Bilancio

├─ backup

│ ├─ BilancioDataBase1.db

│ ├─ BilancioDataBase20.04.2025.db

│ ├─ BilancioDataBase20.04.2025a.db

│ └─ tsconfig.app.json..txtbackup

├─ desktopIcon.png

├─ electron-builder.json

├─ eslint.config.js

├─ index.html

├─ package-lock.json

├─ package.json

├─ playwright.config.ts

├─ README.md

├─ src

│ ├─ assets

│ │ ├─ trayIcon.png

│ │ └─ trayIconTemplate.png

│ ├─ electron

│ │ ├─ dataBase

│ │ │ ├─ dbClass.ts

│ │ │ ├─ dbFunction.ts

│ │ │ ├─ dbQuerySqlString.ts

│ │ │ └─ enum.ts

│ │ ├─ dataBase.ts

│ │ ├─ main.ts

│ │ ├─ menu.ts

│ │ ├─ pathResolver.ts

│ │ ├─ preload.cts

│ │ ├─ resourceManager.ts

│ │ ├─ sharedTypes

│ │ │ ├─ status.backup..js

│ │ │ └─ status.ts

│ │ ├─ tray.ts

│ │ ├─ tsconfig.json

│ │ └─ util.ts

│ └─ ui

│ ├─ App

│ │ ├─ App.css

│ │ └─ App.tsx

│ ├─ assets

│ │ └─ react.svg

│ ├─ BaseChart.tsx

│ ├─ Chart.tsx

│ ├─ components

│ │ ├─ ButtonCancel

│ │ │ ├─ ButtonCancel.module.scss

│ │ │ └─ ButtonCancel.tsx

│ │ ├─ CheckboxSlider

│ │ │ ├─ CheckboxSlider.module.scss

│ │ │ └─ CheckboxSlider.tsx

│ │ ├─ ComboBox

│ │ │ └─ ComboBox.tsx

│ │ ├─ Context

│ │ │ ├─ ElectronProvider.tsx

│ │ │ └─ useOptionsImage.ts

│ │ ├─ DateTimePicker

│ │ │ ├─ DateTimePicer.module.scss

│ │ │ └─ DateTimePicker.tsx

│ │ ├─ FormAddInvoice

│ │ │ ├─ FormAddInvoice.module.scss

│ │ │ └─ FormAddInvoice.tsx

│ │ ├─ FormAddInvoiceDocuments

│ │ │ ├─ FormAddInvoiceDocuments.module.scss

│ │ │ └─ FormAddInvoiceDocuments.tsx

│ │ ├─ FormHomeDate

│ │ │ ├─ FormHomeDate.module.scss

│ │ │ └─ FormHomeDate.tsx

│ │ ├─ GlobalFunctions

│ │ │ └─ GlobalFunctions.ts

│ │ ├─ MainTable

│ │ │ ├─ MainTable.module.scss

│ │ │ └─ MainTable.tsx

│ │ ├─ ModalAddInvoice

│ │ │ ├─ ModalAddInvoice.module.scss

│ │ │ └─ ModalAddInvoice.tsx

│ │ ├─ Navigation

│ │ │ ├─ Navigation.module.scss

│ │ │ └─ Navigation.tsx

│ │ ├─ SingleInput

│ │ │ ├─ SingleInput.module.scss

│ │ │ └─ SingleInput.tsx

│ │ └─ TextInput

│ │ ├─ TextInput.module.scss

│ │ └─ TextInput.tsx

│ ├─ hooks

│ │ ├─ useAllDocumentName.ts

│ │ ├─ useAllInvoices.ts

│ │ ├─ useConnectedTableDictionary.ts

│ │ ├─ useLocalStorage.ts

│ │ ├─ useTableDictionaryDocuments.ts

│ │ └─ useToggle.ts

│ ├─ index.css

│ ├─ main.tsx

│ ├─ pages

│ │ ├─ HomePage

│ │ │ ├─ HomePage.module.scss

│ │ │ └─ HomePage.tsx

│ │ ├─ LayoutPage

│ │ │ ├─ LayoutPage.module.scss

│ │ │ └─ LayoutPage.tsx

│ │ └─ ReportDataPage

│ │ └─ ReportDataPage.tsx

│ ├─ TempStart

│ │ ├─ TempStart.css

│ │ └─ TempStart.tsx

│ ├─ useStatistics.ts

│ └─ vite-env.d.ts

├─ structure.txt

├─ tsconfig.json

├─ tsconfig.node.json

├─ types.d.ts

└─ vite.config.ts

to jest moja struktura plików, chciałbym do mojego programu pisanego w Electron dodać funkcjonalość dodawania faktury w formacie pdf. Użytkownik będzie przeciągał lub wybierał z otwartego okna fakturę w pdf. Będzie ona później dodawana do bazy danych (jeśli będziesz potrzebował strukturę bazy danych to mi napisz). Na tym etapie będzie w pamięci. Faktura będzie dodawana poprzez pole (aktualnie jeszcze go nie ma) znajdujące się w komponencie FormHomeDate.tsx obok przycisku "Dodaj fakturę". Po dodaniu faktury powinna być ona odczytana (chciałbym żeby to było zrobione przez AI offline, ewentualnie online jeśli nie ma takiej