NNPIA LAB 03: REST API

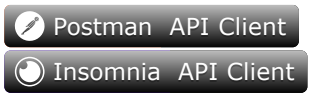
Cílem tohoto cvičení je rozšířit stávající aplikaci o plnohodnotné REST API nad entitou `AppUser`.
Studenti si prakticky vyzkouší práci s HTTP metodami `POST`, `GET`, `PUT`, `DELETE`, pracovat s parametry a ošetřovat
chybové stavy.

Předpoklady

- JDK 21 nebo novější
  - Na školních počítačích je nainstalována JDK 17. Můžete ale použít funkci IntelliJ IDEA pro stažení a nastavení JDK
    21 přímo v IDE (File → Project Structure → SDKs → + → JDK).
- IntelliJ IDEA
  - Doporučena je verze **IntelliJ IDEA Ultimate**
    - Na školních počítačích je nainstalována verze 2023.3.3.
    - Na osobních počítačích doporučujeme využít nejnovější verzi 2025.2.3 nebo novější.
  - Studenti mohou využít **bezplatnou studentskou licenci**
- Verzovací systém Git
- **Webový prohlížeč** ideálně postavený na jádru Chromium
  - Google Chrome, Microsoft Edge, Brave…
- **HTTP klient pro testování API**
  - Postman
  - Insomnia

3.1. Seznámení s REST API klientem

Než začneme rozšiřovat backend, musíme umět manuálně testovat jednotlivé HTTP metody. Webový prohlížeč umožňuje snadno
testovat pouze `GET` požadavky.
Pro práci s dalšími HTTP metodami je lepší použít REST API klienta, který umožňuje ukládat a organizovat požadavky,
nastavovat hlavičky, tělo požadavku a lépe analyzovat odpovědi.



1. Soubor `LAB03.md` vložte do složky `doc` a zahrňte do verzování.
2. Spusťte backend a ujistěte se, že běží na `http://localhost:8080`.

💡 *Pokud Vaše aplikace běží na jiném portu než 8080. Používejte v následujících úkolech Váš port.*

3. Vyberte si jednoho REST API klienta `Postman` nebo `Insomnia` a vytvořte nový požadavek:
    - Otestujte endpoint z minulého cvičení.
        - Metoda: `GET`
        - URL: `http://localhost:8080/api/v1/app-users`

💡 *Oba dva klienti umožňují při prvním spuštění vytvoření účtu. To není nutné, testovat API lze i v `light` módu bez*
přihlášení. Účet ale přináší výhodu v možnosti synchronizovat kolekce HTTP requestů s cloudem a tím i jejich zálohování.

💡 *Ukázka použití nástroje Postman pro testování HTTP požadavků.*

💡 *Ukázka použití nástroje Insomnia pro testování HTTP požadavků.*

3.2. Implementace POST endpointu

Aplikace aktuálně umožňuje pouze čtení dat. Nyní přidáme i možnost vložení.
Metoda `POST` slouží v REST API pro vytvoření nového záznamu.

1. Ve třídě `AppUserService` vytvořte metodu `createAppUser(AppUser appUser)`:
    - `id` nenastavujte z parametru, ale inkrementujte jej uvnitř metody.
    - Přidejte uživatele do kolekce.
    - Vraťte vytvořeného uživatele.
2. Do `AppUserController` přidejte nový endpoint:
    - HTTP metoda: `POST`.
        - URL: `/api/v1/app-users`.
        - Parametr: `@RequestBody AppUser appUser`.
        - Návratový typ: `ResponseEntity<AppUser>`.

💡 *Všimněte si že narozdíl od minulých cvičení má metoda parametr. K vytvoření nového záznamu už potřebujeme znát*
uživatelská data.

3. V novém endpointu zavolejte metodu `createAppUser`:
    - HTTP status kód `201 CREATED`.
    - Jako tělo odpovědi vraťte nově vytvořeného uživatele.

❗ *Bez anotace `@RequestBody` nebude Spring mapovat JSON na objekt.*

💡 *V rámci RESTu by POST metoda měla vracet nově vytvořený záznam.*

4. Otestujte nově vytvořený endpoint pomocí REST API klienta:
    - Restartujte Spring Boot aplikaci.

- Pošlete `POST` požadavek na `http://localhost:8080/api/v1/app-users` s JSON tělem obsahujícím všechny atributy
  třídy `AppUser`.
- Ověřte, že odpověď obsahuje nově vytvořeného uživatele s inkrementovaným `id` a že status code je `201 CREATED`.
- Otestujte i `GET` endpoint a ověřte, že nový uživatel je nyní součástí seznamu.

