CHAT

SEGÉDLET

Készítette: Rajacsics Tamás (rajacsics.tamas@aut.bme.hu)

A gyakorlat célja egy multiplatform alkalmazás készítése Web alapú technológiák segítségével.

TARTALOM

1 Bevezetés

1.1 Előismeretek és felhasznált eszközök

A gyakorlat feltételezi a következő előismereteket:

- JavaScript és TypeScript
- HTML és CSS
- React
- PWA

Használt eszközök:

- Visual Studio Code
    - Debugger for Chrome, vagy Debugger for Firefox
    - npm

1.2 A projekt létrehozása és beállítása

1) Futtassuk a projekt létrehozó scriptet

```
npx create-react-app chat-ui --template typescript
```

2) Ez létrehoz egy könyvtárat chat-ui néven, amiben indítsuk el a Visual Studio Code-ot (Open Folder).
3) indítsuk el a szervert és fordítót, ami minden mentésre újra betölti az oldalt.

```
npm start
```

4) Írjuk felül az index.css-t a függelékben találhatóval, hogy fejlesztés közben rendesen lássuk is az alkalmazásunkat.
5) Állítsuk be a tsconfig.json-ben a noImplicitAny-t false-ra.
6) Opcionális: F5-öt megnyomva és Chrome-ot kiválasztva létrejön a launch.json fájl, ami a debugger beállításait tartalmazza. Módosítsuk, hogy azonos porton működjön, mint a szerver.

Akkor sikerült minden beállítani, ha minden mentésre újra tölt a böngésző.

2 TypeScript

Az alkalmazás logikáját írjuk meg TypeScript segítségével. A szerverhez kapcsolódást WebSockettel tehetjük meg, ami kétirányú kommunikációt biztosít. Ez azt is jelenti, hogy a hívásaink nem kérés-válasz alapon működnek, hanem küldhetünk csomagokat a szervernek, ami ettől függetlenül küld csomagokat nekünk. Az ilyen típusú interfészt nem lehet/érdemes távoli eljáráshívással modellezni és implementálni – helyette kimenő és bejövő csomagok kezelésével oldunk meg mindent.

2.1 chat.d.ts

Első lépés a definíciós fájlok megszerzése, vagy megírása. Általános esetben ezeket megkapjuk, itt most mi írjuk be a típusokat. Hozzunk létre egy chat.d.ts fájlt és vegyük fel a szerver által használt struktúrákat.

```
export interface MessageDto
{
    id: number;
    timeStamp: string;
    referenceTo: number; // 0: normal message, +: update, -: delete
    senderId: string;
    contentType: number;
    content: string;
}
```

```
export interface UserDto
{
    id: string;
    displayName: string;
    tag: string;
    lastSeen: string;
}
```

```
export interface ConversationDto
{
    channelId: string;
    parentChannelId: string;
    name: string;
    description: string;
    data: string;
    state: number; // disconnected, outgoingRequest, incomingRequest, accepted, group
    access: number; // none, read, write, admin
    notificationLevel: number; // none, gray, push
    unreadCount: number;
    memberIds: string[];
    lastMessages: MessageDto[];
}
```

```
export interface InboxDto
{
    user: UserDto;
    contacts: UserDto[];
    conversations: ConversationDto[];
}
```

```
export type OutgoingPacket =
    { type: "login", email: string, password: string, staySignedIn: boolean } |
    { type: "loginWithToken", token: string } |
    { type: "register", email: string, password: string, displayName: string, staySignedIn: boolean } |
    { type: "contactRequest", email: string, firstMessage: string } |
    { type: "message", channelId: string, referenceTo: number, contentType: number, content: string };
```

```
export type IncomingPacket =
    { type: "error", message: string } |
    { type: "login", query: string, token: string, inbox: InboxDto } |
    { type: "message", channelId: string, message: MessageDto } |
    { type: "conversationAdded", conversation: ConversationDto } |
    { type: "conversationRemoved", channelId: string } |
    { type: "user", user: UserDto };
```

Ezek kódot nem generálnak, csak a típusellenőrzéshez és kódkiegészítéshez kellenek.

2.2 Helyi adatbázis

Hozzunk létre egy Proxy osztályt, ami a szerver interfészét modellezi.

```
class Proxy
{
   private ws: WebSocket;

   constructor()
   {
      this.ws = new WebSocket( "ws://echo.websocket.org/" );
      this.ws.addEventListener( "open", () =>
      {
         this.ws.send( "Hello" );
      } );
      this.ws.addEventListener( "message", e =>
      {
      } );
   }
}
export var proxy = new Proxy();
```

Teszteljük le, hogy tudunk-e kapcsolódni, küldeni és fogadni adatot, majd töröljük a send hívást.

