Material UI

Material UI,
viacstránkové aplikácie,
fetch, async

UINF/PAZ1c
epizóda 11

Material UI

- Čo je  ?
  - https://mui.com/
  - Vychádza z Android UI (Material Design)
  - Kolekcia už oštýlovaných komponentov pre React
  - Nepotrebujete CSS
    - Úprava štýlov sa rieši čisto v JS/TS
  - Vstavaná podpora pre svetlý/tmavý režim

- Nainštalujeme ho do projektu
  - npm install @mui/material @emotion/react @emotion/styled @fontsource/roboto @mui/icons-material

Vyčistenie CSS

- Material UI má vlastné komponenty
- Bijú sa s CSS, ktoré máme už vygenerované
  - Máme tam dark mode, ktorý si MUI rieši ináč
  - O týždeň si ukážeme ako "témovať" MUI
- Zatiaľ si vypneme dark mode v CSS
  - Choďte do index.css
    - V **root** zmažte "color-scheme" a "background-color" a nastavte "text-align" na "end"
    - V **body** zmažte "display", "min=width", "min-height"
  - Choďte do App.css
    - Vymažte prvý riadok s "max-width"

Zobrazíme si užívateľov v tabuľke

```
<TableContainer component={Paper}>
 <Table>
  <TableHead>
    <TableRow>
     <TableCell>Name</TableCell>
      ...
    </TableRow>
  </TableHead>
  <TableBody>
    {users.map((user) => (
     <TableRow onClick={() => handleRowClick(user)}>
      <TableCell>{user.id}</TableCell>
        ...
     </TableRow>
    ))}
  </TableBody>
 </Table>
</TableContainer>
```

Podmienečné zobrazenie prístupov

```
<TableContainer component={Paper}>
 <Table>
  ...
 <TableBody>
    {users.map((user) => (
      <>
       <TableRow onClick={() => handleRowClick(user)}>
            ...
       </TableRow>
       <TableRow>
         <TableCell style={{paddingBottom: 0, paddingTop: 0, borderBottom: 'none'}} colSpan={5}>
           <Collapse in={selectedUser?.id === user.id} unmountOnExit>
              ...
           </Collapse>
       </TableRow>
      </>
    ))}
  </TableBody>
 </Table>
</TableContainer>
```

Atribúty používateľa

Počet TableCell pri používateľovi

Zoznam prístupov, ktorý bude viditeľný iba pri kliknutí na používateľa

Stiahnutie dát z REST servera

- Vstavaná funkcia *fetch* na komunikáciu s HTTP (a teda aj REST API)

```
fetch(`${restURL}/users`)
   .then(response => response.json())
   .then(data => setUsers(data as User[]))
   .catch(error => setError(new Error("...", {cause: error})));
```

- *fetch* treba zabaliť do "efektu"
  - To platí pre všetky IO operácie
  - Ináč sa fetch bude volať neustále (60+ krát za sekundu)
  - useEffect sa volá napr. iba pri vytváraní komponentu alebo refreshi stránky
  - ak do poľa závislostí dáme nejakú premennú z useState, tak sa useEffect zavolá aj pri zmene stavu premennej

```
useEffect(() => {
   fetch(`${restURL}/users`)
      . . .
}, [])
```

Cross-Origin Resource Sharing (CORS)

- Prehliadač umožňuje robiť sieťové volania iba na tú istú adresu/port čo máme v "URL políčku"
  - V našom prípade localhost:5173
- Ale REST server beží na localhost:8080
  - Teda fetch nebude fungovať
- Musíme povoliť tzv. CORS
  - Nastavíme REST server a "white-listujeme" adresu, kde beží webové UI (localhost:5173)
  - Browser si zistí, že REST server má web appku vo white-liste
  - A hurá, všetko ide