możliwości, ponieważ chciałbym żeby uczyła się prawidłowo odczytywać dane z faktury, ewentualnie może być biblioteka w VisualStudio, js, ts jeśli takie są). Z faktury będą odczytywane takie dane jak: nazwa faktury, data faktury, nazwa produktu za który jest wystawiana faktura, ilość produktu, jego cena. Po odczytaniu danych będzie otwierało się okno modala w tym komponencie ModalAddInvoice.tsx, który zawiera komponent FormAddInvoice.tsx. W komponencie w TextInput inputName="invoiceName" powinna się wyświetlić nazwa faktury. Komponent FormAddInvoice.tsx zawiera komponent FormAddInvoiceDocuments.tsx w nim znajdują się pola: <Select<ComboBoxOption> //Dokument Combobox, <Select<ComboBoxOption> //MainType Combobox, <Select<ComboBoxOption> //Type Combobox, <Select<ComboBoxOption> //Subtype Combobox, <TextInput inputName="quantity", <TextInput inputName="price". Pola select będą wyświetlały nazwy dokumentu lub dokumentów jeżeli jest ich więcej niż 1. Pola tekstowe będą wyświetlały odpowiednio ilość produktów oraz cenę jednostkową. Zwróć uwagę że produkt o nazwie "dowód rejestracyjny" będzie zawarty tylko w Select Dokument ponieważ nie ma on podtypów. A produkt o nazwie "tablica rejestracyjna tymczasowa samochodowa jednorzędowa żółta" będzie wyświetlany w czterech Select. W Select Dokument będzie "tablica rejestracyjna", w Select MainType będzie "tymczasowa", w Select Type będzie " samochodowa jednorzędowa", w Select Subtype będzie "żółta". Napisz czy potrzebujesz żebym wysłał Ci kod któregoś z komponentów. Zwróć uwagę że wykorzystywane AI lub biblioteki powinny być darmowe

mam problem z twoim kodem, przeanalizuj mój kod i spróbuj znaleźć błąd: types.d.ts import { STATUS, DataBaseResponse, isSuccess } from './src/electron/sharedTypes/status';

type StaticData = { totalStorage: number; cpuModel: string; totalMemoryGB: number; };

type FrameWindowAction = 'CLOSE' | 'MAXIMIZE' | 'MINIMIZE';

type UnsubscribeFunction = () => void;

declare global { //Context HomePage interface FormValuesHomePage { firstDate: Date | null; secondDate: Date | null; isDeleted?: 0 | 1; }

type Statistics = { cpuUsage: number; ramUsage: number; storageUsage: number; }; type TextTempDataBase = { textNazwa: string; }; //Table export type DictionaryDocuments = { DocumentId: number; DocumentName: string; }

export type DictionaryMainType = { MainTypeId: number; MainTypeName: string; }

export type DictionaryType = { TypeId: number; TypeName: string; }

export type DictionarySubtype = { SubtypeId: number; SubtypeName: string; } type InvoiceTable = { InvoiceId?: number; InvoiceName: string; ReceiptDate: string; DeadlineDate: string | null; PaymentDate: string | null; IsDeleted: 0 | 1; } type InvoiceDetailsTable = { InvoiceId?: number; DocumentId: number; MainTypeId: number | null; TypeId: number | null; SubtypeId: number | null; Quantity: number; Price: number; } type InvoiceSave = { invoice: InvoiceTable; details: InvoiceDetailsTable[]; }; type ReturnInvoiceSave = { lastID: number; changes: number; } //ConnectedTable type AllDocumentsName = { DocumentId: number; DocumentName: string; MainTypeId: number | null; MainTypeName: string; TypeId: number | null; TypeName: string; SubtypeId: number | null; SubtypeName: string; Price: number; }

