu `BaseDeleteBlock`. Pro funkčnost bloku pro mazání mu stačí poskytnout serverovou akci, která provede samotné smazání požadovaného záznamu. Jako příklad uvedu blok pro mazání autora 7.1.

```
1     export default function DeleteAuthor({ authorId }: {
          authorId: bigint }) {
2       const router = useRouter();
3       return (
4         <BaseDeleteBlock
5           //deleteAuthor is server action that delets the
                author
6           onDelete={() => deleteAuthor(authorId)}
7           useActionState={useBaseActions}
8           //programatically navigates to author list page
9           onSuccess={() => router.push(Publications.
                getAuthors())}
10          actionText="Delete Author"
11          waitForActionText="Author will be deleted shortly"
12        />
13    );
14  }
```

Výpis kódu 7.1 Blok pro mazání autorů

7.3 Správa uživatelů

Obě stránky — správa uživatelů a správa uživatelských žádostí — využívají stejné komponenty jako sekce správy publikací. Klíčovou roli zde opět hraje komponenta `PaginatedDataGridWithSearch`, doplněná o specifickou funkci pro vykreslování řádků.

Zod Odlišností je přístup k práci s formuláři. Na těchto stránkách provádím validaci formulářů výhradně na straně serveru za pomoci validační knihovny `Zod`. Tento přístup je vhodný pouze pro jednoduché formuláře, u kterých není třeba kontrolovat jejich stav a dynamicky měnit jejich hodnoty. Například pro formuláře na vytváření publikací a konferencí by tento přístup byl nevhodný.

Zod umožňuje definovat tzv. „schéma“, které musí data z formuláře splňovat. Pokud data neodpovídají definici schématu, server vrátí objekt s chybovými hláškami. Tyto chyby pak zobrazuji u jednotlivých polí nebo pomocí vyskakovacího upozornění.

createFormActionWithZod Pro zjednodušení implementace takových formulářů jsem vytvořil obalovou (wrapper) funkci `createFormActionWithZod`, která abstrahuje logiku validace a zpracování formulářových dat.

Tato funkce přijímá dvě hodnoty — funkci, která dále pracuje s validními daty, a schéma definované pomocí Zodu. Obalová funkce nejprve data zpracuje a zvaliduje. Pokud validace selže, vrací objekt s chybami. Pokud je validace úspěšná, předá zpracovaná data dál do vložené funkce.

Obalová funkce také k výsledku vložené akce přidá původní stav formuláře. To je důležité, protože `React` po odeslání formuláře stav automaticky resetuje. Takto jsou původní data dostupná, kdyby bylo potřeba zachovat původní stav formuláře. Implementaci této obalové funkce ukazuje výpis 7.2.

```
export default function createFormActionWithZod
  <T extends z.ZodType, TCustomArgs extends any[] = []>(
    schema: T,
    action: FormAction<T, TCustomArgs>,
    options?: CreateFormActionWithZodOptions
  ) {
  return (...args: TCustomArgs) => {
    return async (
      state: ActionState<T> | null | undefined,
      formData: FormData
    ): Promise<ActionState<T>> => {
      // Processes the data from FormData object.
      const processedSchema = zfd.preprocessFormData(
          formData);
      // Validates the data with provided Zod schema.
      const data = schema.safeParse(processedSchema);
      // If validation fails, returns object with errors.
      if (!data.success)
        return {
          state: 'error',
          errors: data.error.formErrors.fieldErrors,
          formState: processedSchema
        } as const;
      // Calls provided action with parsed data and other
          arguments.
      const result = await action(data.data, state, ...args)
          ;
      // Returns result of the action and adds parsed data.
      if (
        result.formState === undefined &&
        (options?.injectPrevFormState === undefined ||
```

```
29            !!options?.injectPrevFormState)
30        ) {
31          return {
32            ...result,
33            formState: data.data
34          };
35        }
36        return result;
37      };
38    };
39  }
```

Výpis kódu 7.2 Pomocná funkce pro validaci formulářů

7.4 Nastavení účtu

Formulář `UserSettingsForm` s uživatelským nastavením využívá také serverovou validaci a pomocnou funkci `createFormActionWithZod`.

Informace o uživatelově nastavení používají různé komponenty napříč aplikací. Umístil jsem je proto do globální provider komponenty `UserSettingsProvider`. Díky tomu mohou k těmto datům přistupovat i komponenty vykreslované na klientovi.

7.5 Shrnutí změn v databázi

Některé změny v databázi jsem již popsal, například při implementaci autentizace. V této sekci shrnu zbývající úpravy, které jsem v rámci vývoje provedl.

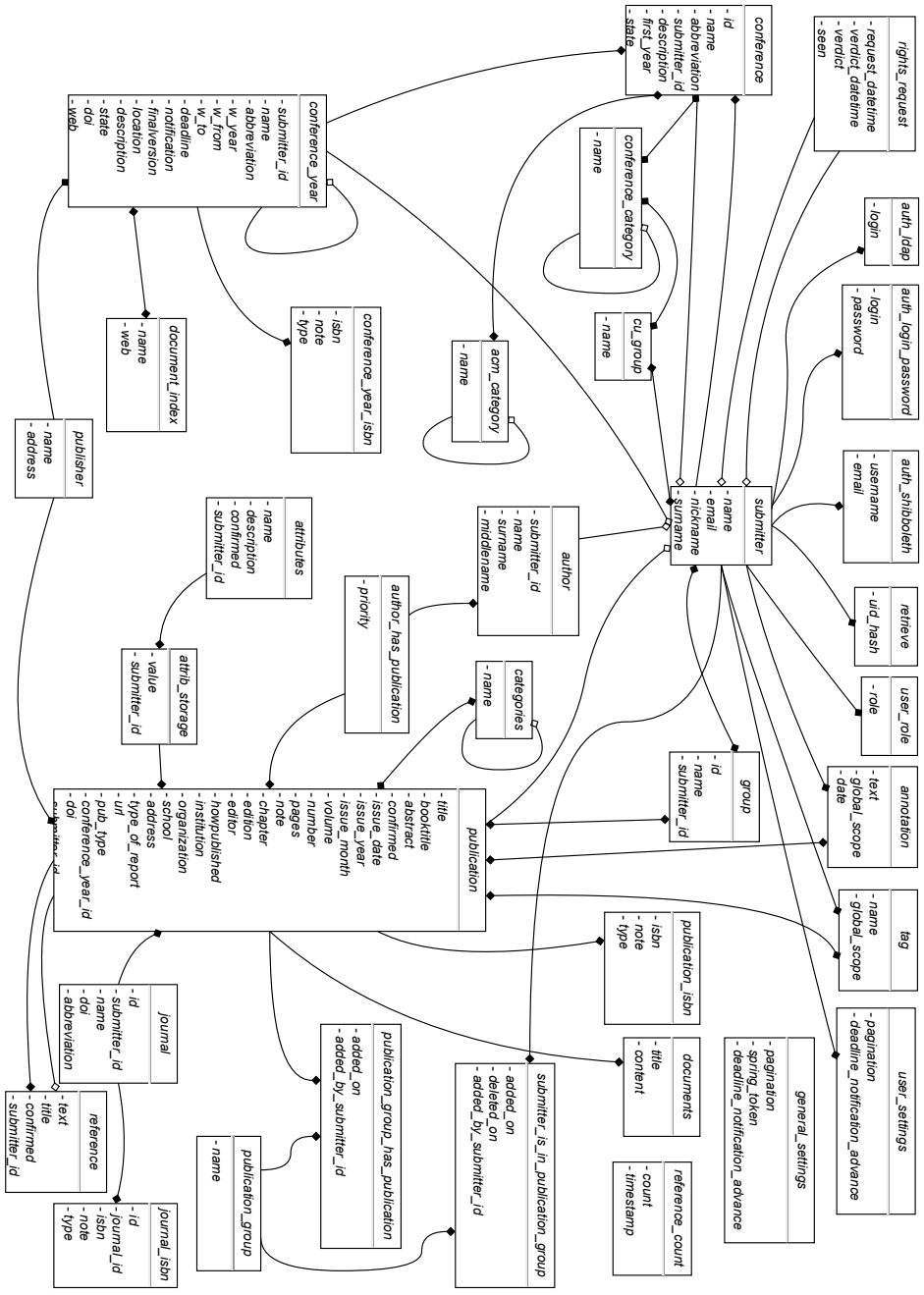
Většina změn spočívala v přejmenování existujících tabulek s cílem zvýšit konzistenci napříč celým schématem. Kromě toho jsem do vybraných relací doplnil integritní omezení. Nejčastěji šlo o tabulky, které vznikly dekompozicí M:N vztahů, u kterých jsem nově nastavil vazbu typu `ON DELETE CASCADE`.

Přidání nových atributů a tabulek je v diagramu databázového modelu zvýrazněno modrou barvou (viz obr. 7.3). Pro srovnání přikládám i původní model databáze 7.4, který jsem převzal z práce [2] a který sloužil jako výchozí stav.

Na kompletní historii změn lze nahlédnout pomocí migračních souborů, umístěných ve složce projektu `/prisma/migrations`.

Obrázek 7.3 Nové databázové schéma

Poznámka: Pro zlepšení čitelnosti jsou ze schématu vynechány tabulky pro dekompozici M:N vztahů. Také jsme vynechali relace, které reprezentují vztah „vytvořil“ a „editoval“ mezi tabulkou `Submitter` *a ostatními tabulkami.*



Obrázek 7.4 Původní databázové schéma [2]

Kapitola 8

Uživatelské testování

Dle zadání jsem provedl uživatelské testování na základě uživatelských scénářů. Testování se účastnili čtyři studenti magisterského programu Fakulty informačních technologií.

Po vyplnění úvodního dotazníku (viz A.1) tester prošel sadou připravených scénářů, které mu byly prezentovány moderátorem. Průchod scénářem byl mnou sledován a následně vyhodnocen. Na konci testování byl testerem vyplněn závěrečný dotazník s doplňujícími otázkami (viz A.2).

Výsledkem testování je soupis chyb, návrhy na změny systému a nové požadavky.

U testování jsem se zaměřil zejména na aspekty, jako je vyhledávání, navigace v aplikaci a dostupnost informací. Proto většina scénářů nezačíná přímo na stránce, kde je daná akce dostupná, a po dokončení dané akce vyžaduji, aby tester její efekt ověřil.

Jelikož každá stránka administrace obsahuje stejné rozložení uživatelského rozhraní a tedy její použití se mezi jednotlivými stránkami pro správu „Authors“, „Attributes“, „Categories“, „Publishers“ a „Journals“ neliší, nebudu opakovat testování obdobných částí pro každou stránku administrace a zaměřím se pouze na prvky, které jsou z hlediska uživatelského rozhraní odlišné.

Scénáře - autoři

1. **SA přidat do oblíbených** - Nyní se nacházíte na stránce s publikacemi a chtěli byste si vyhledat publikaci vašeho oblíbeného autora ”Alana Klementa Mishchcenka”. Najděte jakoukoliv publikaci, která má jako autora ”Alana Klementa Mishchenka”nebo také jen ”Alan Mishchenko”a přidejte jí mezi své oblíbené (starred).
   očekávané kroky:
   - Uživatel zadá do vyhledávacího pole celé nebo část autorova jména.
   - Uživatel v seznamu publikací najde řádek s autorovým jménem.
   - Uživatel klikne na hvězdičku u daného řádku.