Nastavenie CORS

- V konfiguračnej triede REST servera:
  - Napr. vytvorme si triedu s @Configuration

```
@Value("${cors.allowedOrigins}")
private String allowedOrigins;

@Bean
public WebMvcConfigurer corsConfigurer() {
 return new WebMvcConfigurer() {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
   registry
    .addMapping("/**")
    .allowedOrigins(allowedOrigins.split(","))
    .allowedMethods("*");
  }
 };
}
```

- V **resources/application.properties**:

```
cors.allowedOrigins=${CORS_ALLOWED_ORIGINS:http://localhost:5173,http://localhost:8080}
```

Blokujúce volania (zvyčajne IO volania)

- Práca so súbormi
  - Disky sú 1000 – 100 tisíc krát pomalšie ako CPU
- Práca so sieťou (DB, HTTP)
  - asi 100 tisíc - milión krát pomalšie CPU
  - V praxi často viac krát (10 miliárd krát)
- **blokujeme CPU**
  - OS nemá rád blokovanie a prepne kontext
    - vezme nám CPU a dá inej appke
  - *fetch* robí sieťové volanie
- V GUI **neslobodno** blokovať CPU
  - Aplikácia sa môže "sekať"
  - Počas blokujúceho volania užívateľ nemôže nič robiť

(Ne)blokujúce volania v JS/TS

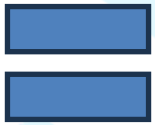
- Konkurentné programovanie pomocou **korutín**
  - Kooperatívny multitasking
- IO funkcie vracajú Promise objekty (sľuby)
  - Obsahuje úlohu, ktorú OS/runtime vykonáva na pozadí
    - IO multiplexing v OS (non-blocking IO – **NIO**)
  - Runtime v nekonečnom cykle kontroluje stav Promise-ov
    - Tzv. event loop
  - Keď je sľub splnený (fulfilled), tak sa spustí then callback, ktorý môže vracať nový Promise
  - Aplikácia neblokuje CPU => OS sa na nás nehnevá
  - Výnimky (rejected promise) môžeme odchytávať cez catch callback

```
fetch(`${restURL}/users`)
    .then(response => response.json())
    .then(data => setUsers(data as User[]))
    .catch(error => setError(new Error("...", {cause: error})));
```

(Ne)blokujúce volania v JS/TS

Alternatívne môžeme pracovať s **async-await**, čo je *procedurálny* syntax-cukor nad *funkcionálnymi* Promise-ami

```
fetch(`${restURL}/users`)
    .then(response => response.json())
    .then(data => setUsers(data as User[]))
    .catch(error => setError(new Error("Could not fetch
          users", {cause: error})));
```

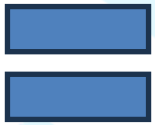


```
try {
    const response = await fetch(`http://localhost:8080/users`)
    const data = await response.json()
    setUsers(data as User[])
} catch (error) {
    setError(new Error("Could not save user", {cause: error}))
}
```

(Ne)blokujúce volania v JS/TS

Alternatívne môžeme pracovať s **async-await**, čo je imperatívny syntax-cukor nad Promise-ami

```
fetch(`${restURL}/users`)
    .then(response => response.json())
    .then(data => setUsers(data as User[]))
    .catch(error => setError(new Error("Could not fetch
          users", {cause: error})));
```



```
try {
    co
    co
    se
} cat
    se
}
```

Blokujúce volanie cez **await** možeme používať iba v tzv. **async** funkciách, čo **useEffect** nie je. Takže tam async-await nevieme (jednoducho) použiť.

Mimochodom: "Neblokujúce" volania v Jave (pre fajnšmekrov)

- V Jave štandardne blokujeme CPU
- Môžeme však IO volať v druhom vlákne

```
new Thread(() -> loadCSVtoDB()).start();
```

  - budete mať na predmete KOPR
  - druhé vlákno zvyčajne beží na inom CPU jadre
  - preemtívny multitasking cez OS
  - vlákna na pozadí vytvára OS
    - Ak druhé vlákno blokuje CPU, tak hlavné vlákno stále beží

Mimochodom: "Neblokujúce" volania v Jave (pre fajnšmekrov)