// type AllInvoicesDb1 = { // InvoiceId: number; // InvoiceName: string; // ReceiptDate: string; // DeadlineDate: string |null; // PaymentDate: string; // IsDeleted: 0 | 1; // InvoiceDetailsId: number; // DocumentId: number; // MainTypeId: number |null; // TypeId: number|null; // SubtypeId: number |null; // Quantity: number; // Price: number; // DocumentName: string; // MainTypeName: string; // TypeName: string; // SubtypeName: string; // } type AllInvoicesDb = { InvoiceId: number; InvoiceName: string; ReceiptDate: string; DeadlineDate: string | null; PaymentDate: string; IsDeleted: 0 | 1; DocumentNames: string; MainTypeNames: string; TypeNames: string; SubtypeNames: string; Quantities: string; Prices: string; [key: string]: string[] | null; }

type AllInvoices = { InvoiceId: number; InvoiceName: string; ReceiptDate: string; DeadlineDate: string | null; PaymentDate: string; IsDeleted: 0 | 1; DocumentNames: string[]; MainTypeNames: string[]; TypeNames: string[]; SubtypeNames: string[]; Quantities: string[]; Prices: string[]; [key: string]: string[] | null; }

type LastRowInvoice = { InvoiceId: number; InvoiceName: string; ReceiptDate: string; DeadlineDate: string; PaymentDate: string | null; IsDeleted: 0 | 1; }

type JakasFunkcja = { jakisTekst: string; jakisNumer: number; } // const STATUS\_SUKCES = 'sukces'; // const STATUS\_BLED = 'blad';

// type PrzykladowaFunkcjaResult = // | { status: typeof STATUS\_SUKCES; dane: JakasFunkcja } // | { status: typeof STATUS\_BLED; komunikat: string }; const STATUS1 = { Sukces: "sukces", Error: "error" } as const type PrzykladowaFunkcjaResult = | { status: typeof STATUS1.Sukces; dane: JakasFunkcja } | { status: typeof STATUS1.Error; komunikat: string };

type EventPayloadMapping = { statistics: Statistics; getStaticData: StaticData; changeView: View; sendFrameAction: FrameWindowAction; getTableDictionaryDocuments: DataBaseResponse; getConnectedTableDictionary:DataBaseResponse; queryToDB: unknown[]; getAllDocumentName: DataBaseResponse; getAllInvoices: AllInvoices[]; getLastRowFromTable: unknown; przykladowaFunkcja: JakasFunkcja; przykladowaFunkcja2: PrzykladowaFunkcjaResult; // addInvoice: unknown; addInvoice: ReturnInvoiceSave; addInvoiceDetails: ReturnInvoiceSave; }; type View = 'CPU' | 'RAM' | 'STORAGE'; type FrameWindowAction = 'CLOSE' | 'MAXIMIZE' | 'MINIMIZE'; interface Window { electron: { subscribeStatistics: ( callback: (statistics: Statistics) => void ) => UnsubscribeFunction; getStaticData: () => Promise; subscribeChangeView: ( callback: (view: View) => void ) => UnsubscribeFunction; sendFrameAction: (payload: FrameWindowAction) => void; getTableDictionaryDocuments: (payload) => Promise<DataBaseResponse>; getConnectedTableDictionary: (tableName, documentId, mainTypeId, typeId, subTypeId) => Promise<DataBaseResponse>; queryToDB: () => Promise; getAllDocumentsName: () => Promise<DataBaseResponse>; getAllInvoices: (payload) => Promise; addInvoice: (payload) => Promise; addInvoiceDetails: (invoice, invoiceDetails) => Promise; getLastRowFromTable: () => Promise; przykladowaFunkcja: (payload, numer) => Promise; przykladowaFunkcja2: (payload, numer) => Promise; }; } }

export { };

src\electron\preload.cts const electron = require('electron');