Írjuk meg a küldő függvényt, ami a megfelelő formátumban elküldi a csomagot a szervernek.

```
private sendPacket( packet: OutgoingPacket )
{
   this.ws.send( JSON.stringify( packet ) );
}
```

Implementáljunk a helyi tárat. Jelen esetben ez a szerverről bejött adatok tárolását jelenti.

```
inbox: InboxDto | null = null;
```

Majd kezeljük a bejövő csomagokat.

```
this.ws.addEventListener( "message", e =>
{
   let p = <IncomingPacket>JSON.parse( e.data );
   switch ( p.type )
   {
      case "error":
         alert( p.message );
         break;
      case "login":
         this.inbox = p.inbox;
         break;
      case "message":
         let cid = p.channelId;
         this.inbox!.conversations.find( x => x.channelId === cid )?.lastMessages.push( p.message );
         break;
      case "conversationAdded":
         this.inbox!.conversations.push( p.conversation );
         break;
   }
} );
```

2.3 Események publikálása

Szükségünk van egy mechanizmusra, amivel a komponensek fel/le tudnak iratkozni eseményekre, amiket publikálni szeretnénk. Ennek egyik lehetséges módja az addEventListener minta. Ehhez készítsünk egy támogató osztályt EventProducer néven.

```
export class EventProducer
{
   private listeners: { type: string, listener, obj?: Object }[] = [];

   addEventListener( type: string, listener, obj?: Object )
   {
      this.listeners.push( { type, listener, obj } );
   }
   removeEventListener( type: string, listener )
   {
      this.listeners.splice( this.listeners.findIndex( x => x.type === type && x.listener === listener ), 1 );
   }
}
```

Továbbá szükségünk van az esemény elsütésére.

```
protected dispatch( type: string, ...args )
{
    for ( let listener of this.listeners.filter( x => x.type === type ) )
        listener.listener.call( listener.obj, ...args );
}
```

Itt a spread operátort használjuk a tetszőleges számú paraméter átvételére.

Illetve a komponensek támogatására készítsünk egy removeAll függvényt is.

```
removeAllEventListener( obj: Object )
{
    if ( !obj )
        throw new Error( "Must specify object" );
    this.listeners = this.listeners.filter( x => x.obj !== obj );
}
```

2.3.1 EventMap

Jelenleg a listenernek nincs típusa (any), mert nem tudjuk, hogy milyen eseményhez milyen szignatúra kell. Hogyan tudjuk ezt típusossá tenni? A megoldás az EventMap szerkezet, amit a nyelv kezelni tud.

Kérjünk be egy olyan típusparamétert (generics), ami minden eseményhez hozzárendel egy függvényszignatúrát. Ennek le tudjuk kérdezni mind a kulcsait (keyof), mind az értékeit.

A type típusát cseréljük le type: keyof M típusra. Ez azért fog helyesen működni, mert a keyof M egy unió típust készít az összes lehetséges kulccsal.

Az egyes függvényeket pedig írjuk át a következőképpen

```
addEventListener<K extends keyof M>( type: K, listener: M[ K ], obj?: Object )
```

2.3.2 EventMap felhasználása

Definiáljunk egy EventMap-et. Vegyük fel az összes lehetséges eseményt a hozzá tartozó függvényszignatúrával a Proxy.ts-ben.

```
interface ProxyEventMap
{
    "login": () => void;
    "message": ( channelId: string, message: MessageDto ) => void;
    "conversation": ( channelId: string ) => void;
}
```

A Proxy osztályt származtassuk le az EventProducer-ból.

```
class Proxy extends EventProducer<ProxyEventMap>
```

Majd süssük el az eseményeket a megfelelő helyeken. (login üzenetben a login-t, stb.)

```
case "login":
   this.inbox = p.inbox;
   this.dispatch( "login" );
   break;
...
this.dispatch( "message", cid, p.message );
...
this.dispatch( "conversation", p.conversation.channelId );
```

Próbáljuk ki, regisztráljunk be a saját címünkkel, vagy valami egyedi kóddal (a szerver nem ellenőrzi az email cím helyességét)! Írjunk kódot a register megívására, és nézzük meg, hogy jön-e login csomag!