Limitácia vlákien

- Vlákna sú super pre výpočtovo náročné appky (CPU bound) aj desktop appky.
- Webové služby však s vláknami veľmi neškálujú (sú IO bound)
  - Typická Java web appka/služba -> 1 online user = 1 vlákno
  - bežný PC zvládne naraz iba cca 10k vlákien
    - ...oproti miliónom korutín v JS/TS
    - Limit je RAM
      - 1 vlákno = cca 1 MB RAM, 10k vlákien = cca 10 GB RAM
  - Koľko ľudí *naraz* používa Instagram/Netflix/Bolt...?
    - Asi o dosť viac ako 10k, že áno...
  - Napr. Netflix má veľa Javy, ale nepoužívajú veci "typické" pre bežný korporát

Efektívne blokujúce volania mimo JS/TS (pre fajnšmekrov)

- C#, Kotlin, Rust: vlákna + korutiny
  - Samostatný event loop pre každé CPU jadro
  - "Dispatcher" korutiny rovnomerne rozdeľuje
  - Aj Java cez externé knižnice/frejmworky (RxJava, Reactor, Vert.x, Quarkus, Spring WebFlux…)
    - Neohrabané, u nás veľmi málo rozšírené, čeknite https://netflixtechblog.com/
- Go, Java 19+: *zásobníkové* korutiny (gorutiny/virtuálne vlákna)
  - hybrid medzi vláknami a korutinami
  - "preemtívne kooperatívny" multitasking
  - Interne tam beží (aj) event loop s IO multiplexing
  - Programátor s tým pracuje ako s vláknami
    - Žiadne async-await, ani Promise
    - Takmer drop-in náhrada za klasické vklákna
  - Podpora viac CPU jadier

Mazanie entít

- Spravme si tlačítko na mazanie hneď v tabuľke
- https://mui.com/material-ui/material-icons/

```
const handleDeleteClick = (userId: number | undefined) => {
   fetch(`${restURL}/users/${userId}`, {
      method: 'DELETE'
   }).then(() => setUsers(users.filter(user => user.id !== userId)))
      .catch(error => setError(new Error("Could not delete user", {cause: error})));
};
```

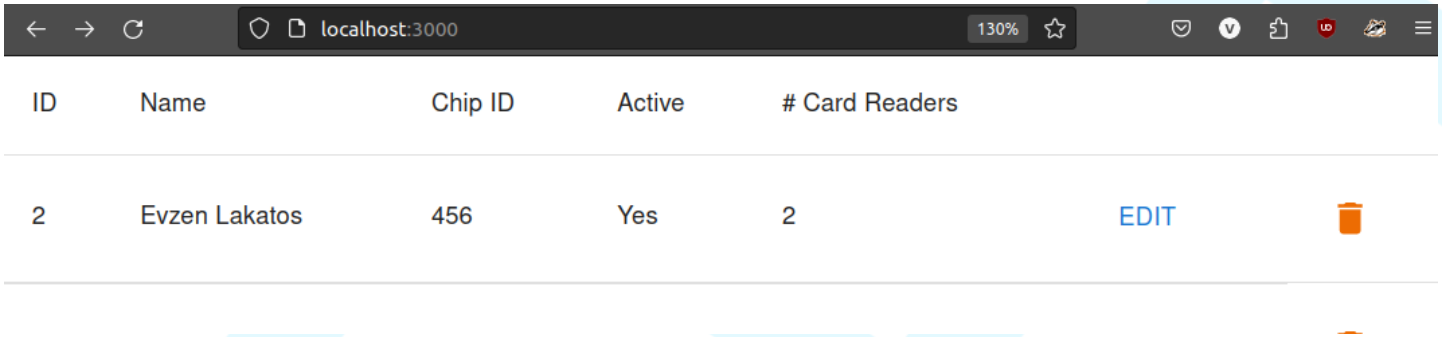
```
import DeleteIcon from '@mui/icons-material/Delete';
. . .

<TableCell>
   <IconButton onClick={() => handleDeleteClick(user.id)}
            color='warning'><DeleteIcon/></IconButton>
</TableCell>
```