electron.contextBridge.exposeInMainWorld('electron', { subscribeStatistics: (callback) => { return ipcOn('statistics', (stats) => { callback(stats); }) }, subscribeChangeView: (callback) => ipcOn('changeView', (view) => { callback(view); }), getStaticData: () => ipcInvoke('getStaticData'), sendFrameAction: (payload) => ipcSend('sendFrameAction', payload), getTableDictionaryDocuments: (payload) => ipcInvoke2('getTableDictionaryDocuments', payload), getConnectedTableDictionary: (tableName, documentId, mainTypeId, typeId, subTypeId) => ipcInvoke2('getConnectedTableDictionary', tableName, documentId, mainTypeId, typeId, subTypeId), queryToDB: () => ipcInvoke('queryToDB'), getAllDocumentsName: () => ipcInvoke('getAllDocumentName'), getAllInvoices: (payload) => ipcInvoke2('getAllInvoices', payload), addInvoice: (payload) => ipcInvoke2('addInvoice', payload), addInvoiceDetails: (invoice, invoiceDetails) => ipcInvoke2('addInvoiceDetails', invoice, invoiceDetails), getLastRowFromTable: () => ipcInvoke('getLastRowFromTable'), // przykladowaFunkcja: (payload) => ipcInvoke('przykladowaFunkcja'), przykladowaFunkcja: (payload, numer) => ipcInvoke2('przykladowaFunkcja', payload, numer), przykladowaFunkcja2: (payload, numer) => ipcInvoke2('przykladowaFunkcja2', payload, numer), // addInvoice: (payload) => ipcInvoke('addInvoice'), } satisfies Window["electron"]);

function ipcInvoke( key: Key

): Promise { return electron.ipcRenderer.invoke(key ); } function ipcInvoke2( key: Key, ...params: any[] ): Promise { return electron.ipcRenderer.invoke(key, ...params); } function ipcOn( key: Key, callback: (payload: EventPayloadMapping[Key]) => void ) { const cb = (\_: Electron.IpcRendererEvent, payload: any) => callback(payload); electron.ipcRenderer.on(key, cb); return () => electron.ipcRenderer.off(key, cb); }

function ipcSend( key: Key, payload: EventPayloadMapping[Key] ) { electron.ipcRenderer.send(key, payload); }

src\electron\main.ts import { app, BrowserWindow, ipcMain, Menu, Tray } from "electron"; import path from "path"; import { ipcMainHandle, ipcMainHandle2, ipcMainOn, isDev } from "./util.js"; import { getStaticData, pollResources } from "./resourceManager.js"; import {getAssetPath, getDBPath, getPreloadPath, getUIPath} from "./pathResolver.js"; import { createTray } from "./tray.js"; import { createMenu } from "./menu.js"; import { addInvoice, addInvoiceDetails, getAllDocumentsName, getAllInvoices, getConnectedTableDictionary, getTableDictionaryDocuments, przykladowaFunkcja, przykladowaFunkcja2, queryToDB } from "./dataBase/dbFunction.js"; export type DictionaryDocuments = { DocumentId: number; DocumentName: string; }[];

Menu.setApplicationMenu(null);

app.on("ready", () => {

const mainWindow = new BrowserWindow({ width: 1024, height: 768, resizable: true, webPreferences: { preload: getPreloadPath(),

},

// disables default system frame (dont do this if you want a proper working menu bar) // frame: false, }); if(isDev()) { mainWindow.loadURL("http://localhost:5123"); } else { mainWindow.loadFile(getUIPath()); }

pollResources(mainWindow); ipcMainHandle('getStaticData', () => { return getStaticData(); });

ipcMainHandle2('getTableDictionaryDocuments', (payload) => { return getTableDictionaryDocuments(payload); }); ipcMainHandle2('getConnectedTableDictionary', (tableName, documentId, mainTypeId, typeId, subTypeId) => { return getConnectedTableDictionary(tableName, documentId, mainTypeId, typeId, subTypeId); }); ipcMainHandle('getAllDocumentName', () => { return getAllDocumentsName(); }); ipcMainHandle2('getAllInvoices', (payload) => { return getAllInvoices(payload); }); ipcMainHandle2('addInvoice', (payload) => { return addInvoice(payload); }); ipcMainHandle2('addInvoiceDetails', (invoice, invoiceDetails) => { return addInvoiceDetails(invoice , invoiceDetails); }); ipcMainHandle2('przykladowaFunkcja', (payload, jakisNumer) => { return przykladowaFunkcja(payload , jakisNumer); }); ipcMainHandle2('przykladowaFunkcja2', (payload, jakisNumer) => { return przykladowaFunkcja2(payload , jakisNumer); }); // ipcMainHandle('addInvoice', ( invoice:unknown) => { // return addInvoice(invoice); // }); ipcMainHandle('queryToDB', () => { return queryToDB.secondMetod(); }); ipcMainOn('sendFrameAction', (payload) => { switch (payload) { case 'CLOSE': mainWindow.close(); break; case 'MAXIMIZE': mainWindow.maximize(); break; case 'MINIMIZE': mainWindow.minimize(); break; } });