A chat szerver címe: "wss://raja.aut.bme.hu/chat/"

3 React

3.1 Első komponens létrehozása: Login.tsx

Hozzunk létre egy fájlt Login.tsx néven a következő tartalommal.

```
import React, { Component } from 'react';

export class Login extends Component
{
   render()
   {
      return (
         <div className="login">
            <img src="logo512.png" width="256" />
            <input type="email" placeholder="Email (someone@example.com)" />
            <input type="password" placeholder="Password" />
            <button type="button">Login</button>
            <a href="https://www.google.hu/search?q=privacy">Privacy Policy</a>
         </div> );
   }
}
```

Tegyük be az App komponensbe (figyeljünk az importra). Közben alakítsuk át az Appot osztállyá, hogy azonos formája legyen minden komponensnek.

```
import React, { Component } from 'react';
import { Login } from './Login'

export default class App extends Component
{
  render()
  {
    return (
      <div className="app">
        <Login />
      </div>
    );
  }
}
```

Általában jó ötlet minden komponens gyökerét megjelölni egy CSS névvel, hogy később kényelmesen tudjuk formázni.

3.1.1 Állapot kezelés

Vegyünk fel állapotot a Login komponensben, hogy tudjuk kezelni a bemenetet (másik lehetőség a bemenet kezelésére a DOM-ban tárolt állapot, de akkor nem tudunk gépelés közben segíteni a felhasználónak).

```
state = { email: "", password: "" };
```

Majd szinkronizáljuk az állapotot a vezérlőkkel (value és onChange).

```
<input ... value={ this.state.email } onChange={ e => this.setState( { email: e.target.value } ) } />
```

És ugyanúgy a password mezőre is.

3.1.2 Gomb kezelése

Kezeljük a gombot egy külön függvényben.

```
<button type="button" onClick={ () => this.onClick() }>Login</button>
```

Az implementáció egyelőre üres.

```
onClick()
{
}
```

3.1.3 Feltételes elemek és belső logika

Kezeljük a regisztrációt is ebben a komponensben. Legyen egy állapotunk, hogy éppen regisztrálunk, vagy belépünk, és ennek megfelelően tegyünk ki még egy mezőt, ahova a nevét tudja beírni a felhasználó.

```
state = { email: "", password: "", displayName: "", register: false };
```

Feltételes elemet megoldhatjuk például az && operátorral.

```
{ this.state.register &&
   <input type="text" placeholder="Display Name (Agent Smith)" value={ this.state.displayName }
      onChange={ e => this.setState( { displayName: e.target.value } ) } /> }
```

Illetve a gombra írjuk ki a megfelelő szöveget.

```
<button type="button" onClick={ () => this.onClick() }>
   { this.state.register ? "Register" : "Login" }
</button>
```

Már csak a kettő közötti váltást kell megoldani.

```
<p>{ this.state.register ? "Switch back to " : "Have no account yet? Go and " }
   <a href="" onClick={ e => { e.preventDefault(); this.setState( { register: !this.state.register } ); } }>
      { this.state.register ? "Login" : "Register" }
   </a>
</p>
```

Majd implementáljuk az onClick függvényt.

```
onClick()
{
   if ( this.state.register )
      proxy.sendPacket( { type: "register", email: this.state.email, password: this.state.password,
         displayName: this.state.displayName, staySignedIn: false } );
   else
      proxy.sendPacket( { type: "login", email: this.state.email, password: this.state.password,
         staySignedIn: false } );
}
```

Találjunk ki egy egyedi kódot (pl. neptun kód), és ha valaki beírja ezt a Login űrlapon az email mezőbe, akkor a displayName mező tartalmát írjuk felül a saját keresztnevünkre!

3.2 Funkcionalitás kiemelése: TextInput.tsx

Gyakori, hogy egy komponens kezd túl összetett lenni, vagy egy már létező vezérlőt szeretnénk felokosítani. Ilyen esetekben elkészítünk egy másik komponenst, aminek azonos/hasonló a használata, és lecseréljük a korábban használt helyeken.

Írjunk meg egy jobban működő bemeneti mezőt a beépített <input> helyett. Hozzunk létre egy új fájlt TextInput.tsx néven.

3.2.1 Props

Az interfésze a következő.

```
export interface TextInputOptions
{
   value?: string;
   onChange?: ( value: string ) => void;
   type?: "text" | "password" | "email";
   placeholder?: string;
   onEnter?: () => void;
   autofocus?: boolean;
};
```

Támogatja az eddig is használt value és onChange állapotkezlést, placeholder szöveget, illetve ad eseményt az Enter megnyomására is.