2. **SA proklik z publikace a editace** - Nyní se koukáte na tabulku s publikacemi, která obsahuje publikace od autora "Alan Klement Mishchenko". Zjistili jste, že Alanovi chybí prostřední jméno. Přidejte tomuto autorovi prostřední jméno "Klement". Zkontrolujte, že se změna skutečně provedla.
   očekávané kroky:
   - Uživatel klikne na autorovo jméno u vyhledané publikace.
   - Uživatel Otevře nabídku s názvem "Edit".
   - Uživatel do pole s označením "Middlename"napíše Klement a formulář uloží.
   - Uživatel zkontroluje, že se změnilo jméno na začátku stránky.
3. **SA vytváření** - Nyní se nacházíte na domovské stránce. Chtěli byste vytvořit autora, který bude reprezentovat vaší osobu a spárovat ho s vaším účtem (účet na který jste momentálně přihlášení). Vytvořte nového autora s vaším jménem a příjmením a spárujte ho s momentálně přihlášeným uživatelem. Ověřte, že došlo k vytvoření autora a že je spárován s vaším účtem.
   očekávané kroky:
   - Uživatel klikne v postranním panelu na záložku "Authors".
   - Uživatel vyplní formulář svým jménem a příjmením.
   - Uživatel přiřadí k tomuto autorovi právě přihlášeného uživatele kliknutím na tlačítko "Me", případně vyhledá uživatele v seznamu.
   - Uživatel uloží formulář a zkontroluje, že po přesměrování se nachází na stránce autora s jménem a příjmením, které do formuláře zadal a že v poli pro spárovaného uživatele je přiřazený právě přihlášený uživatel.
4. **SA mazání** - Nyní se nacházíte na stránce správy autorů. Chtěli byste smazat autora, kterého jste v předchozím scénáři vytvořili. Najděte tohoto autora a smažte ho. Ověřte, že ke smazaní skutečně došlo.
   očekávané kroky:
   - Uživatel vyhledá v seznamu všech autorů předešle vytvořeného autora, zadáním jeho jména do vyhledávacího pole.
   - Uživatel otevře stránku s vyhledaným autorem a klikne na tlačítko "Delete".
   - Uživatel potvrdí svoji volbu ve vyskakovacím okně a počká až se autor odstraní.
   - Uživatel se pokusí vyhledat autora v seznamu, aby ověřil, že autor už se v něm nenachází.

Scénáře - kategorie

1. **SC hledání kategorie and vytváření podkategorie** - Nyní se nacházíte na stránce správy kategorií publikací a chtěli byste přidat podkategorii ke kategorii s názvem "Glitches". Najděte kategorii s názvem "Glitches"a

vytvořte její podkategorii s názvem ”Bugs”. Ověřte, že k vytvoření a že jde skutečně o podkategorii kategorie ”Glitches”.
očekávané kroky:

- Uživatel přejde pomocí postranní navigace na stránku se správou kategorií.
- Uživatel vyhledá pomocí vyhledávacího pole v seznamu kategorií tu s názvem ”Glitches”.
- Uživatel otevře stránku s detailem této kategorie. Do pole s názvem ”Add subcategory”zadá ”Bugs”a klikne na tlačítko ”Add”.
- Uživatel vyhledá v postranním seznamu kategorií kategorii ”Glitches”a zkontroluje, že Bugs se nachází pod touto kategorií.

Scénáře - časopisy

1. **SČ vytváření** - Nyní se nacházíte na stránce se správou časopisů. Vytvořte nový časopis s názvem ”Přírodní katastrofy”. K časopisu také přidejte jedno ISBN a jedno ISSN. Jako ISBN a ISSN můžete použít jakýkoliv neprázdný řetězec. Zkontrolujte, že došlo k vytvoření časopisu a že má všechny vyplněné atributy.
   očekávané kroky:

   - Uživatel vyplní zadané údaje do formuláře s názvem ”Add journal”a formulář uloží.
   - Uživatel zkontroluje, že u detailu časopisu se ukazují ISSN i ISBN.

2. **SČ editování** - Nacházíte se na stránce administrace časopisu vytvořeného v předchozím scénáři. Odstraňte z časopisu údaj o ISSN a ke stávajícímu ISBN přidejte poznámku ”neověřeno”.
   očekávané kroky:

   - Uživatel otevře formulář s nápisem ”Edit journal”.
   - Uživatel najde ve formuláři část s ISSN a klikne na křížek v pravém horním rohu.
   - Uživatel do pole ”note”k části ISBN přidá poznámku a formulář uloží.
   - Uživatel ověří, že se data změnila.

Scénáře - administrace uživatelů

1. **SAU - registrace** - Nyní se nacházíte na stránce se seznamem uživatelských účtů. Váš kamarád Marek by chtěl použít aplikaci Pubconf. Vytvořte nový uživatelský účet pro přihlášení s emailem a heslem. Jako email použijte ”`marek.langos@email.com`”. Do polí jméno a příjmení vyplňte ”Marek”a ”Langoš”. Ověřte, že k došlo k vytvoření účtu a že nový uživatel má správné jméno a příjmení.
   očekávané kroky:

- Uživatel přejde na stránku registrace uživatelů pomocí tlačítka ”register user”.
- Uživatel vyplní formulář podle zadaných údajů a formulář odešle pomocí tlačítka ”register”
- Uživatel vyhledá nově vytvoření účet v seznamu uživatelů pomocí vyhledávacího pole.

2. **SAU - editace** - Nyní se nacházíte na stránce se seznamem uživatelských účtů. Chtěli byste Markově účtu přidělit možnost přidávat a editovat publikace. Vyhledejte jeho účet a přidejte mu roli ”Submitter”. Ověřte, že role byla přidána.
očekávané kroky:

   - Uživatel vyhledá zmíněný účet pomocí vyhledávacího pole.
   - Uživatel klikne na tlačítko editace.
   - Uživatel ve formuláři zaškrtne roli ”Submitter”a dá uložit.
   - Uživatel vyhledá daný účet a ověří, že ve sloupci ”Level”je role ”Submitter”

3. **SAU - odhlášení a přihlášení** - Odhlašte se z účtu, ke kterému jste momentálně přihlášeni. Po odhlášení se přihlašte na Markův účet, který jste v předešlém scénáři vytvořili, pomocí zadaného emailu a hesla. Ověřte, že jste přihlášeni.
očekávané kroky:

   - Uživatel v pravém horním rohu klikne na tlačítko ”Sign out”.
   - Uživatel zadá přihlašovací údaje do polí email a password.
   - Uživatel ověří, že v horní navigaci vidí jméno ”marek.langos”.

4. **SAU - žádost o uživatelská práva** - Nyní se nacházíte na domovské stránce. Chtěli byste zažádat o přidělení adminských práv. Přejděte do nastavení vašeho účtu a pošlete žádost o zvýšení práv na admin.
očekávané kroky:

   - Uživatel v horní navigaci klikne na tlačítko se jménem svého účtu.
   - Uživatel kline na tlačítko ”request admin rights”.
   - Uživatel ověří, že se tlačítko změnilo.

5. **SAU - schválení uživatelské žádosti** - Nyní se nacházíte na domovské stránce administrátorského účtu. Přejděte na stránku s uživatelskými žádostmi a vyhledejte žádost vytvořenou v předešlém scénáři (od uživatele Marek Langoš). Žádost potvrďte. Ověřte, že Marek obdržel roli administrátora.
očekávané kroky:

   - Uživatel přejde pomocí postranního menu na stránku se žádostmi.
   - Uživatel se podívá na žádosti a u řádku s uživatelem ”marek.langos”klikne na tlačítko ”approve”.

- Uživatel v seznamu účtů vyhledá Markův účet a zkontroluje, že ve sloupci ”Level”je role ”admin”.

6. **SAU - uživatelské nastavení účtu** - Nyní se nacházíte na stránce s uživatelským nastavení. Chtěli byste upravit základní chování tabulek s publikacemi. Změňte zobrazení tabulek z pohledu ”Citation”na pohled ”Full Row”, dále změňte základní stav sekce s dodatečnými informacemi z ”Collapsed”na ”Expanded”a nakonec nastavte počet vylistovaných položek z 25 na 10. Ověřte, že došlo ke změně konfigurace.
   očekávané kroky:
   - Uživatel v pravé části nastavení profilu změní přepínače dle zadání.
   - Ve spodní části formuláře změní počet položek na hodnotu 10 a uloží formulář.
   - Uživatel přejde na stránku s publikacemi a ověří, že došlo ke zmíněným změnám.

8.1 Výsledky testování

V této sekci provedu vyhodnocení záznamů a poznámek z testování. Podrobně rezeberu jeden průchod scénářem s poznámkami pro každý jednotlivý případ.

Celkově se při testování objevilo jen několik chyb, přičemž většina z nich se opakovala u všech participantů. Zbytek zhodnocení proto shrnuji pomocí tabulky s chybami a návrhy možných řešení.

Níže jako ukázku vypíši komentáře k prvnímu testování scénář po scénáři.

Scénáře – autoři

1. **SA – Přidat do oblíbených**
   Scénář proběhl bez problémů.
2. **SA – Proklik z publikace a editace**

   - Tester si nevšiml možnosti prokliku na autora přes odkaz u vyhledané publikace a místo toho přešel do správy autorů, kde autora ručně vyhledal.
   - Tester omylem vyplnil i pole pro spárování uživatele, protože si spletl tlačítko „Me“ s tlačítkem pro uložení. Následně nemohl přijít na to, jak hodnotu z pole odstranit.

3. **SA – Vytváření**

   - Tester nejprve kliknul na nastavení účtu, poté rychle přešel do správy autorů. Tentokrát nepoužil tlačítko „Me“, ale svůj účet vyhledal manuálně. Následně si uvědomil, že mohl tlačítko použít.

4. **SA – Mazání**

   - Tester chvíli tápal, jak aktivovat tlačítko „Smazat“.

Scénáře – kategorie

1. **SC – Hledání kategorie a vytváření podkategorie**

   - Tester bez problémů kategorii nalezl a vytvořil její podkategorii.
   - Tester nevěděl, jak si ověřit, že nově vytvořená kategorie „Bugs“ je skutečně podkategorií „Glitches“. Očekával, že se to projeví v drobečkové navigaci ve tvaru `Publications/Categories/Glitches/Bugs`.

Scénáře – časopisy

1. **SČ – Vytváření**

   - Tester přepínal mezi možnostmi ISSN a ISBN a chtěl přidat obě varianty. Domníval se, že se přepnutím volby pole přemění a předchozí hodnota zůstane přiřazena správně. Vůbec si nevšiml tlačítka pro přidání dalšího záznamu. Tuto funkcionalitu objevil až při editaci.
2. **SČ – Editace**

   - Scénář proběhl podle očekávání.