createTray(mainWindow); handleCloseEvents(mainWindow); createMenu(mainWindow); });

function handleCloseEvents(mainWindow: BrowserWindow) { let willClose = false; mainWindow.on("close", (e) => { if (willClose) { return; } e.preventDefault(); mainWindow.hide(); if (app.dock) { app.dock.hide(); } });

app.on('before-quit', () => { willClose = true; });

mainWindow.on('show', () => { willClose = false; }); }

src\electron\dataBase\dbFunction.ts import Database from './dbClass.js'; import \* as sqlString from "./dbQuerySqlString.js"; import { DbTables, InvoicesTable } from './enum.js'; import { STATUS, DataBaseResponse, isSuccess } from '../sharedTypes/status.js';

// Tworzymy instancję bazy danych const db = new Database();

// Pobierz tablicę DictionaryDocuments // export async function getTableDictionaryDocuments() { // try { // const rows = await db.all(sqlString.getTableDictionaryDocumentsSqlString()); // return rows || []; // } catch (err) { // console.error('getTableDictionaryDocuments() Błąd podczas pobierania dokumentów:'); // return []; // } // };

// export type Status = (typeof STATUS)[keyof typeof STATUS];

// Pobierz tablicę DictionaryDocuments // export async function getTableDictionaryDocuments() { // try { // const rows = await db.all(sqlString.getTableDictionaryDocumentsSqlString()); // console.log("dbFunction.ts getTableDictionaryDocuments()", rows); // return { // status: STATUS.Success, // data: rows ?? [], // }; // } catch (err) { // console.error("getTableDictionaryDocuments() Błąd podczas pobierania dokumentów:", err); // return { // status: STATUS.Error, // message: "Błąd podczas pobierania dokumentów z bazy danych.", // }; // } // };

export async function getTableDictionaryDocuments(tableName: DbTables) { try { let query = ""; switch (tableName) { case DbTables.DictionaryDocuments: query = sqlString.getTableDictionaryDocumentsSqlString(tableName); break; case DbTables.DictionaryMainType: query = sqlString.getTableDictionaryDocumentsSqlString(tableName); break; case DbTables.DictionaryType: query = sqlString.getTableDictionaryDocumentsSqlString(tableName); break; case DbTables.DictionarySubtype: query = sqlString.getTableDictionaryDocumentsSqlString(tableName); break; default: throw new Error(`Nieznana tabela: ${tableName}`); }

const rows = await db.all(query); console.log("dbFunction.ts getTableDictionaryDocuments()", tableName, rows); return { status: STATUS.Success, data: rows ?? [], }; } catch (err) { console.error("getTableDictionaryDocuments() Błąd podczas pobierania dokumentów:", err); return { status: STATUS.Error, message: "Błąd podczas pobierania dokumentów z bazy danych.", }; } };

export async function getConnectedTableDictionary(tableName: DbTables, documentId?: number,mainTypeId?: number,typeId?: number,subTypeId?: number) { try { let query = ""; switch (tableName) { case DbTables.DictionaryDocuments: query = sqlString.getTableDictionaryDocumentsSqlString(tableName); break; case DbTables.DictionaryMainType: if (!documentId) { throw new Error("documentName is required for DictionaryMainType"); } query = sqlString.getConnectedTableDictionaryDocumentsDictionaryMainTypeSqlString(documentId); break; case DbTables.DictionaryType: if (!documentId || !mainTypeId) { throw new Error("documentName and mainTypeId is required for DictionaryType"); } query = sqlString.getConnectedTableDictionaryMainTypeDictionaryTypeSqlString(documentId, mainTypeId); break; case DbTables.DictionarySubtype: if (!documentId || !mainTypeId || !typeId) { throw new Error("documentName, mainTypeId and typeId is required for DictionarySubtype"); } query = sqlString.getConnectedTableDictionaryTypeDictionarySubtypeSqlString(documentId, mainTypeId, typeId); break; default: throw new Error(`Nieznana tabela: ${tableName}`); } console.log("getConnectedTableDictionary", query); const rows = await db.all(query); console.log("dbFunction.ts getConnectedTableDictionary()", tableName, rows); return { status: STATUS.Success, data: rows ?? [], }; } catch (err) { console.error("getTableDictionaryDocuments() Błąd podczas pobierania dokumentów:", err); return { status: STATUS.Error, message: `Błąd podczas pobierania dokumentów z bazy danych. ${err} coś tam`, }; } };