A props típusossá tétele típusparaméterrel lehetséges.

```
export class TextInput extends Component<TextInputOptions>
```

Belső állapota a beírt szöveg, illetve szükségünk van arra is, hogy fokuszált-e.

```
state = { value: this.props.value, focus: false };
```

A render a következő.

```
return (
   <div className="text-input">
      <input type={ this.props.type ?? "text" } value={ this.state.value }
         onChange={ e =>
         {
            this.setState( { value: e.target.value } );
            this.props.onChange?.( e.target.value );
         } }
         onBlur={ () => this.setState( { focus: false } ) }
         onFocus={ () => this.setState( { focus: true } ) } />
      <div className="focus-indicator"></div>
      <label className={ this.state.value || this.state.focus ? "subsided" : "" }>
         { this.props.placeholder }
      </label>
   </div> );
```

Van benne egy input, illetve a placeholder szöveg akkor is látszik, ha beleírtunk valamit.

3.2.2 Feltételes attribútumok

Az onEnter és autofocus opcionális tulajdonságokat feltételesen adjuk hozzá. A renderben hozzunk létre egy objektumot, aminek megadjuk az attribútum-érték párosokat.

```
let attrs = {} as any;
if ( this.props.autofocus )
   attrs.autoFocus = true;
if ( this.props.onEnter )
   attrs.onKeyDown = e =>
   {
      if ( e.keyCode === 13 )
         this.props.onEnter!();
   };
```

És használjuk fel a spread operátorral az inputon.

```
{ ...attrs }
```

3.2.3 Komponensenkénti CSS

Adjunk hozzá egy új fájlt TextInput.css néven, aminek a tartalma a függelékben található. A komponensben import segítségével lehet használni.

```
import './TextInput.css'
```

3.2.4 Felhasználása

Miután kész a TextInput, mindenhol használjuk ezt input helyett. Az egyetlen különbség, hogy az onChange eseménykezelőben e.target.value helyett simán e-t használunk. A Login komponens első TextInput-ján állítsunk be autoFocus-t is, illetve mindegyiken onEnter-t.

```
<TextInput type="email" placeholder="Email (someone@example.com)" value={ this.state.email }
  onChange={ e => this.setState( { email: e } ) } onEnter={ () => this.onClick() } autofocus={ true } />
```

3.3 Gyerek komponens funkcionalitásának publikálása

Készítsünk egy többször is felhasználható input+button komponenst TextInputAndButton néven. Használjuk fel a már létező TextInput komponenst és publikáljuk ki a teljes funkcionalitását.

3.3.1 Létező Props bővítése

Bővítsük a már létező TextInputOptions két mezővel.

```
export interface TextInputAndButtonOptions extends TextInputOptions
{
  buttonContent?: string;
  onClick?: ( text: string ) => boolean | void;
}
```

Használjuk fel. Figyeljük meg a spread operátor használatát, ami átadja az összes props-ot a belső komponensünknek.

```
export class TextInputAndButton extends Component<TextInputAndButtonOptions>
{
   onClick()
   {
   }

   render()
   {
      return (
         <div className="text-input-and-button">
            <TextInput { ...this.props } onEnter={ () => this.onClick() } />
            <button type="button" onClick={ () => this.onClick() }>
               { this.props.buttonContent }
            </button>
         </div>
      );
   }
}
```

3.3.2 REF

A TextInput értékét csak akkor tudjuk kiolvasni, ha ráteszünk egy ref-et (vagy tároljuk az állapotát, de ahhoz sok mindent át kéne írni itt).

```
textInput = React.createRef<TextInput>();
```

Majd tegyük rá a komponensre.

```
<TextInput { ...this.props } ref={ this.textInput } onEnter={ () => this.onClick() } />
```

Használjuk fel az onClick-ben.

```
if ( this.props.onClick?.( this.textInput.current?.state.value ?? "" ) )
   this.textInput.current?.setState( { value: "" } );
```

Ha true-t ad vissza, akkor töröljük az inputot – kényelmes használni.

3.4 ALKALMAZÁS FELÜLETE: MAIN.TSX

Hozzuk létre a következő komponenseket (App, Login és TextInputAndButton már készen vannak): Main, LeftPane, RightPane, ConversationCard és MessageCard.

App

- Login
- Main
  - LeftPane
    - TextInputAndButton
    - ConversationCard

RightPane

MessageCard

TextInputAndButton

Az alkalmazás felületét a Main komponens implementálja, ha be vagyunk lépve, különben a Login komponens látszik.