Scénáře – administrace uživatelů

1. **SAU – Registrace**

   - Tester byl velmi frustrovaný z validace hesla. Po každé úpravě se objevila nová chyba. Jinak registrace proběhla úspěšně.
2. **SAU – Editace**

   - Scénář proběhl podle očekávání.
3. **SAU – Odhlášení a přihlášení**

   - Tester měl potíže si vybavit heslo, které zvolil. Jednalo se spíše o problém návrhu scénáře než aplikace.
4. **SAU – Žádost o uživatelská práva**

- Tester našel tlačítko pro odeslání žádosti, ale nebyl si jistý, zda se akce skutečně provedla.

5. **SAU – Schválení žádosti**

- Scénář proběhl dle očekávání.

6. **SAU – Uživatelské nastavení účtu**

- Uživatel si nebyl jistý, k čemu nastavení slouží a co ovlivňuje. Neviděl rozdíl mezi jednotlivými pohledy a neustále se snažil přejít na stránku s publikacemi pomocí levého postranního menu.

Pokračování vyhodnocení Ostatní testování jsem zpracoval podobným způsobem. Souhrn nalezených problémů je uveden v následující tabulce 8.1.

Vážnost chyby hodnotíme na třístupňové škále: *Nedostatek*, *Chyba* a *Kritická chyba*.

- **Nedostatek** – chyba mírně ztěžuje práci nebo nemá výrazný dopad.
- **Chyba** – nebrání chodu systému, ale komplikuje uživatelskou zkušenost.
- **Kritická chyba** – brání dokončení úkolu nebo použití systému a musí být opravená.

8.2 Shrnutí

Testování proběhlo hladce a všem participantům se nakonec podařilo úspěšně dokončit všechny scénáře. Nenalezl jsem žádné fundamentální chyby, které by uživateli bránily v dokončení požadovaných úkonů.

Na druhou stranu jsem identifikoval řadu drobných nedostatků, které snižují intuitivnost použití systému. Ve většině případů šlo o problémy spojené s umístěním nebo označením interaktivních prvků.

Pozitivní je, že chyby, které jsem během testování objevil, nejsou náročné na opravu. Většinu z nich jsem proto rovnou v rámci této práce odstranil nebo pro ně navrhl konkrétní řešení.

#	Popis chyby	Scénář	Vážnost	Návrh řešení
1	Hledání neodpouští uživateli překlepy.	Různé, nejvíce 1. SA	Nedostatek	Mohl bych použít PostgreSQL „full-text“ vyhledávání s metrikou podobnosti.
2	Matoucí popis tlačítka pro potvrzení změn. Uživatelé očekávají „Save“.	2. SA a 2. SČ	Nedostatek	Přejmenovat tlačítko u editace na „Save“.
3	Chybějící ikonky u navigace.	Různé	Nedostatek	Přidat ikonky, které budou korespondovat s ikonami v jiných částech aplikace.
4	Nevýrazná horní navigace a chybějící proklik u kořene postraní navigace.	Různé	Chyba	Zvětšit a zvýraznit horní navigaci. Udělat zaštitující prvky jako odkaz na danou sekci. Vlastnost zavřít navigaci mohu úplně odebrat.
5	Špatná poloha a nedostatečná funkce tlačítka „Me“ při výběru autora.	3. SA	Nedostatek	Talčítko by mělo vyplnit nejen účet se kterým nového autora spárovat, ale i ostatní údaje jako jméno a příjmení. Také by se mělo přesunout dále od tlačítka uložit.
6	Aktivování tlačítka pro smazání přes checkbox je zbytečné.	4. SA	Nedostatek	Odeberu potřebu zaškrtnout políčko pro aktivaci tlačítka „Smazat“.
7	Špatně navržený způsob přidání ISBN a ISSN k časopisu.	1. SČ	Kritická chyba	Je potřeba zvýraznit tlačítko pro přidání dalšího ISBN/ISSN nebo ho zjednodušit.
8	Zobrazení poznámky u ISSN a ISBN. Není jasné, že se jedná o poznámku.	2. SČ	Chyba	Označím poznámku ikonkou nebo ji vizuálně odliším.
9	Při zadávání hesla se ukazují požadavky postupně. To je matoucí.	1. SAU	Nedostatek	Pokud dojde k nesplnění podmínek při zadávání hesla, zobrazit všechny požadavky najednou.
10	Při odeslání požadavku o přidělení role se neukáže žádné potvrzení.	4. SAU	Chyba	Po zmáčknutí tlačítka přidám oznámení o provedení akce.

Tabulka 8.1 Přehled nalezených chyb a nedostatků při uživatelském testování

Kapitola 9
Návrhy na vylepšení

Přestože se nám podařilo aplikaci dokončit prakticky se všemi požadovanými funkcionalitami, existují oblasti, které by bylo vhodné dále rozvíjet a optimalizovat.

DevOps Navržená konfigurace DevOps je funkční a spolehlivá. Přesto zde existuje prostor pro optimalizaci, zejména v oblasti využití cache a artefaktů v rámci GitLab pipeline (viz [25]). Vylepšení by mohlo zkrátit dobu sestavení a zvýšit efektivitu automatizovaných procesů.

Reference V aktuální verzi aplikace chybí funkcionalita pro přidávání a schvalování referencí mezi publikacemi, která byla přítomná ve starší verzi systému. Tato funkcionalita je poměrně komplexní a ani v původní implementaci nebyla optimálně navržena. Do budoucna by se tedy vyplatilo její návrh přepracovat a zaměřit se na samostatný vývoj této části.

Vyhledávání Vyhledávání představuje klíčovou funkcionalitu celé aplikace, a proto je žádoucí jej dále vylepšovat. Na základě zpětné vazby z uživatelského testování by bylo vhodné implementovat podporu pro nepřesné vyhledávání založené například na editační vzdálenosti. Zároveň by bylo vhodné doplnit pokrytí této oblasti robustnějšími automatizovanými testy.

Kapitola 10
Závěr

Cílem této práce bylo přepracovat původní aplikaci *PubConf*, určenou pro správu konferencí a publikací, do moderního technologického stacku odpovídajícího aktuálním požadavkům Fakulty informačních technologií ČVUT. Při reimplementaci jsem kladl důraz na využití současných technologií, jako jsou Next.js, TypeScript, Prisma a Docker, které umožňují sjednotit frontend i backend do jedné fullstack aplikace. Výsledná architektura tak vyniká modularitou, přehledností a dobrou rozšiřitelností.

Součástí řešení bylo také zabezpečení přístupu prostřednictvím autentizační služby EntraID a návrh efektivního vývojového a nasazovacího procesu s využitím nástrojů DevOps, konkrétně GitLab CI/CD. V práci jsem detailně popsal implementaci klíčových částí systému a vybraných technologií. Dále jsem připravil a přiložil uživatelskou příručku k aplikaci.

Důležitou částí projektu bylo uživatelské testování, jehož cílem bylo ověřit srozumitelnost a použitelnost navrženého rozhraní. Na základě získané zpětné vazby jsem identifikoval a odstranil několik drobných nedostatků, které by mohly uživatelům komplikovat práci se systémem.

Na závěr jsem shrnul možná vylepšení a návrhy pro další rozvoj aplikace, které mohou sloužit jako podklad pro budoucí práce navazující na tento projekt.

Příloha A

Testování

Úvodní formulář

Informovaný souhlas účastníka testování

Tento formulář slouží k informování účastníka o účelu a průběhu testování aplikace PubConf2, která je součástí vývoje v rámci diplomové práce.

Prosíme, přečtěte si následující body:

1. Účast na testování je **dobrovolná** a můžete ji kdykoli bez udání důvodu ukončit.
2. Data získaná během testování (např. zpětná vazba, poznámky, měření času) budou použita výhradně pro účely této práce a nebudou nikde veřejně publikována v identifikovatelné podobě.
3. Testování bude probíhat formou vykonávání předem připravených úkolů v prostředí testované aplikace.
4. Po skončení testu budete požádáni o vyplnění krátkého dotazníku nebo o slovní zhodnocení.

Souhlas účastníka:

Souhlasím s účastí na uživatelském testování aplikace a se zpracováním anonymizovaných údajů pro účely této práce.

Jméno účastníka: .....

Datum: .....

Podpis: .....

(Nepovinné) Další údaje:

- Role / znalost testované domény: .....
- Věk: .....
- Zkušenost s podobnými systémy: [ ] žádná [ ] základní [ ] pokročilá

Obrázek A.1 Úvodní formulář

Závěrečný formulář

1. Jak se vám aplikace líbila?
2. Byla navigace v aplikaci dobře použitelná? Byly její prvky dobře rozmístěny?
3. Chyběla vám při plnění scénářů nějaká funkcionalita, kterou byste od aplikace očekávali, nebo která by vám plnění úkolu zjednodušila?
4. Přišel vám výpis údajů o publikaci přehledný?
5. Přišly vám formuláře pro správu údajů jako jsou přidání autora, nebo editace časopisu přehledné a dobře umístěné?
6. Byla z názvů a popisek jednotlivých tlačítek jasná jejich funkce? Napadá vás nějaké tlačítko, ke kterému by se hodilo dát vysvětlivku?

Obrázek A.2 Závěrečný formulář

Příloha B

Uživatelská příručka

Czech technical university
Faculty of information technology

PubConf
User Documentation

May 3, 2025

David Kocman, Tomáš Vondra

Creating a publication

For the purpose of creating a publication a form is available, that changes its content based on the chosen type of publication. The first part of the form that contains some general and mandatory fields can be seen of the Figure 1.

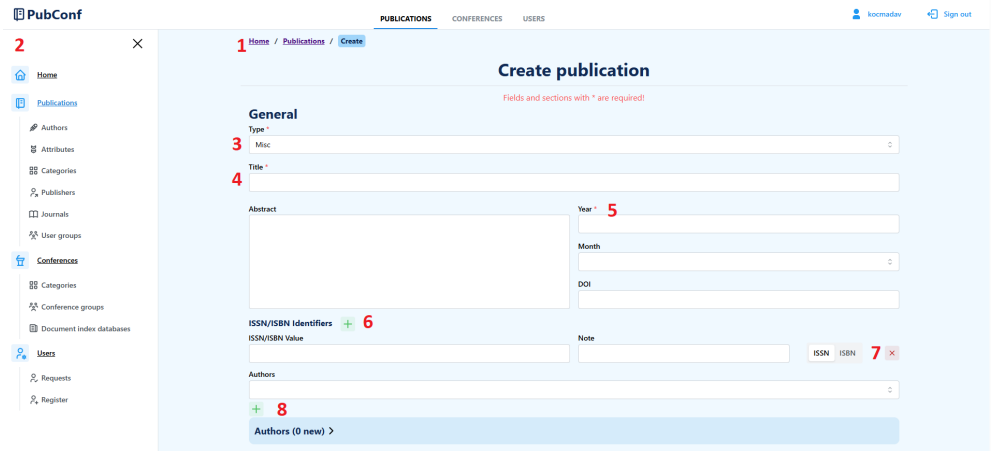


Figure 1 Form for creating a publication.