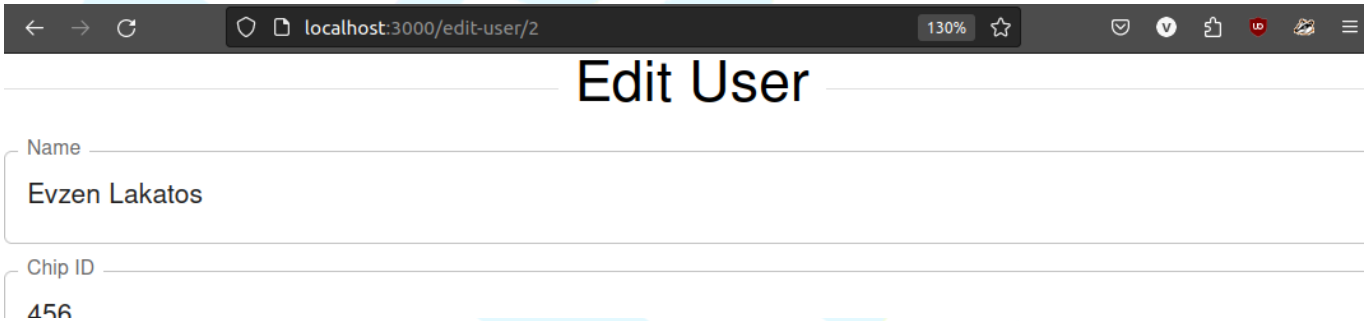
ID	Name	Chip ID	Active	# Card Readers	
2	Evzen Lakatos	456	Yes	2	
4	fero	7852	Yes	1	

Editácia entity (idea)

- V tabuľke bude mať každý riadok tlačítko EDIT



- Klik nás presmeruje na novú stránku



Viac stránková aplikácia

Nainštalujeme router

```
npm i react-router
```

main.tsx:

```
createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <BrowserRouter>
      <Routes>
        <Route index element={<App/>}/>
        <Route path="/users/edit/:id" element={<UserEdit />}/>
      </Routes>
    </BrowserRouter>
  </StrictMode>,
)
```

Implementácia UserEdit.tsx

- Vytiahneme si ID entity z URL parametra

```
export function UserEdit() {
   const {id} = useParams();


  ...
}
```

Implementácia UserEdit.tsx

- Stiahneme si usera a vš. chip readerov

```
export function UserEdit() {
  . . .
  useEffect(() => {
   fetch(`${restURL}/users/${id}`)
      .then(response => {
        if (!response.ok) {
          throw new Error(`${response.status}`);
        }
        return response.json();
      })
      .then(data => setUser(data as User))
      .then(() => setLoadedUser(true))
      .catch(error => setError(new Error(`Failed to fetch user: ${error}`)));

   fetch(`${restURL}/chip-readers`)
      ...;

}, [id]);

  . . .
}
```

Implementácia UserEdit.tsx

- Môžeme poriešiť "chybové" situácie

```
if (error) {
   return (
      <Box sx={{display: 'flex'}}>
         <Alert severity="error">{error.message}</Alert>
      </Box>
   )
}

if (!loadedUser || !loadedChipReaders) {
   return (
      <Box sx={{display: 'flex'}}>
         <CircularProgress/>
      </Box>
   )
}
```

Implementácia UserEdit.tsx

- Komponent bude vracať iné komponenty (TextField) na editovanie usera

- Pozor, premennú user nesmieme meniť
- Zavoláme setUser a vytvoríme kópiu súčasného usera so zmeneným menom
- Na to má JS/TS špeciálnu syntax
  - const novýUser = {...starýUser, name: "Nové Meno"}

Implementácia UserEdit.tsx

```
<Stack spacing={2}>
  <Divider component="div" role="presentation">
    <Typography variant="h4">Edit user</Typography>
  </Divider>
  <TextField
    fullWidth
    label="Name"
    value={user.name}
    error={!isUserNameValid(user?.name)}
    helperText={!isUserNameValid() ? "Name must not be empty" : ""}
    onChange={(e) => {
      setUser({...user, name: e.target.value})
    }}/>
  ...
</Stack>
```