A Main komponens kiteszi a bal és jobb oldalt egymás mellé (row CSS ezt megoldja), és kezeli a kiválasztott chatet.

```
export class Main extends Component
{
   state = { selectedConversation: undefined as ( ConversationDto | undefined ) };
   render()
   {
      return (
         <div className="main row">
            <LeftPane
               inbox={ proxy.inbox! }
               selectedConversation={ this.state.selectedConversation }
               onSelect={ c => this.setState( { selectedConversation: c } ) } />
            <RightPane conversation={ this.state.selectedConversation } />
         </div>
      );
   }
}
```

Egyelőre még nincs meg a bal és jobb oldal, így ez nem fordul.

3.5 Listák és külső események

3.5.1 LeftPane.tsx

A LeftPane megkapja az inboxot, illetve kezeli a meghívást és chat kiválasztását.

A komponens props-a és render kódja a következő. Vegyük észre a lista generálást.

```
export class LeftPane extends Component<{
    inbox: InboxDto, selectedConversation: ConversationDto | undefined,
    onSelect: ( c: ConversationDto ) => void
}>
{
    render()
    {
        return (
            <div className="left-pane">
                <p className="my-tag">My tag: { this.props.inbox.user.tag }</p>
                <TextInputAndButton type="text" placeholder="Add user by Tag (Name#123)"
                    buttonContent="Inv"
                    onClick={ text => this.sendContactRequest( text ) } />
                <div className="conversations">
                    { this.props.inbox.conversations.map( x =>
                        <ConversationCard
                            key={ x.channelId }
                            conversation={ x }
                            selected={ x === this.props.selectedConversation }
                            onSelect={ () => this.props.onSelect( x ) } /> ) }
                </div>
            </div>
        );
    }
}
```

A sendContactRequest kódja egyszerű.

```
sendContactRequest( email: string )
{
    proxy.sendPacket( { type: "contactRequest", email, firstMessage: "Hello" } );
    return true;
}
```

Az egyetlen hiányzó funkcionalitás az új chat esemény kezelése.

```
componentDidMount()
{
    proxy.addEventListener( "conversation", c => this.forceUpdate(), this );
}

componentWillUnmount()
{
    proxy.removeAllEventListener( this );
}
```

Amikor jön egy üzenet, frissítjük a komponenst, illetve leiratkozunk az eseményről, amikor a komponensünk megszűnik.

3.5.2 ConversationCard.tsx

Szintén külső eseményeket kezel a ConversationCard komponens. Amikor jön egy üzenet, akkor frissíteni kell. A render a következő.

```
export class ConversationCard extends Component<{
   conversation: ConversationDto,
   selected: boolean,
   onSelect: () => void
}>
{
   render()
   {
      let lastMessage = this.props.conversation.lastMessages.length > 0 ?
         this.props.conversation.lastMessages[ this.props.conversation.lastMessages.length - 1 ] : null;
      return (
         <div className={ "conversation-card" + ( this.props.selected ? " selected" : "" ) }
            onClick={ () => this.props.onSelect() }>
            <div className="row">
               <span className="channel-name">{ this.props.conversation.name }</span>
               <span className="time">
                  { lastMessage && new Date( lastMessage.timeStamp ).toLocaleTimeString() }
               </span>
            </div>
            <span className="last-message">{ lastMessage?.content }</span>
         </div>
      );
   }
}
```

A külső események kezelése ismét a componentDidMount és componentWillUnmount életciklus kezelő függvényekkel lehetséges.

```
componentDidMount()
{
   proxy.addEventListener( "message", ( cid, m ) =>
   {
      if ( cid === this.props.conversation.channelId )
         this.forceUpdate();
   }, this );
}
componentWillUnmount()
{
   proxy.removeAllEventListener( this );
}
```

Ha van szám a neptun kódodban, akkor ne az idejét írd ki az utolsó üzenetnek, hanem a dátumát!

3.6 Fragment: RightPane.tsx

Nézzük meg a RightPane komponens render kódját. Ha több elemet kell visszaadjuk, akkor használhatjuk a <Fragment> kulcsszót (vagy a rövidítését: <>). Így nem kell őket külön tömbbe tenni. A kód többi része ismert elemekből áll.

```
export class RightPane extends Component<{ conversation?: ConversationDto }>
{
    componentDidMount()
    {
        proxy.addEventListener( "message", ( cid, m ) =>
        {
            if ( cid === this.props.conversation?.channelId )
                this.forceUpdate();
        }, this );
    }
```

```
    componentWillUnmount()
    {
        proxy.removeAllEventListener( this );
    }
```

```
    onSend( text: string )
    {
        proxy.sendPacket( { type: "message", channelId: this.props.conversation!.channelId, referenceTo: 0
, contentType: 0, content: text } );
        return true;
    }
```