Figure 1 contains these labels:

1. Breadcrumb navigation (ubiquitous).
2. Side menu.
3. Type of publication – selecting the type changes the input fields that are shown to the user. Required.
4. Title of the publication – required.
5. Year of the publication – required.
6. Add new ISSN/ISBN identifier – maximum number of identifiers is 3.
7. Select for the type of identifier and a button for deleting an identifier entry – the user defines the identifier type.
8. Authors – required for all type except for `Misc` and `Manual`, in `Proceedings` the author select is not shown at all.

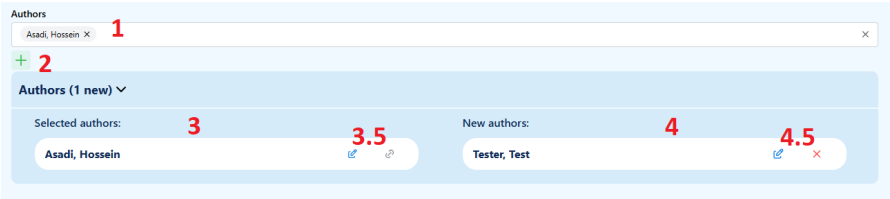


Figure 2 Selection and management of authors from the publication form.

Figure 2 contains the administration of authors from the publication form:

1. The author Select.
2. Add new author button that opens the modal for creation.
3. The list of selected authors. These entries contain action buttons for editing the author (opens the edit modal) or opening a modal with the author's related publications (3.5).
4. The list of new authors. These entries contain action buttons for editing the new author (opens the edit modal) or deleting the new author entry (4.5).

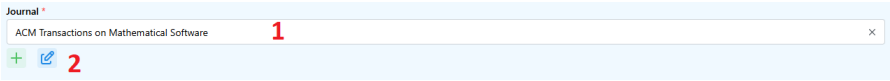


Figure 3 Selection and administration of a related entity from the publication form.

The Figure 3 shows the general functionality of the administration of related entities:

1. The list of entities to select.
2. The action buttons for editing or creating a new entity. The edit button is not visible unless an entity is selected. Both buttons open a modal with a form.

Figure 4 shows the detail of a new related entity created from the publication form:

1. The action buttons – now a delete button can be seen to delete the new record.
2. Details of the newly created entity.

Figure 5 shows the behavior of the `Booktitle` and the `Publisher` fields when the *inproceedings* type is selected.

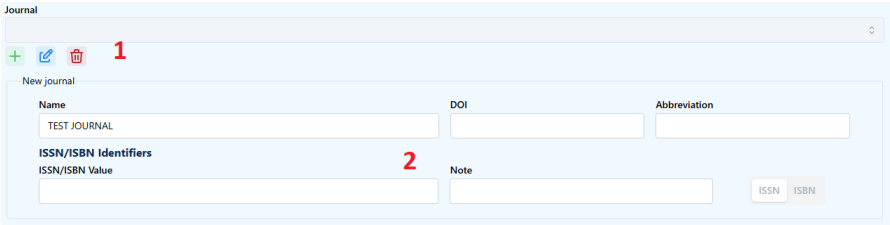


Figure 4 Administration and detail of a new entity from publication form.

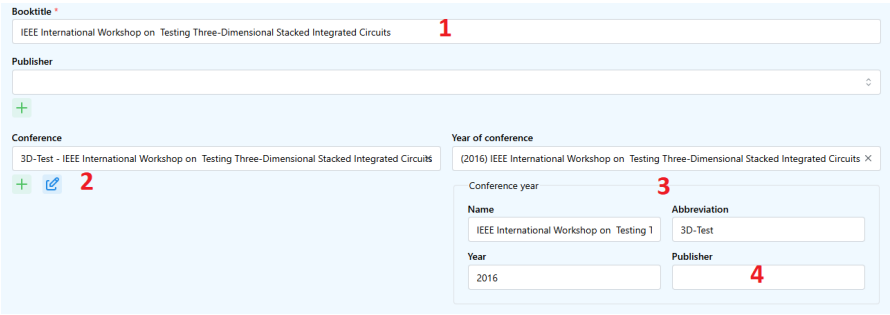


Figure 5 The behavior of the `Bootktitle` and the `Publisher` fields when *inproceedings* type is selected.

1. The `Booktitle` field is automatically filled with the name of the conference year.
2. Conference list and action buttons to create a new conference or edit the conference (e.g. create new years, rename).
3. The list of conference years of the selected conference and its detail.
4. When the year has an assigned publisher, the publication's publisher is overridden with this one.

Figure 6 shows the selection input of the publication categories and their administration interface:

1. Search bar.
2. The list of categories to choose from.
3. When hovering over a category, some action buttons are shown. These buttons open a modal with a form to create a subcategory, edit this category or delete the category.
4. Add a new category button that opens a modal with a form.

Figure 7 contains the two hidden parts of the publication form:

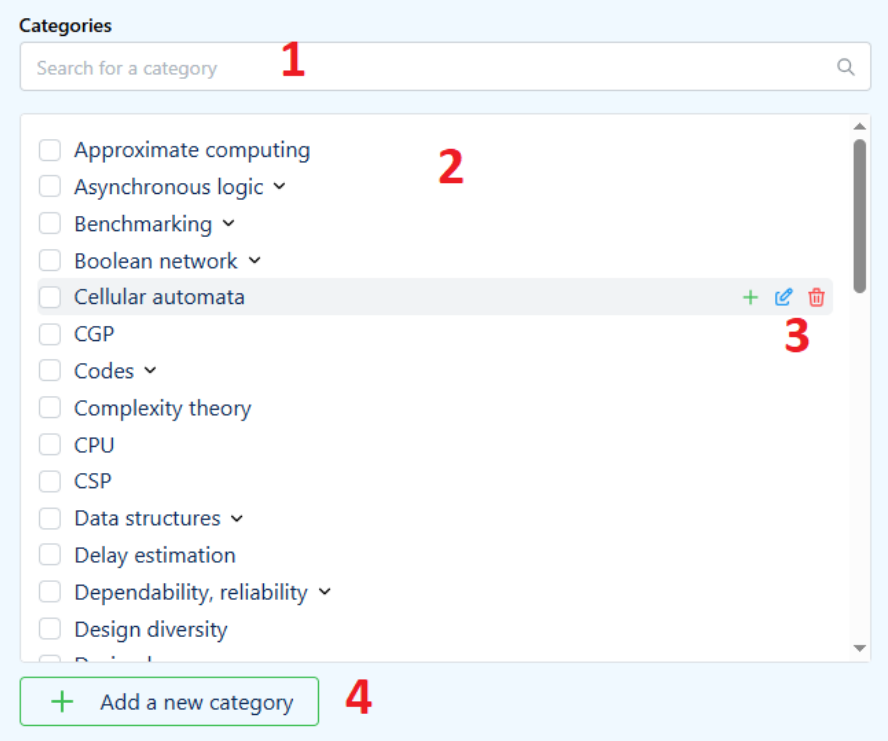


Figure 6 Selection and administration of publication categories. Categories are the only entity for which any changes are committed directly to the database.

Figure 7 Second and third parts of the form that are hidden by default.

1. Optional arguments section – this section changes its contents based on the chosen type. It contains fields that are not mandatory for the publication entry and a file input.
2. Further attributes – this part of the form contains a list of further attributes that are used for an extension of already existing attributes.
3. Submit button – this button validates the form inputs and then creates a publication and uploads any new files.

Figure 8 illustrates the view on the optional arguments of the publication and file input:

1. The optional arguments.

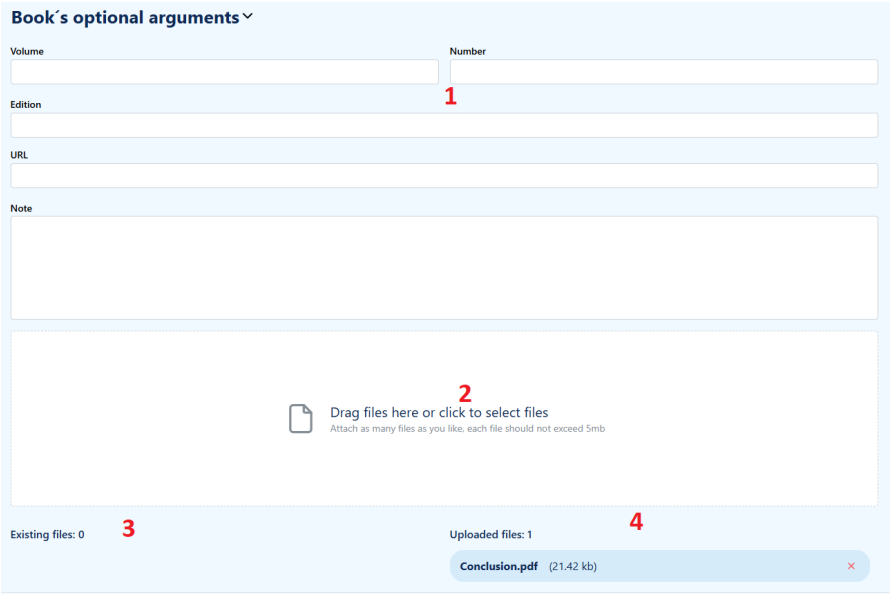


Figure 8 The view of the second part of the publication form.

2. The file input.
3. List of already existing files – when a publication is edited, any files associated with this publication that are already uploaded are shown here.
4. List of new files to be uploaded.

Figure 9 Further attributes and their administration.

Figure 9 illustrates the view on the third part of the publication form – further custom attributes:

1. Add a new attribute button that opens a modal with a form.
2. The input for the value of this attribute for this publication.
3. Delete or edit an attribute – again, opens a modal with a form.

A publication record can also be added to the database with the help of citation definition or imported from Springer.
Figure 10 shows the import by a citation definition modal:

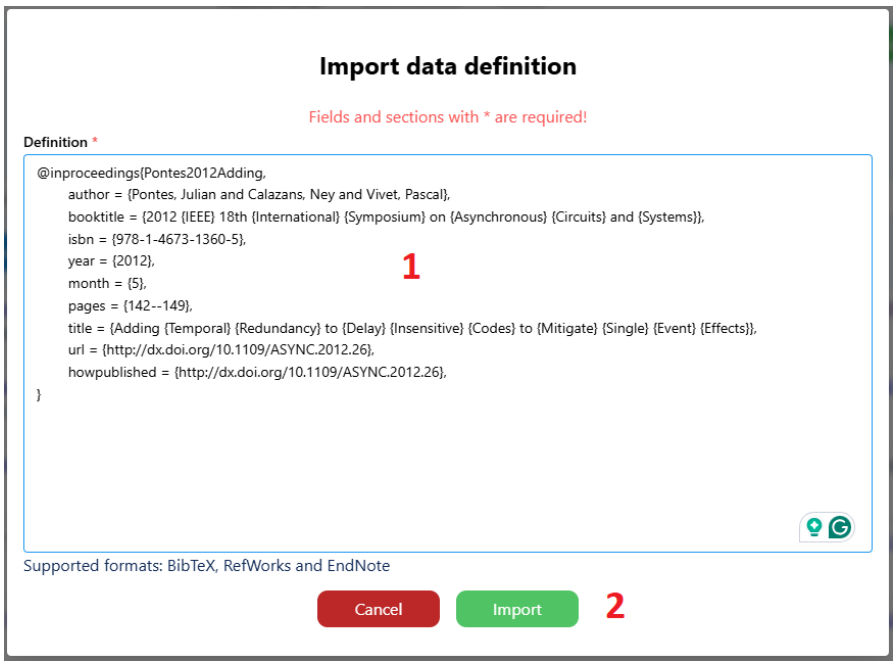


Figure 10 Import publication by a citation definition.