export async function getAllDocumentsName() { try { const rows = await db.all(sqlString.getAllDocumentsNameSqlString()); return { status: STATUS.Success, data: rows ?? [], }; } catch (err) { console.error('getAllDocumentName() Błąd podczas pobierania dokumentów:', err); return { status: STATUS.Error, message: `Błąd podczas pobierania dokumentów z bazy danych. ${err} coś tam`, }; } };

// Pobierz wszystkie faktury export async function getAllInvoices(formValuesHomePage: FormValuesHomePage) { try { const rows = await db.all(sqlString.getAllInvoicesSqlString(formValuesHomePage)); return rows || [];

} catch (err) { console.error('getAllInvoices() Błąd podczas pobierania faktur:', err); return []; } };

// Pobierz ostatni wiersz z tabeli export async function getLastRowFromTable(tableName: DbTables, tableNameId: InvoicesTable) { try { const row = await db.get(sqlString.getLastRowFromTableSqlString(tableName, tableNameId)); return row || []; } catch (err) { console.error('getLastRowFromTable() Błąd podczas pobierania faktur:', err); return []; } }; export async function addInvoice(invoice: InvoiceTable): Promise { const sql = ` INSERT INTO Invoices (InvoiceName, ReceiptDate, DeadlineDate, PaymentDate, IsDeleted) VALUES (?, ?, ?, ?, ?) `; const params = [ invoice.InvoiceName, invoice.ReceiptDate, invoice.DeadlineDate || null, invoice.PaymentDate || null, invoice.IsDeleted, ];

try { const result = await db.run(sql, params); if (!result.lastID || !result.changes) { throw new Error("Nie udało się dodać faktury."); } return { lastID: result.lastID, changes: result.changes }; } catch (err) { console.error('Błąd podczas dodawania nowej faktury:', err); throw err; } }

export async function addInvoiceDetails(invoice: InvoiceTable, invoiceDetails: InvoiceDetailsTable[]): Promise { const sql = ` INSERT INTO InvoiceDetails (InvoiceId, DocumentId, MainTypeId, TypeId, SubtypeId, Quantity, Price) VALUES (?, ?, ?, ?, ?, ?, ?) `;

try { const resultAddInvoice = await addInvoice(invoice); if (resultAddInvoice.changes && resultAddInvoice.lastID) { console.log(`Dodano nową fakturę z ID: ${resultAddInvoice.lastID}`); for (const detail of invoiceDetails) { const params = [ resultAddInvoice.lastID, detail.DocumentId, detail.MainTypeId || null, detail.TypeId || null, detail.SubtypeId || null, detail.Quantity, detail.Price, ]; const resultDetail = await db.run(sql, params); if (!resultDetail.changes) { throw new Error("Nie udało się dodać szczegółów faktury."); } } return { lastID: resultAddInvoice.lastID, changes: resultAddInvoice.changes }; } throw new Error("Nie udało się dodać faktury."); } catch (err) { console.error('Błąd podczas dodawania szczegółów faktury:', err); throw err; } } //Dodaj fakturę do tabeli Invoice // export async function addInvoice(invoice: InvoiceTable): Promise { // const sql = ` // INSERT INTO Invoices (InvoiceName, ReceiptDate, DeadlineDate, PaymentDate, IsDeleted) // VALUES (?, ?, ?, ?, ?) // `; // const params = [ // invoice.InvoiceName, // invoice.ReceiptDate, // invoice.DeadlineDate|| null, // invoice.PaymentDate || null, // invoice.IsDeleted = 0, // ];

// try { // const result = await db.run(sql, params); // return result; // } catch (err) { // console.error('Błąd podczas dodawania nowej faktury:', err); // throw err; // } // }