Implementácia UserEdit.tsx

- Boolean atribúty môžeme zobraziť cez Switch

```
<Stack spacing={2}>
   ...
   <FormControlLabel label="Active" control={
   <Switch
      checked={user.active}
      onChange={(e) => {
         setUser({...user, active: e.target.checked})
      }}
   />
 }/>
      ...
</Stack>
```

Implementácia UserEdit.tsx

- Vieme aj validovať vstupy užívateľa

```
. . .

function isUserNameValid(userName?: string): boolean {
   return userName? userName.length > 0 : false;
}



return (
 . . .
 <TextField
   . . .
   error={!isUserNameValid(user?.name)}
   helperText={!isUserNameValid(user?.name) ? "Name must not be empty" : ""}
   . . .
 />
 . . .
);
```

Implementácia UserEdit.tsx

- Funkcia na uloženie užívateľa

```
const navigate = useNavigate();

async function handleSubmit() {
   try {
      await fetch(`${restURL}/users`, {
         method: 'PUT',
         headers: {
            'Content-Type': 'application/json'
         },
         body: JSON.stringify(user)
      })
   } catch (error) {
      setError(new Error("Failed to save user", {cause: error}))
   }
   navigate('/');
}
```

- Funkcia navigate nás po uložení vráti na hlavnú stránku

Implementácia UserEdit.tsx

- Tlačidlá na uloženie a zrušenie

```
<ButtonGroup>
  <Button
   disabled={!isUserNameValid(user?.name)}
   endIcon={<CheckIcon/>}
   variant="contained"
   color="primary"
   onClick={handleSubmit}>
    Save
  </Button>
  <Button variant="contained" color="inherit" onClick={() => navigate('/')}>
    Cancel
  </Button>
</ButtonGroup>
```

Úprava UserList.tsx

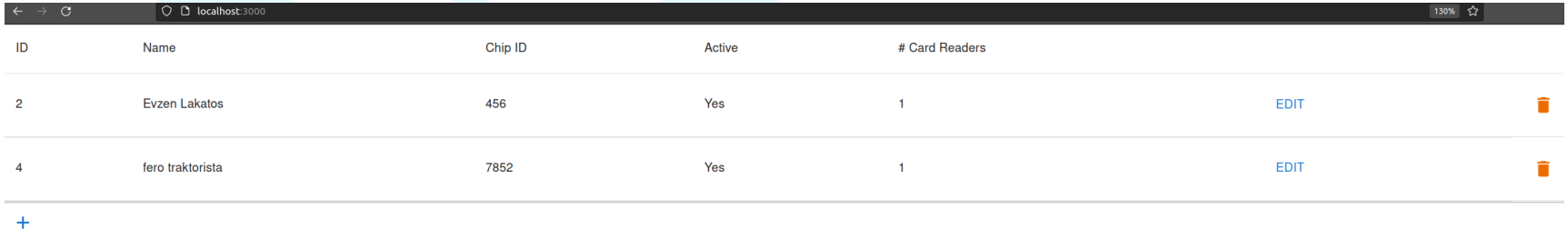
```
const handleEditClick = (userId: number | undefined) => {
    if (userId === undefined) {
        return;
    }

    navigate(`/edit-user/${userId}`);
};
```

```
<TableBody>
  {users.map((user) => (
    <TableRow onClick={() => handleRowClick(user)}>
      . . .
      <TableCell>
        <Button onClick={() => handleEditClick(user.id)}>Edit</Button>
      </TableCell>
      . . .
))}
</TableBody>
```

Pridanie entity

- Analogicky ako editácia
- Akurát pridáme tlačítko + na pridanie entity pod tabuľku
- Upravíme UserEdit
  - Ak nemá *id* v *useParams*, tak robíme **create**



Otázky?