1. Textarea for the raw definition to be pasted in.
2. After pasting the definition the "Import" button checks for any errors in the string and fills the publication form with the provided attributes.

After importing, the raw definition or Springer response can be seen on the top of the publication form. When the definition or response contains any warnings, the warning messages are rendered below the raw definition. Figure 11 shows this behavior:

1. The raw definition/returned data. Here are also any warnings rendered.
2. The pre-populated fields with the corresponding values from the input attributes.

The modal for importing the publication from Springer can be seen on Figure 12:

1. When the user has no API key set then this error message will be displayed.
2. The DOI, ISSN or ISBN identifier of the article.
3. In this textarea the returned data will be displayed for the user to review.
4. The "Try" button sends the request to the Springer API and fills the textarea with a response.

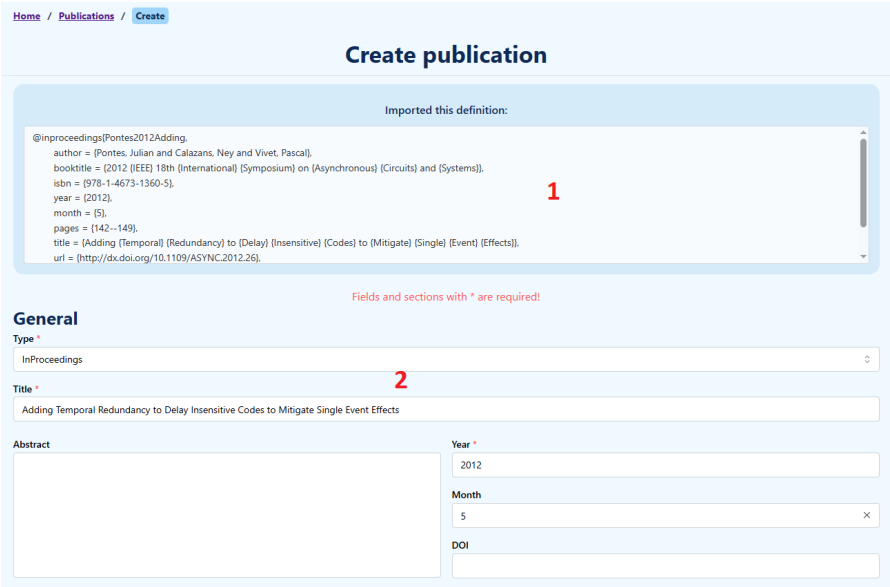


Figure 11 The raw definition can be seen at the top of the publication form.

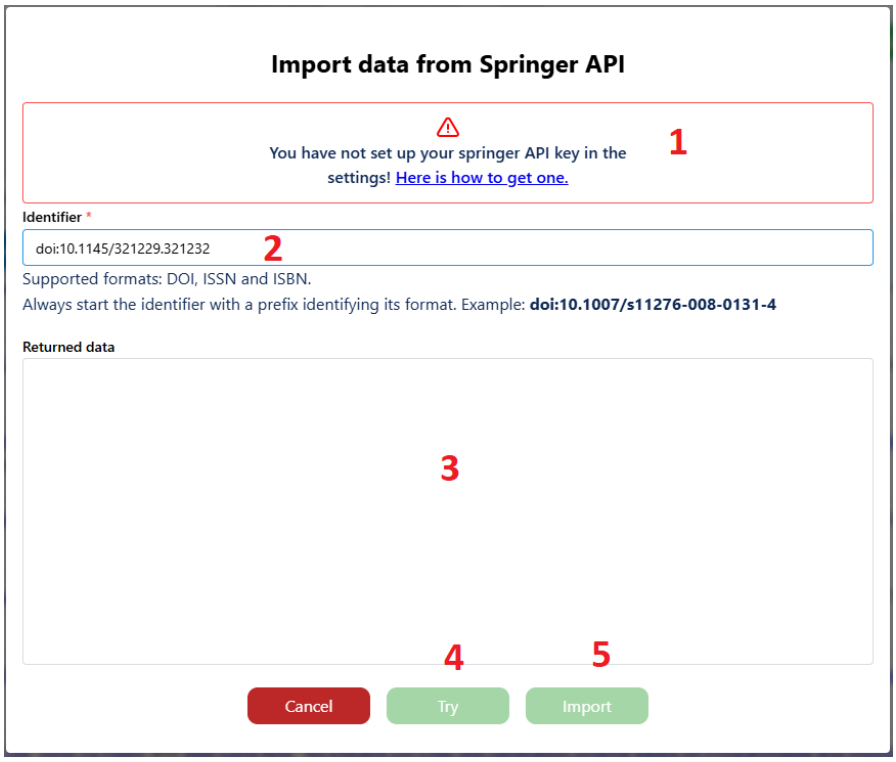


Figure 12 Import publication from the Springer API.

5. When the user fetched the correct data, the "Import" button navigates the

user to the filled-out form.

Detail of the publication

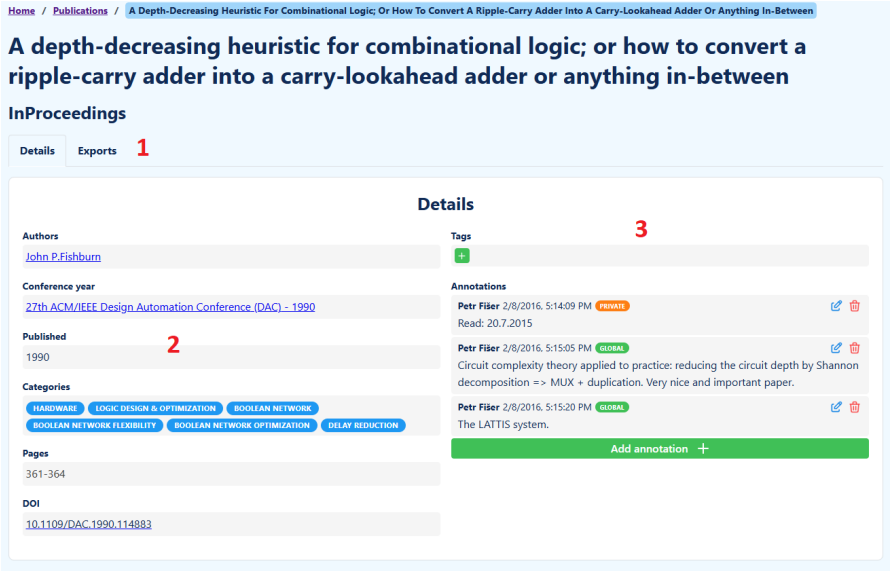


Figure 13 The detail of a publication, part 1.

When clicking on a record in the publication list, the user is navigated to a publication detail page. Figure 15 describes the first part of the page:

1. The detail tabs – detail or exports page.
2. The information about the publication.
3. Annotations or tags administration.

Figure 15 contains the second part of the details page:

1. The files of the publication.
2. Edit or delete buttons.
3. Metadata – created by and modified by.

Exporting a publication means creating a citation format from publication information for the purpose of citing it. Figure 15 describes the export tab:

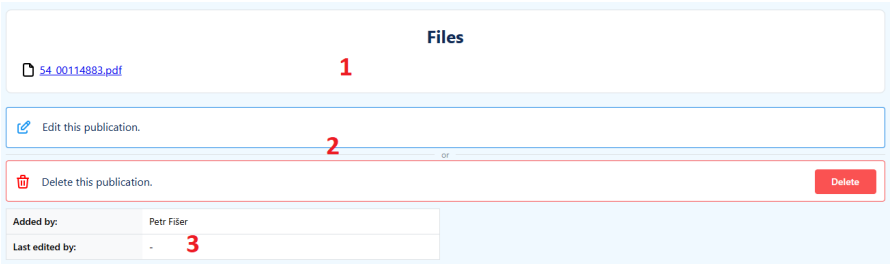


Figure 14 The detail of a publication, part 2.

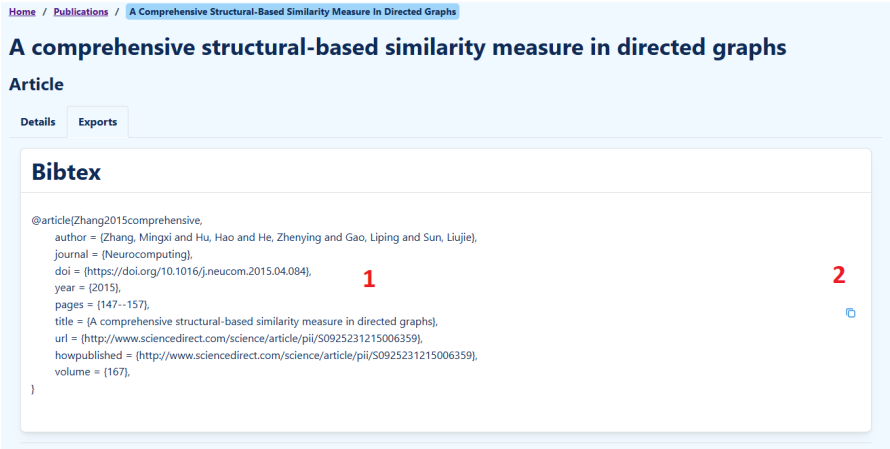


Figure 15 The publication exports page.

1. The final citation.
2. A button for copying the citation.

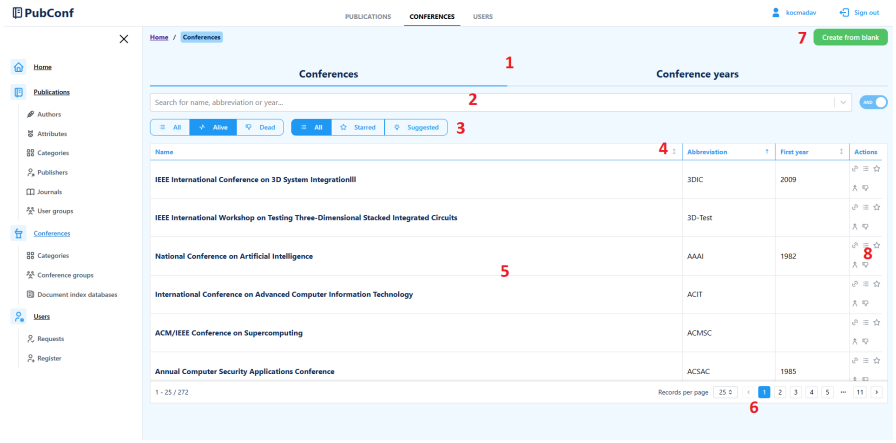


Figure 16 List of conferences and their filters.