💡 Použijte volně dostupné AI k vytvoření vzorových dat.

3.3. Implementace DELETE endpointu s parametrem

Pokud budeme chtít odstranit existující záznam, použijeme HTTP metodu `DELETE` podle konvencí REST API. Nicméně
potřebujeme znát i identifikátor `id`, abychom věděli, který záznam smazat. V REST API se pro tento účel používají
path variables.

Spring Boot Cvičení

1. V `AppUserService` implementujte metodu `deleteAppUser(Long id)`:
   - Pokud uživatel existuje, odeberte jej z kolekce.
2. V controlleru vytvořte metodu:
   - HTTP metoda: `DELETE`
   - URL: `/api/v1/app-users/{id}`
   - Parametr: `@PathVariable Long id`
   - HTTP status kód `204 NO CONTENT` při úspěšném smazání.

💡 *DELETE obvykle nevrací tělo odpovědi*

❗ *Spring Boot zpracovává HTTP požadavky vícevláknově. Pro každý přicházející požadavek vyčlení Tomcat server jedno
volné vlákno. Protože může běžet více požadavků paralelně, vzniká riziko, že dvě a více
vláken přistoupí ke sdílené kolekci ve stejný okamžik. To může vést k nekonzistenci dat nebo
vyvolat `ConcurrentModificationException`. Tento problém lze v paměti řešit pomocí
thread-safe kolekcí, jako je například `CopyOnWriteArrayList`. Nicméně v dalším cvičení nahradíme
tuto kolekci databází, která souběžný přístup k datům bezpečně řídí pomocí transakcí a zámků. Pro účely tohoto cvičení
tedy můžete thread-safety kolekce ignorovat.*

💡 *V logách máte i informaci, které vlákno log vypsalo. Identifikátor vláka je umístěn v hranatých závorkách `[nio-8080-exec-3]` a `[nio-8080-exec-4]`*

```
2026-02-21T18:48:33.238+01:00 DEBUG 38835 --- [backend] [nio-8080-exec-3]
o.s.web.servlet.DispatcherServlet        : Completed 204 NO_CONTENT
2026-02-21T18:48:34.862+01:00 DEBUG 38835 --- [backend] [nio-8080-exec-4]
o.s.web.servlet.DispatcherServlet        : DELETE "/api/v1/app-users/1", parameters={}
```

3.4. Error handling pomocí `@ResponseStatus`

Správně navržené REST API musí vracet smysluplné HTTP status kódy při chybě.
V předchozím DELETE endpointu vyhledáváme uživatele podle `id`. Pokud uživatel neexistuje, měli bychom vrátit
`404 NOT FOUND`.

V tomto úkolu využijeme anotaci `@ResponseStatus`, která umožňuje svázat konkrétní výjimku s konkrétním HTTP status
kódem.

Spring Boot Cvičení

1. Vytvořte vlastní výjimku `AppUserNotFoundException` v balíčku `exception`.
    - Třída bude dědit z `RuntimeException`.
    - Označte ji anotací `@ResponseStatus(HttpStatus.NOT_FOUND)`.
    - Přidejte konstruktor, který přijímá `Long id` a vytvoří chybovou zprávu ve formátu: `"User with id {id} not found"`.
2. V metodě `deleteAppUser(Long id)` vyhoďte tuto výjimku, pokud uživatel s daným ID neexistuje.
3. Otestujte chování:
    - Pošlete `DELETE` požadavek na neexistující ID.
    - Ověřte, že aplikace vrací:
        - Status code `404 NOT FOUND`
        - Chybovou zprávu v odpovědi.

💡 *Použití `@ResponseStatus` je jednodušší varianta řešení. Pro komplexnější aplikace se často používá globální handler*
pomocí `@RestControllerAdvice`, který umožňuje centralizovat správu chyb.