// const invoice = { // InvoiceName: "D01PF0031", // ReceiptDate: "2016-06-14", // DeadlineDate: "2016-07-28", // PaymentDate: "2016-07-10", // IsDeleted: 0 as const, // } // const invoiceDetails = [ // // Rekord 1 // { DocumentId: 1, MainTypeId: null, TypeId: null, SubtypeId: null, Quantity: 23, Price: 40 }, // // Rekord 2 // { DocumentId: 3, MainTypeId: 4, TypeId: 3, SubtypeId: 5, Quantity: 4, Price: 24 }, // // Rekord 3 // { DocumentId: 3, MainTypeId: null, TypeId: 2, SubtypeId: 2, Quantity: 7, Price: 42.34 } // ]; // //Dodaj szczegóły faktury do tabeli Invoice // export async function addInvoiceDetails(invoice: InvoiceTable, invoiceDetails: InvoiceDetailsTable[]) { // const sql = ` // INSERT INTO InvoiceDetails (InvoiceId, DocumentId, MainTypeId, TypeId, SubtypeId, Quantity, Price) // VALUES (?, ?, ?, ?, ?, ?, ?)`; // // const params = [ // // invoice.InvoiceName, // // invoice.ReceiptDate, // // invoice.DeadlineDate , // // invoice.PaymentDate|| null, // // invoice.IsDeleted, // // ]; // // // Dane szczegółowe faktury // // const invoiceDetails = [ // // // Rekord 1 // // { DocumentId: 1, MainTypeId: null, TypeId: null, SubtypeId: null, Quantity: 23, Price: 40 }, // // // Rekord 2 // // { DocumentId: 3, MainTypeId: 4, TypeId: 3, SubtypeId: 5, Quantity: 4, Price: 24 }, // // // Rekord 3 // // { DocumentId: 3, MainTypeId: null, TypeId: 2, SubtypeId: 2, Quantity: 7, Price: 42.34 } // // ];

// try { // const resultAddInvoice = await addInvoice(invoice); // if (resultAddInvoice.changes && resultAddInvoice.lastID) { // console.log(`Dodano nowa fakture z ID -lastId: ${resultAddInvoice.lastID}`); // console.log(`Dodano nowa fakture z ID -changes: ${resultAddInvoice.changes}`); // for (const detail of invoiceDetails) { // const invoiceDetailsTableToSave = [resultAddInvoice.lastID, detail.DocumentId, detail.MainTypeId, detail.TypeId, detail.SubtypeId, detail.Quantity, detail.Price]; // console.log(invoiceDetailsTableToSave) // const result = await db.run(sql, invoiceDetailsTableToSave); // console.log({ result }) // } // }

// // for (const detail of invoiceDetails) { // // stmt.run( // // [newInvoiceId, detail.DocumentId, detail.MainTypeId, detail.TypeId, detail.SubtypeId, detail.Quantity, detail.Price], // // (err) => { // // if (err) reject(err); // // } // // ); // // } // return ""; // } catch (err) { // console.error('Błąd podczas dodawania nowej faktury:', err); // throw err; // } // }

// addInvoiceDetails(invoice,invoiceDetails);

// export async function addInvoice(invoice: { // InvoiceName: string; // ReceiptDate: string; // DeadlineDate?: string; // PaymentDate: string; // IsDeleted: 0 | 1; // }) { // const sql = ` // INSERT INTO Invoices (InvoiceName, ReceiptDate, DeadlineDate, PaymentDate, IsDeleted) // VALUES ($InvoiceName, $ReceiptDate, $DeadlineDate, $PaymentDate, $IsDeleted) // `; // const params = { // $InvoiceName: invoice.InvoiceName, // $ReceiptDate: invoice.ReceiptDate, // $DeadlineDate: invoice.DeadlineDate || null, // $PaymentDate: invoice.PaymentDate, // $IsDeleted: invoice.IsDeleted, // };

// try { // const result = await db.run(sql, params); // console.log(`Dodano nową fakturę z ID: ${result.lastID}`); // return result.lastID; // } catch (err) { // console.error('Błąd podczas dodawania nowej faktury:', err); // throw err; // } // } export async function przykladowaFunkcja(tekst2: string, jakisNumer: number) { try { const obiekt = { jakisTekst: tekst2, jakisNumer: jakisNumer }; return obiekt; } catch (err) { console.error('fetchDocuments() Błąd podczas pobierania dokumentów:', err); const obiekt = { jakisTekst: tekst2, jakisNumer: jakisNumer }; return obiekt; } }

export async function przykladowaFunkcja2(tekst2: string, jakisNumer: number): Promise { try { const obiekt = { jakisTekst: tekst2, jakisNumer: jakisNumer }; return { status: "sukces", dane: obiekt }; } catch (err) { const errorMessage = (err as Error).message || 'Nieznany błąd'; return { status: "error", komunikat: errorMessage }; } } // Przykład funkcji asynchronicznej w module Electron do wstawiania nowej faktury