Figure 16 contains the list of conferences and their filters to view, search and filter conferences:

1. The conference and conference years tab – clicking the tab changes the view.
2. Search bar - search by title, abbreviation or year.
3. Filter buttons – see dead or alive or all conferences or see favourites or suggested conferences.
4. Headers with order by switches.
5. The data.
6. Pagination options.
7. Button that navigates to the create conference form.
8. The action buttons – show associated publications or this conference's years or set this year as alive/dead or this conference as favourite or merge this conference with another one.

Figure 17 contains the list of conference years and their filters:

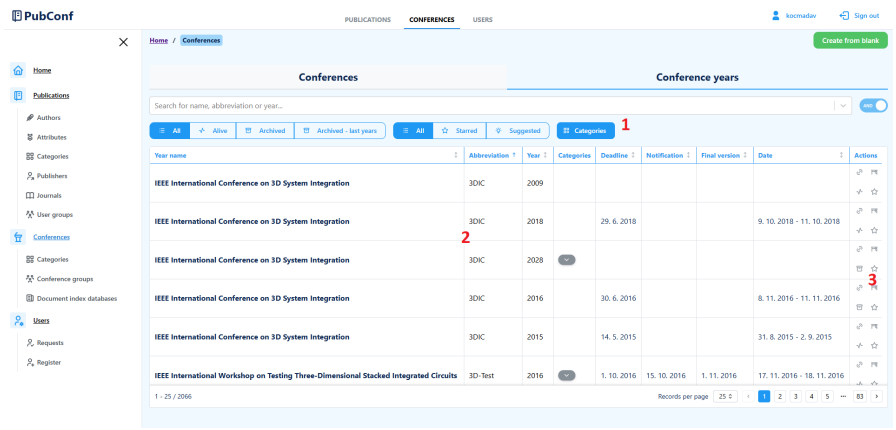


Figure 17 List of conference years and their filters.

1. Conference year filters – filter by archived, alive or last years of archived or see favourite (this filters favourite conferences, not years) or suggested years or filter by categories.
2. The data.
3. The action buttons – show associated publications or workshops or set this year as alive/archived or this conference as favourite.

Details

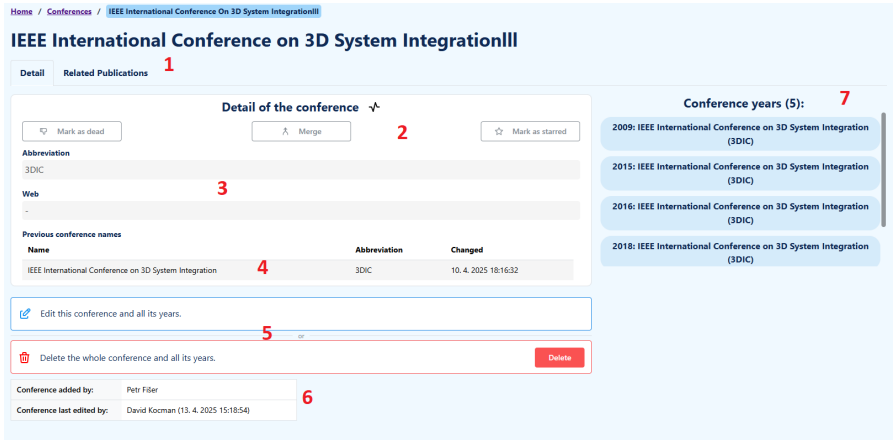


Figure 18 Detail of a conference.

After clicking on any of the record in the conference or conference year list, the user is navigated to a detail page. Figure 18 contains the detail page of a conference:

1. The tabs –deatil or export page.
2. Action buttons – set as alive/dead or merge with another category worse as favourite.
3. Information about the conference.
4. Previous names.
5. Edit or delete buttons.
6. Metadata – created by and modified by.
7. List of conference years of this conference.

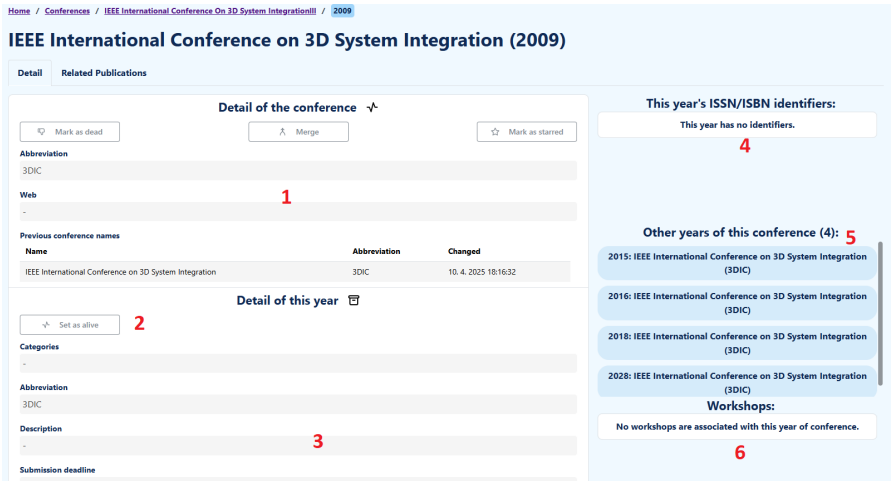


Figure 19 Detail of a conference year, part 1.

Figure 19 contains the detail page of a conference year, part one:

1. The conference detail
2. Action button – set as alive/archived.
3. Information about the year.
4. List of ISSN/ISBN identifiers of this year.
5. List of other conference years.
6. List of workshops.

Figure 19 contains the detail page of a conference year, part two:

1. Edit or delete buttons.
2. Metadata of the conference and the year of conference – created by and modified by.

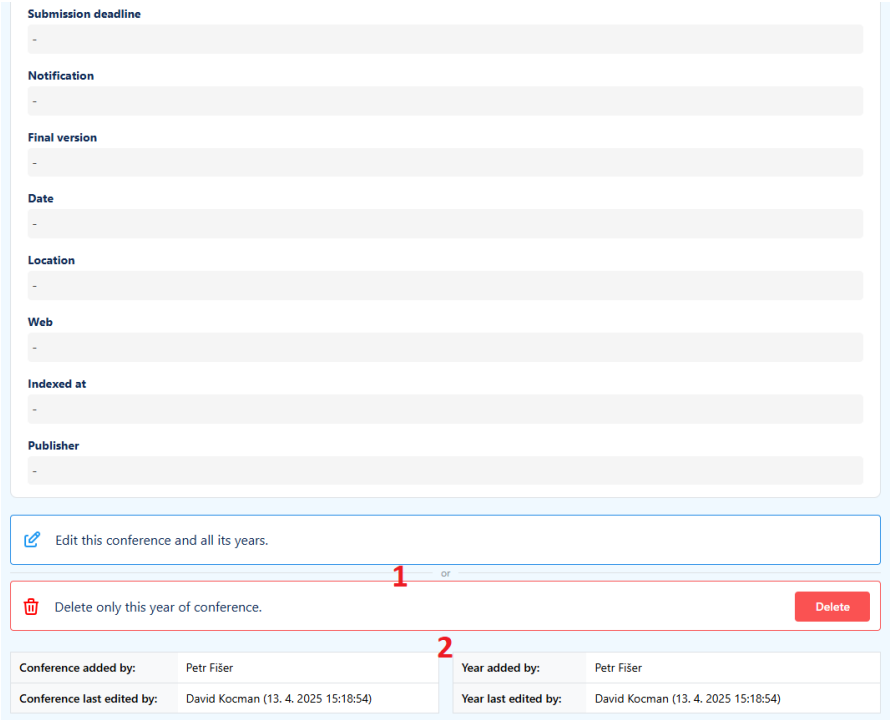


Figure 20 Detail of a conference year, part 2.

Creating a conference

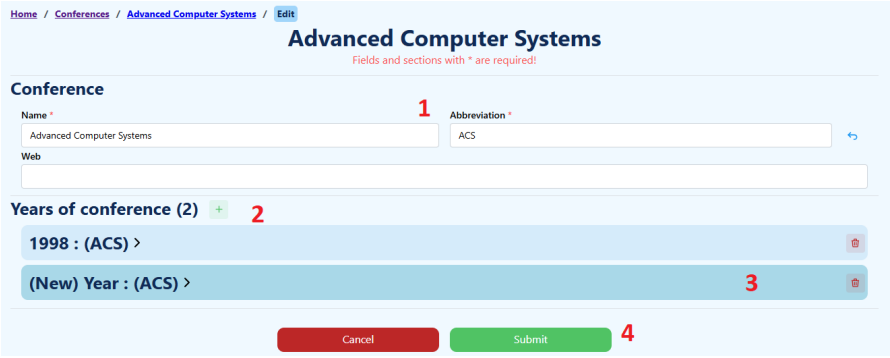


Figure 21 The form for creating or editing a conference.

Figure 21 contains the form for creating or editing conferences and all its years:

1. Conference information – next to the abbreviation field is the rollback to one of the previous names button. After clicking, a dropdown with all previous names is shown, and the user can change the current name and abbreviation to one of the previous ones.

2. Conference years list and create a new year button.
3. The year dropdown – for the sake of saving space the years are hidden in a dropdown. The trash can on the right removes the entry.
4. Submit button – validates the form and creates a new conference.

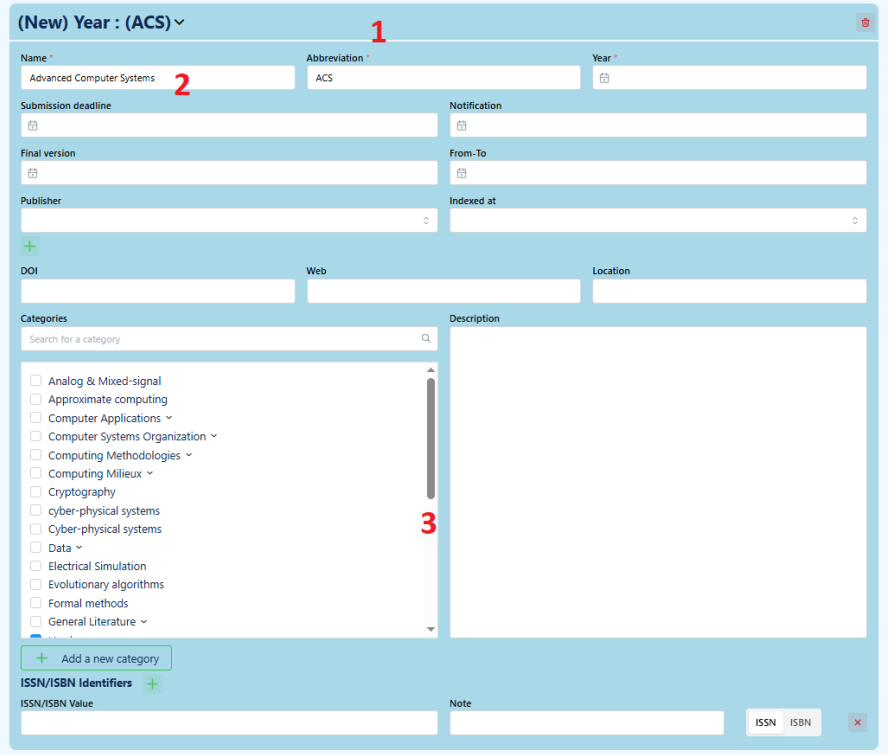


Figure 22 The contents of the year dropdown.