Minden küldésnél írj a szöveg elé egy kódot így: „X: text”, ahol az első karakter (X) a saját neptun kódod első karaktere legyen.

```
  render()
  {
    return (
      <div className="right-pane column">
        { this.props.conversation &&
          <>
            <div className="conversation-header">
              <p>{ this.props.conversation?.name }</p>
            </div>
            <div className="messages">
              { this.props.conversation?.lastMessages.map( x =>
                <MessageCard key={ x.id } message={ x }
                  own={ x.senderId === proxy.inbox?.user.id } /> ) }
            </div>
            <div className="send-message row">
              <TextInputAndButton type="text" placeholder="Type something awesome here..."
                buttonContent="Send" onClick={ x => this.onSend( x ) } />
            </div>
          </>
        }
      </div>
    );
  }
}
```

3.7 PureComponent: MessageCard.tsx

Ez a komponens nagyon egyszerű. Csak a props-ban tárolt adatoktól függ a render, így lehet PureComponent (akkor is lehetne, ha lenne egyszerű állapota).

```
export class MessageCard extends PureComponent<{ message: MessageDto, own: boolean }>
{
  render()
  {
    return (
      <div className={ "message-card" + ( this.props.own ? " own" : "" ) }>
        <div className="bubble">
          <span className="text">{ this.props.message.content }</span>
          <span className="time">
            { new Date( this.props.message.timeStamp ).toLocaleTimeString() }
          </span>
        </div>
      </div>
    );
  }
}
```

4 Progressive Web App

4.1 Mobilbarát alkalmazás

Jelenleg az alkalmazásunk responsive ugyan, de vékony képernyőn (mobil) nem igazán működik. Oldjuk meg ezt úgy, hogy vagy csak a bal, vagy csak a jobb oldalt jelenítjük meg kicsi képernyőn.

4.1.1 Mobil RightPane.tsx

Módosítsuk a jobb oldal felső részét úgy, hogy legyen egy Back gomb is rajta.

Ehhez a gombnyomást vissza kell küldjük a Main-nek, hogy kezelni tudja a váltást. Ezt (onBack) vegyük fel a props-ba.

```
export class RightPane extends Component<{ conversation?: ConversationDto, onBack: () => void }>
```

A gombot vegyük fel (a row CSS-t tegyük rá, hogy egymás mellé kerüljenek).

```
<div className="conversation-header row">
  <button type="button" className="only-narrow"
    onClick={ () => this.props.onBack() }>Back</button>
  <p>{ this.props.conversation?.name }</p>
</div>
```

4.1.2 Mobil Main.tsx

Tegyünk rá egy CSS-t, ami azt mutatja, hogy a bal, vagy jobb oldal van fókuszban, illetve Back gombra állítsuk be undefinedra a kiválasztott chatet.

```
let className = "main row " + ( this.state.selectedConversation ? "right" : "left" );
return (
  <div className={ className }>
    <LeftPane
      inbox={ proxy.inbox! }
      selectedConversation={ this.state.selectedConversation }
      onSelect={ c => this.setState( { selectedConversation: c } ) } />
    <RightPane conversation={ this.state.selectedConversation }
      onBack={ () => this.setState( { selectedConversation: undefined } ) } />
  </div>
);
```

A jobb oldalnak van egy bordere, ami az elválasztást végzi, de szükségtelen, ha csak a jobb oldal látszik. Oldjuk meg CSS-ben, hogy csak akkor legyen ott, ha mindkét oldal látszik!

4.2 Konfigurálás és telepítés

4.2.1 manifest.json

Első lépés a manifest fájl létrehozása és megírása. Az új project varázsló létrehoz egyet, így nekünk csak szerkeszteni kell.

Írjuk át a következő beállításokat.

```
"short_name": "MyChat",
"name": "My Chat",
```

4.2.2 Service Worker

A Service Worker is meg van írva, nekünk csak engedélyezni kell az index.tsx utolsó sorában.

```
serviceWorker.register();
```

4.2.3 Publikálás és package.json

Az utolsó változtatást a package.json fájlban kell tegyük. Adjuk hozzá a következőt.

```
"homepage": ".",
```

Ez lehetővé teszi, hogy bárhol futtassuk az alkalmazást, ne csak a gyökérben menjen.

Végül fordítsunk egy release buildet, ahogy az a readme.md fájlban meg van adva.