3.5. DTOs

V aktuální implementaci používáme třídu `AppUser` přímo jako `@RequestBody`.
To však není ideální řešení. Klient by neměl určovat hodnotu `id`, protože ta je generována aplikací.
Zároveň nechceme tuto hodnotu odebrat z třídy `AppUser`, protože ji budeme potřebovat při práci s databází v dalších
cvičeních.

V tomto úkolu oddělíme vstupní model (request) od doménového modelu pomocí DTO (Data Transfer Object) a převod mezi nimi
provedeme v **service třídě**.

Spring Boot Cvičení

1. Vytvořte nový balíček `dto` a v něm třídu `AppUserRequestDto`.
    - Třída bude obsahovat pouze:

        - `String email`
        - `String password`
        - `Boolean active`
    - Atribut `id` zde nebude.
    - Využijte Lombok pro generování konstruktoru a getterů/setterů.
2. Upravte `POST` endpoint:
    - Místo `AppUser` použijte jako `@RequestBody` typ `AppUserRequestDto`.
3. V třídě `AppUserService` implementujte metodu pro převod z `AppUserRequestDto` na `AppUser`:
    - **Metodu implementujte přetížením.**
    - Hodnoty `email`, `password` a `active` převeďte z DTO.
    - `id` nastavujte výhradně na serveru (např. inkrementací).
    - Přidejte nového uživatele do kolekce a vraťte jej.
4. Restartujte aplikaci a ověřte, že:
    - `POST` endpoint funguje stejně jako před úpravou.
    - Ověřte, že `id` je generováno serverem a není možné jej nastavit z klienta.

💡 *Oddělení DTO od doménového modelu je běžná praxe v profesionálním vývoji.*

💡 *DTO se používá i pro návratové hodnoty. V příštím cvičení si ukážeme jaké to může mít výhody při práci s databází.*

3.6. Implementace PUT endpointu a GET s parametrem

Tvorba dalších endpointů už je repetitivní. Pro aktualizaci existujícího záznamu se používá HTTP metoda `PUT` nebo
`PATCH`.
Také bychom chtěli mít možnost získat konkrétní záznam na základě jeho ID. Pro tento účel můžeme vytvořit další `GET`
endpoint, který bude mít jako parametr ID uživatele.

Vibe coding Dobrovolný úkol

1. Pomocí funkce `Agent` sepište prompt který implementuje `PUT` endpoint pro aktualizaci uživatele:
    - V promptu můžete používat symbol `#` pro specifikování souborů se kterými má agent pracovat.
    - Připomeňte agentovi aby se zaměřil i na správné návratové HTTP status kódy a ošetření chybových stavů.
2. Stejným způsobem vytvořte `GET` endpoint pro získání uživatele podle ID.

💡 *GET endpoint už máme ve třídě implementovaný. Nicméně můžeme implementovat další pokud dodržíme unikátní URL. V*
tomhle případě přibude dodatečný `path` parametr. Použijte funkci `Ask` pro vysvětlení této problematiky.

3.7. Validace vstupů pomocí Bean Validation

Uživatelé mohou posílat nevalidní data. Například prázdný email nebo heslo. Pro zajištění integrity dat je důležité validovat vstupy.

Vibe coding Dobrovolný úkol

1. Pomocí funkce `Ask` se zeptejte AI asistenta na možnosti validace vstupů v Spring Bootu. Zjistěte jaké závislosti
   jsou pro to potřeba.
   - Následně s ním konzultujte scénaře chybných dat která by uživatel mohl poslat.
   - Pomocí `#` specifikujte v promptu třídu `AppUser` aby AI mělo kontext o atributech.
2. Následně využijte funkci `Agent` pro implementaci validace vstupů pro `POST` a `PUT` endpointy.
   - Připomeňte agentovi aby se zaměřil i na chybové HTTP status kódy a ošetření chybových stavů.
   - Instruujte agenta aby Vám všechny nové anotace vysvětlil.

Odevzdání

- Po dokončení všech úkolů vytvořte **commit** se všemi provedenými změnami a **pushněte jej do vzdáleného repozitáře**.
- Název commitu musí **začínat označením LAB03** a obsahovat **stručný popis změn**.

```
LAB03 - REST API

Implementace POST, PUT, DELETE a error handling.
```

Užitečné odkazy a zdroje

- Exception Handling in Spring MVC
- REST API Best Practices
- What is a Data Transfer Object (DTO)?
- Validation in Spring Boot
- Builder