Figure 22 contains the contents of the year dropdown:

1. The dropdown header – the year and abbreviation is shown.
2. Mandatory fields – name, abbreviation, and year.
3. Same category select and administration interface as in Figure 6.

Navigation and Home Page

Home Page

The Home page contains lists of recently edited and visited publications and conferences:

1. Recently edited and visited publications
2. Recently edited and visited conferences

Navigation

The navigation lets the user switch between application pages.

The top navigation contains links for the main pages and user-specific actions:

1. **Home** – Home page
2. **Publications** – Publications list
3. **Conferences** – Conferences list
4. **Users** – Users list (accessible only to `admin` users)
5. **vondrt12** – Signed-in user's profile page
6. **Sign out** – Logs out the current user

Publication List

Create Publication

The top three buttons let the user create a new publication in three ways:

1. {·} **Import from definition** – Create from a BibTeX definition.
2. **Import from Springer** – Create using DOI, ISSN, or ISBN via the Springer API.
3. **Create from blank** – Create from scratch.

Filters

Users can use these options to filter the entries shown in the table below:

1. **Search input** – Enter multiple text values (author, title, category, group, year). Results must satisfy *all* specified values (logical AND).
2. **AND/OR** – Determine whether the search values are combined using logical AND or OR.
3. **Category** – Opens a popup to filter by one or more *Categories*.
4. **Groups** – Opens a popup to filter by one or more *Groups*.
5. **Publication Types** – Opens a popup to filter by one or more *Publication types*.
6. **Relation to user filters**:
   7. **All** – No filter applied.
   8. **Starred** – Show only publications *starred* by the user.
   9. **Annotated by** – Show only publications *annotated* by the user.
   10. **My** – Show only publications *created* by the user.

List Item Functions

Each result table row has four actions in the item's actions section:

1. – Copy the publication's citation.
2. – Star the publication (add to favorites).

3. – Show additional information (if available).
4. – Opens a popup to select the *Groups* for the publication.

Result Table Controls

On the bottom left, the user can control the *view* of the table. Options:

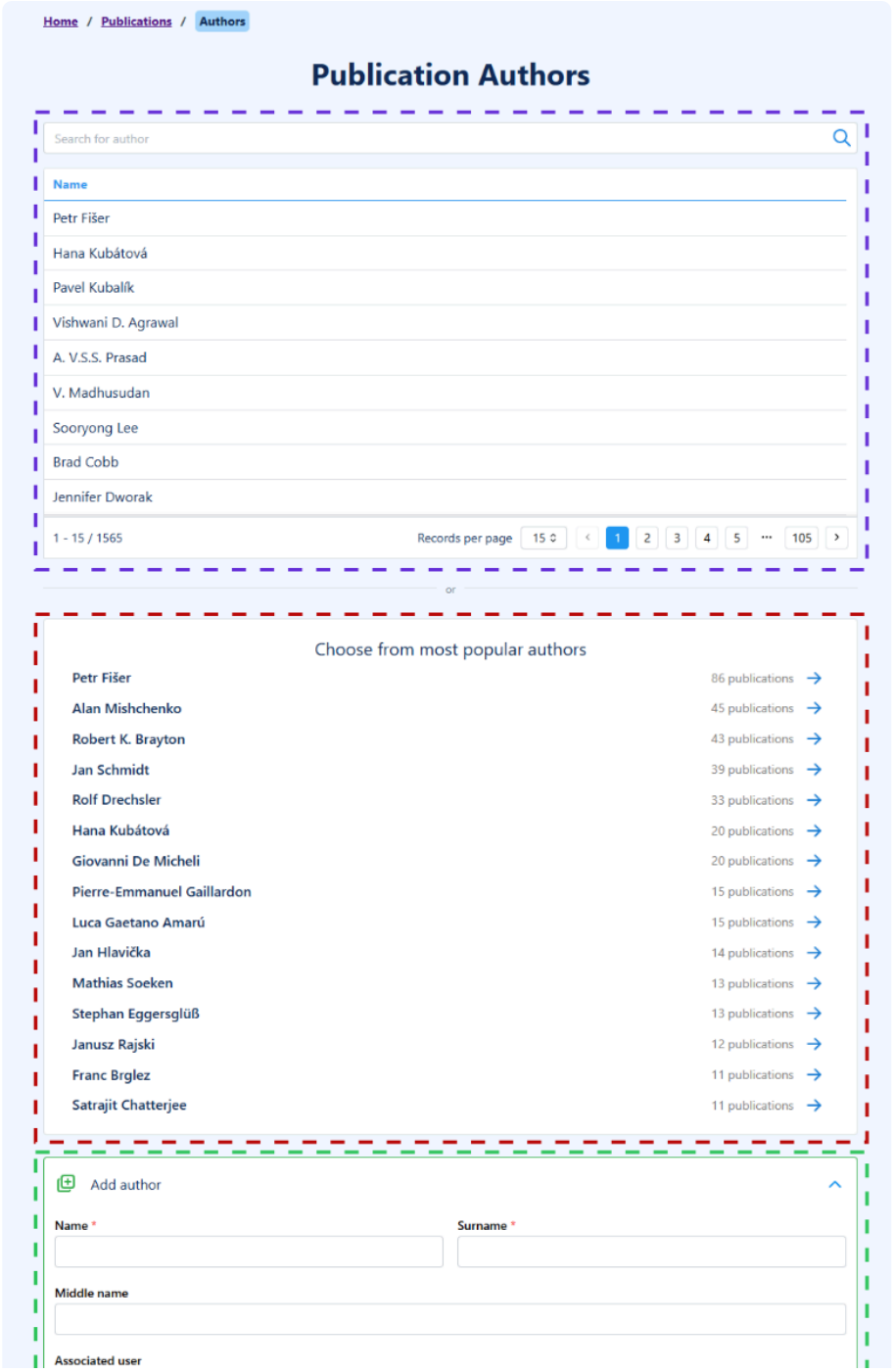
1. **Citation** – Citation view.
2. **Full row** – Full-row view with icons and dividers.
3. **Grid** – Grid view with individual cells.

The user can also **Expand** to show additional information for all results or **Collapse** to hide it.

On the bottom right, the user can control table *pagination*.

Entity Picker

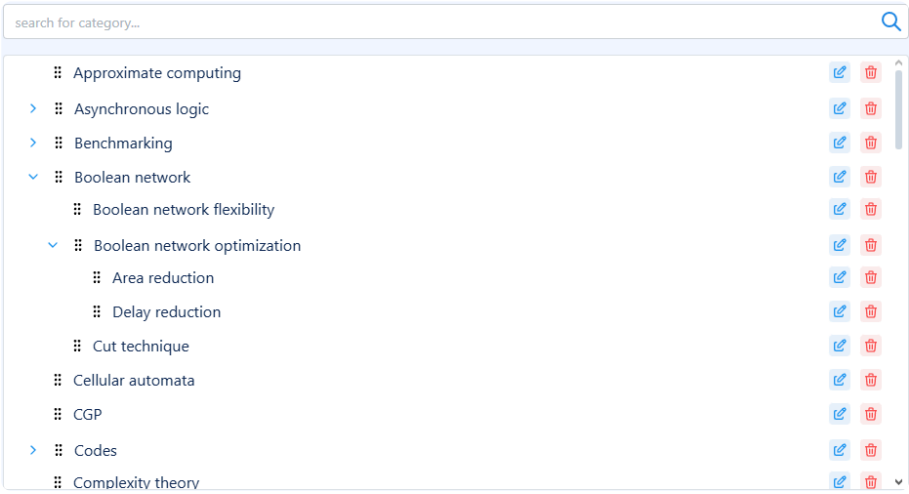
Entity administration always starts with a list view that lets the user search for an entity and navigate to its detail page.


Home / Publications / Authors
Publication Authors
Search for author
Name
Petr Fišer
Hana Kubátová
Pavel Kubalík
Vishwani D. Agrawal
A. V.S.S. Prasad
V. Madhusudan
Sooryong Lee
Brad Cobb
Jennifer Dworak
1 - 15 / 1565
Records per page 15
1 2 3 4 5 ... 105
or
Choose from most popular authors
Petr Fišer 86 publications
Alan Mishchenko 45 publications
Robert K. Brayton 43 publications
Jan Schmidt 39 publications
Rolf Drechsler 33 publications
Hana Kubátová 20 publications
Giovanni De Micheli 20 publications
Pierre-Emmanuel Gaillardon 15 publications
Luca Gaetano Amarú 15 publications
Jan Hlavička 14 publications
Mathias Soeken 13 publications
Stephan Eggersglüß 13 publications
Janusz Rajski 12 publications
Franc Brglez 11 publications
Satrajit Chatterjee 11 publications
Add author
Name *
Surname *
Middle name
Associated user

The list view contains:

1. Entities picker – Search by name; click a row to view details.
2. Entities popular list – Shows entities with the most publication connections.
3. Add entity form – Form to add a new entity.

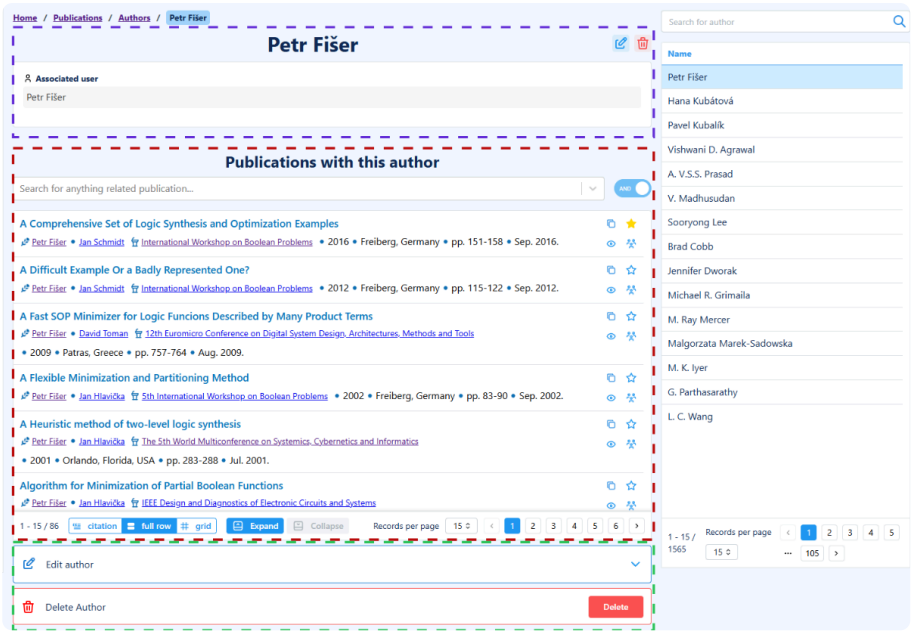
Category Picker



The category picker offers extra functionality:

1. ⠿ **Reordering** – Drag & drop to reorder categories; moving a parent moves its children.
2. **Inline editing** – Edit a category name directly in the picker.
3. **Inline deleting** – Delete a category (and its children) directly in the picker.