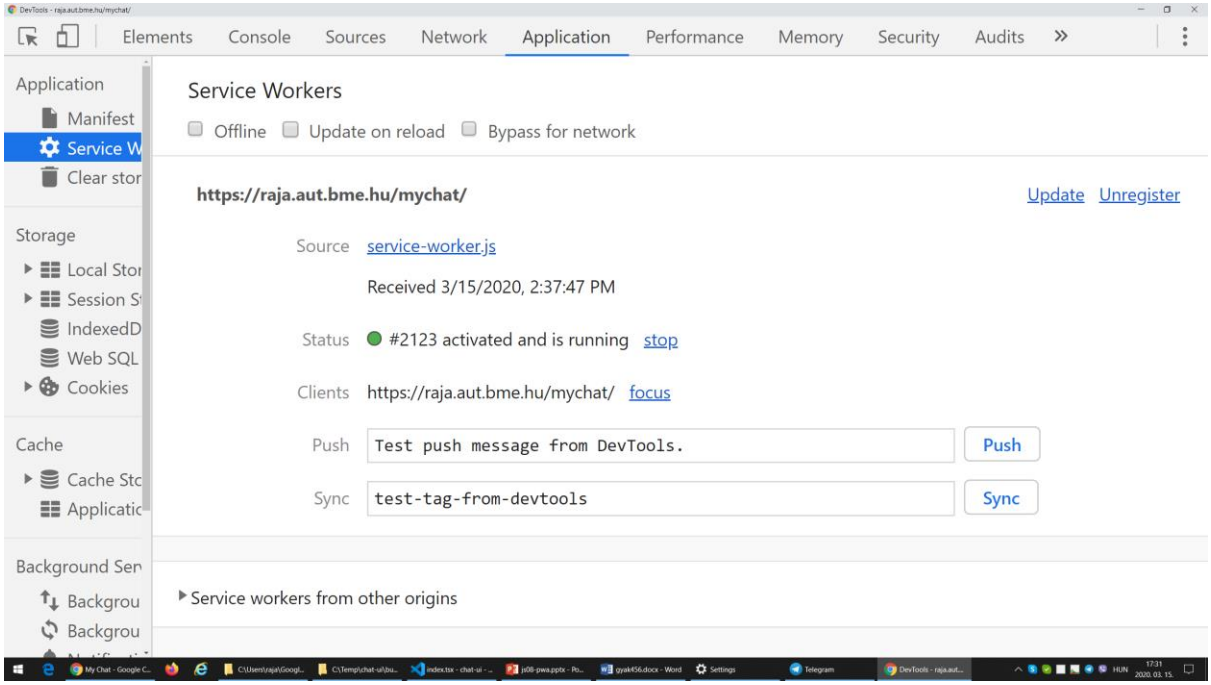
```
npm run build
```

Az eredményt másoljuk fel a végső helyére a webszerveren, ami HTTPS-ként tudja visszaadni. A publikálás fejezet GitHub-ra teszi ki.

Próbáljuk ki.

4.3 PWA eszközök

Menjünk be a DevTools Application menüjébe, és nézzük meg a Manifest, Service Workers, és Cache Storage menüket.


Elements
Console
Sources
Network
Application
Performance
Memory
Security
Audits
Application
Manifest
Service W
Clear stor
Storage
Local Stor
Session S
IndexedD
Web SQL
Cookies
Cache
Cache Stc
Applicatic
Background Ser
Backgrou
Backgrou
Service Workers
Offline
Update on reload
Bypass for network
https://raja.aut.bme.hu/mychat/
Update
Unregister
Source
service-worker.js
Received 3/15/2020, 2:37:47 PM
Status
#2123 activated and is running
stop
Clients
https://raja.aut.bme.hu/mychat/
focus
Push
Test push message from DevTools.
Push
Sync
test-tag-from-devtools
Sync
Service workers from other origins

5 Publikálás

Bárhova publikálhatjuk a forrást, itt most GitHub-ra fogjuk.

5.1 GitHub

Első körben commitoljunk mindent (VSCode bal oldalon a 3. ikon – Source Control), hogy létrejöjjön egy csomag, amit fel tudunk majd tenni.

Regisztráljunk GitHub-on (https://github.com/), ha még nincs accountunk, és hozzunk létre egy repository-t my-chat néven (vagy akármi más néven). A következő két parancsot hajtsuk végre a kódunk gyökérkönyvtárában. A linket a github adja.

```
git remote add origin https://github.com/YOURACCOUNT/my-chat.git
git push -u origin master
```

Ha most frissítjük a github oldalunkat, akkor látnunk kell az fájljainkat és a README.md tartalmát.

5.2 Release verzió

Adjuk hozzá a korábban elkészített build könyvtárat a Source Control-hoz a .gitignore fájl szerkesztésével. Töröljük ki, vagy kommentezzük ki a /build sort.

```
# production
#/build
```

Ezek után tegyük fel ezt is GitHub-ra (commit és push).

5.3 GitHub Pages

GitHub képes hostolni statikus oldalakat. A projekt oldalán menjünk be a Settings-be, majd engedélyezzük a GitHub Pages-t a Source állításával:

```
master branch
```

Ezek után működik az oldalunk a megadott linken, csak hozzá kell tenni a /build-et a végére.

https://ACCOUNT.github.io/my-chat/build/

6 Függelék

6.1 TextInput.css

```
.text-input {
    position: relative;
    padding: 12px 0;
}
.text-input input {
    width: 100%;
    padding: 6px 6px;
    border: 0;
    outline: none;
}
.text-input label {
    position: absolute;
    color: gray;
    font-style: italic;
    left: 6px;
    top: 18px;
    pointer-events: none;
    transition: all 0.15s;
}
.text-input label.subsided {
    left: 0;
    top: 0;
    font-size: 9pt;
    transition: all 0.15s;
    color: lightgray;
}
.text-input .focus-indicator {
    position: absolute;
    bottom: 12px;
    width: 100%;
    border: solid gray;
    border-width: 0 0 1px 0;
}
.text-input input:focus+.focus-indicator {
    border-color: black;
    border-color: var( --focus-indicator-color, black);
    border-width: 0 0 2px 0;
}
```

6.2 INDEX.CSS

```
html, body, div {
  margin: 0;
  padding: 0;
}

* {
  box-sizing: border-box;
}

html, body, main, .app, .main {
  width: 100%;
  height: 100%;
}

body {
  font-family: 'Segoe UI', sans-serif;
  font-size: 16px;
  line-height: 1.5;
}

input, button {
  font-family: inherit;
  font-size: inherit;
  line-height: inherit;
}

p {
  margin: 0 0 6px 0;
}

input {
  padding: 4px 6px;
}

button {
  padding: 6px 12px;
  background-color: #00A5C6;
  color: white;
  font-weight: 600;
  border: none;
  border-radius: 2px;
}

button:hover {
  background-color: #00BBE0;
```

```
}

.row {
  display: flex;
}

.row>* {
  min-width: 0;
  flex: 1;
}

.column {
  display: flex;
  flex-direction: column;
}

.column>* {
  min-height: 0;
  flex: 1;
}

.login {
  max-width: 400px;
  margin: 0 auto;
  text-align: center;
}

.left-pane {
  flex: 0 0 300px;
}

.my-tag {
  padding: 6px 10px;
  font-weight: 500;
}

.text-input-and-button {
  display: flex;
  align-items: center;
}

.text-input-and-button>*:first-child {
  flex: 1;
}

.left-pane .text-input-and-button {
  margin: 6px 10px;
```

```
}

.left-pane .text-input-and-button button {
  flex: 0 0 auto;
  margin: 0;
}

.conversation-card {
  cursor: pointer;
  padding: 6px 10px;
  overflow: hidden;
  text-overflow: ellipsis;
}

.conversation-card:hover {
  background-color: #eee;
}

.conversation-card .channel-name {
  font-weight: 500;
  color: black;
}

.conversation-card .time {
  text-align: right;
  color: #888;
}

.conversation-card .last-message {
  color: #888;
  text-overflow: ellipsis;
  white-space: nowrap;
}

@media (max-width: 650px) {
  .left .right-pane {
    display: none;
  }
  .right .left-pane {
    display: none;
  }
  .left .left-pane {
    flex: 1;
  }
}

.right-pane {
```

```
  border: solid #aaa;
  border-width: 0 0 0 1px;
}

.right-pane .conversation-header {
  padding: 6px 8px;
  font-weight: 500;
  color: black;
  flex: 0 0 auto;
  align-items: center;
}

.right-pane .conversation-header>p {
  margin: 6px 10px;
}

.right-pane .conversation-header>button {
  flex: 0 0 auto;
}

@media (min-width: 650px) {
  .only-narrow {
    display: none;
  }
}

.messages {
  border: solid #aaa;
  border-width: 1px 0;
  overflow-y: scroll;
}

.message-card {
  margin: 6px 8px;
}

.message-card.own {
  margin: 6px 8px;
  text-align: right;
}

.message-card .bubble {
  display: inline-block;
  background: white;
  padding: 6px 10px;
  border-radius: 5px;
  box-shadow: 1px 1px 5px gray;
```

```
}

.message-card.own .bubble {
  background: #eee;
}

.message-card .time {
  margin: 0 0 0 10px;
  color: #aaa;
}

.right-pane .send-message {
  padding: 6px 8px;
  flex: 0 0 auto;
}
```