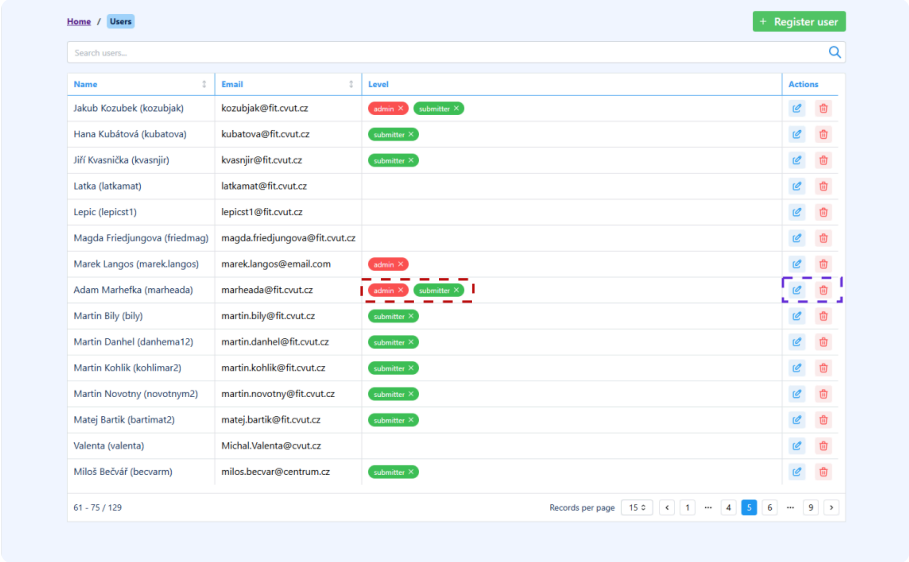
Entity Detail Page



The side picker provides the same actions as on the entity picker page. Additionally:

1. Detail information – Displays additional data about the entity.
2. Publication list – Shows publications linked to the entity.
3. Entity actions – Edit or delete the entity; for categories, add a subcategory.

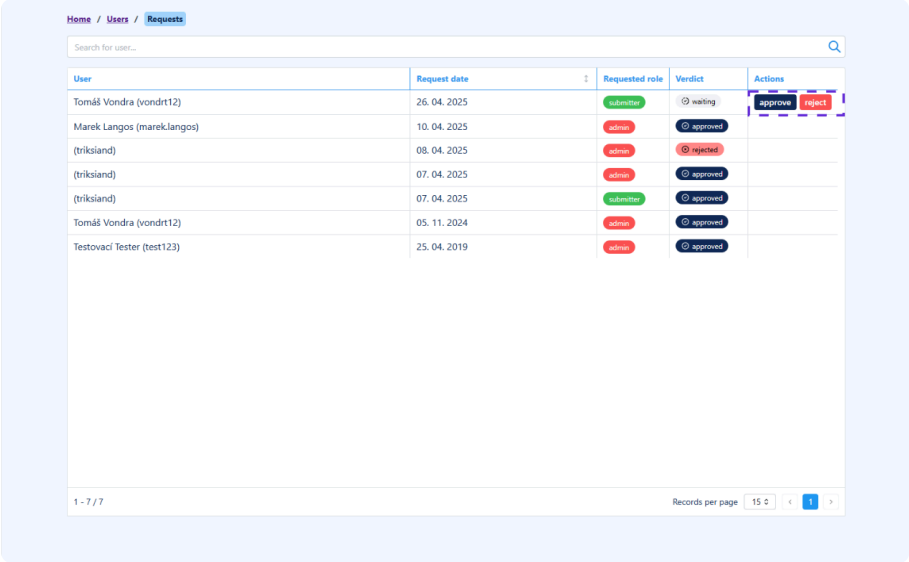
Users List



On this page, an administrator can search registered users. Each row includes actions to remove roles, edit, or delete the account.

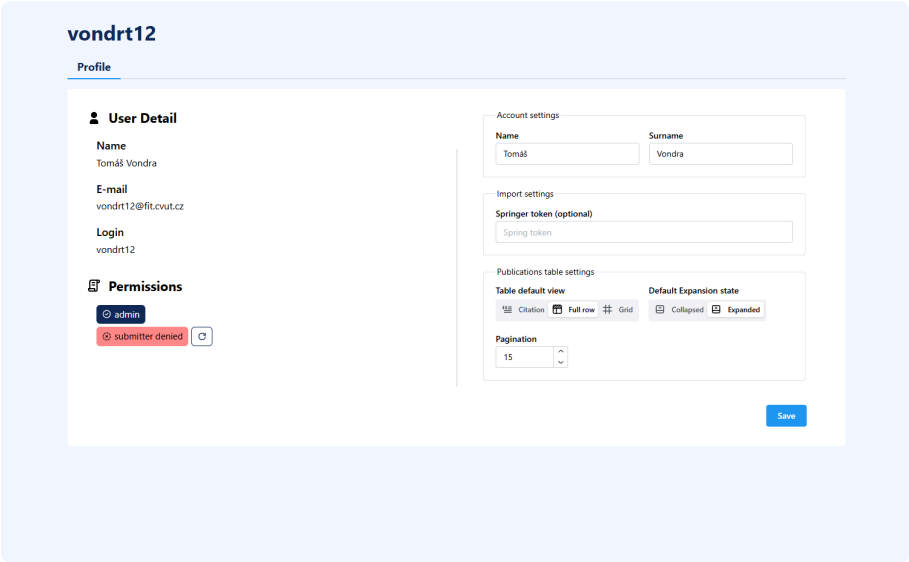
The + Register user button navigates to the registration page for adding external users. External users sign in with email and password and require administrator registration. FIT ČVUT students and employees sign in via Microsoft Entra ID.

User Requests



Administrators can approve or reject role requests. The table holds up to *100* requests; when full, the oldest handled request is deleted.

User Profile



Permissions

In this section, users can request the *admin* and *submitter* roles:

- *Admin* – manage other users' permissions and requests.
- *Submitter* – edit publications, conferences, and related entities.
  If a request is denied, the user can reapply with the **Reapply** button.

User Preferences

- **Account settings** – Edit first and last name.
- **Import settings** – Enter a personal Springer token for importing publications via the Import from Springer option.
- **Publication table settings**:
  1. **Table default view** – Choose the default table view.
  2. **Default expansion state** – Set whether additional information is visible by default.
  3. **Pagination** – Specify how many items each table displays at once.

Bibliografie

1. KOCMAN, David. *Reimplementace databáze konferencí a publikací II*. 2025. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
2. HAJTOL, Peter. *Databáze konferencí a publikací IV*. 2022. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
3. ŠTIPL, Stanislav. *Databáze konferencí a publikací II*. 2018. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
4. ICT ODDĚLENÍ FIT ČVUT. *Seznam podporovaného externího softwaru (interní dokumentace)* [`https://help.fit.cvut.cz/employees/external-sw/index.html`]. 2025.
5. TEAM, React. *ReactDOM.hydrateRoot* [online]. 2024. [cit. 2025-04-07]. Dostupné z: `https://react.dev/reference/react-dom/client/hydrateRoot`.
6. NUXT TEAM. *Rendering - Nuxt 3 Documentation* [online]. 2024. [cit. 2025-04-07]. Dostupné z: `https://nuxt.com/docs/guide/concepts/rendering`.
7. SQLITE DEVELOPMENT TEAM. *Appropriate Uses For SQLite* [online]. 2025. [cit. 2025-04-08]. Dostupné z: `https://www.sqlite.org/whentouse.html`.
8. SQLITE DEVELOPMENT TEAM. *SQLite and Multiple Connections* [online]. 2025. [cit. 2025-04-08]. Dostupné z: `https://www.sqlite.org/faq.html#q5`.
9. POSTGRESQL GLOBAL DEVELOPMENT GROUP. *Chapter 13: Concurrency Control* [online]. 2025. [cit. 2025-04-08]. Dostupné z: `https://www.postgresql.org/docs/current/mvcc.html`.

10. POSTGRESQL GLOBAL DEVELOPMENT GROUP. *Chapter 3.4: Transactions* [online]. 2025. [cit. 2025-04-08]. Dostupné z: https://www.postgresql.org/docs/current/tutorial-transactions.html.
11. VERCEL. *How to Build a Fullstack App with Next.js, Prisma, and Postgres* [online]. 2025. [cit. 2025-04-08]. Dostupné z: https://vercel.com/guides/nextjs-prisma-postgres.
12. MICROSOFT CORPORATION. *What is Microsoft Entra ID?* [online]. 2025. [cit. 2025-04-08]. Dostupné z: https://learn.microsoft.com/en-us/entra/fundamentals/whatis.
13. MICROSOFT CORPORATION. *What is DevOps?* [online]. 2025. [cit. 2025-04-08]. Dostupné z: https://learn.microsoft.com/en-us/devops/what-is-devops.
14. VERCEL INC. *Next.js Configuration Reference* [online]. 2024. [cit. 2025-04-13]. Dostupné z: https://nextjs.org/docs/app/building-your-application/configuring/.
15. VERCEL. *Next.js Documentation – Routing Fundamentals* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://nextjs.org/docs/app/building-your-application/routing.
16. VERCEL. *Next.js Documentation – Client and Server Components* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://nextjs.org/docs/getting-started/react-essentials.
17. VERCEL. *Next.js Documentation – Server Actions* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions.
18. VERCEL. *Next.js Documentation – Middleware* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://nextjs.org/docs/app/building-your-application/routing/middleware.
19. PRISMA. *Next.js and Prisma Guide* [online]. 2024. [cit. 2025-04-13]. Dostupné z: https://www.prisma.io/docs/guides/nextjs#25-set-up-prisma-client.
20. PRISMA. *Prisma Documentation* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://www.prisma.io/docs.
21. SASS LANGUAGE TEAM. *Sass: Syntactically Awesome Stylesheets* [online]. 2024. [cit. 2025-04-13]. Dostupné z: https://sass-lang.com.
22. MICROSOFT. *OIDC protocol convergence scenarios - web app* [online]. 2024. [cit. 2025-04-12]. Dostupné z: https://learn.microsoft.com/en-us/entra/identity-platform/media/v2-protocols-oidc/convergence-scenarios-webapp.svg.
23. JED WATSON ET AL. *React Select* [online]. 2024. [cit. 2025-04-13]. Dostupné z: https://react-select.com/home.

24. FLORESCU, Ion Cristian. *Mantine DataTable Documentation* [online]. 2024. [cit. 2025-04-13]. Dostupné z: https://icflorescu.github.io/mantine-datatable/.

25. GITLAB. *Compute the cache key from the lock file* [online]. 2024. [cit. 2025-05-01]. Dostupné z: https://docs.gitlab.com/ci/caching/#compute-the-cache-key-from-the-lock-file.