// Funkcja, która zwraca obiekt bazy danych zakładamy, że baza jest już otwarta

// async function insertInvoice(db:Database) {

// // Dane faktury // const invoiceData = { // InvoiceName: 'D01PF00297', // ReceiptDate: '2016-04-22', // DeadlineDate: '2016-05-30', // PaymentDate: '2016-05-12', // IsDeleted: 0 // };

// // Dane szczegółowe faktury // const invoiceDetails = [ // // Rekord 1 // { DocumentId: 1, MainTypeId: null, TypeId: null, SubtypeId: null, Quantity: 23, Price: 40 }, // // Rekord 2 // { DocumentId: 3, MainTypeId: 4, TypeId: 3, SubtypeId: 5, Quantity: 4, Price: 24 }, // // Rekord 3 // { DocumentId: 3, MainTypeId: null, TypeId: 2, SubtypeId: 2, Quantity: 7, Price: 42.34 } // ];

// // Zmienna do przechwytywania nowego InvoiceId // let newInvoiceId;

// try { // await new Promise((resolve, reject) => { // db.serialize(() => { // db.run("BEGIN TRANSACTION;");

// // Wstawienie faktury // db.run( // `INSERT INTO Invoices (InvoiceName, ReceiptDate, DeadlineDate, PaymentDate, IsDeleted) // VALUES (?, ?, ?, ?, ?);`, // [invoiceData.InvoiceName, invoiceData.ReceiptDate, invoiceData.DeadlineDate, invoiceData.PaymentDate, invoiceData.IsDeleted], // function (err) { // if (err) return reject(err); // newInvoiceId = this.lastID; // Pobieramy identyfikator nowo wstawionej faktury

// // Wstawienie szczegółów faktury // const stmt = db.prepare( // `INSERT INTO InvoiceDetails (InvoiceId, DocumentId, MainTypeId, TypeId, SubtypeId, Quantity, Price) // VALUES (?, ?, ?, ?, ?, ?, ?);` // );

// for (const detail of invoiceDetails) { // stmt.run( // [newInvoiceId, detail.DocumentId, detail.MainTypeId, detail.TypeId, detail.SubtypeId, detail.Quantity, detail.Price], // (err) => { // if (err) reject(err); // } // ); // }

// stmt.finalize((err) => { // if (err) return reject(err); // db.run("COMMIT TRANSACTION;", (err) => { // if (err) return reject(err); // resolve(); // }); // }); // } // ); // }); // }); // console.log("Faktura została pomyślnie dodana, InvoiceId:", newInvoiceId); // } catch (error) { // console.error("Błąd podczas dodawania faktury:", error); // db.run("ROLLBACK TRANSACTION;"); // } finally { // db.close(); // } // }

export const queryToDB = { firstMetod: async function fetchDocuments() { try { const rows = await db.all('SELECT \* FROM DictionaryDocuments'); console.log("fetchDocuments()", rows); return rows || [];

} catch (err: unknown) { console.error('fetchDocuments() Błąd podczas pobierania dokumentów:', err); return []; }

}, secondMetod: async function fetchDocuments() { try { const rows = await db.all('SELECT \* FROM DictionaryMainType'); console.log("fetchDocuments()", rows); return rows || [];

} catch (err) { console.error('fetchDocuments() Błąd podczas pobierania dokumentów:'); return []; }

} };

// import { app } from 'electron'; // app.on('before-quit', async () => { // try { // await db.close(); // console.log('Połączenie z bazą danych zostało zamknięte.'); // } catch (err) { // console.error('Błąd przy zamykaniu bazy danych:', err); // } // });

src\ui\components\FormAddInvoice\FormAddInvoice.tsx import { useEffect, useState } from "react"; import { nanoid } from "nanoid"; import { DateTimePicker } from "../DateTimePicker/DateTimePicker"; import { FormAddInvoiceDocuments } from "../FormAddInvoiceDocuments/FormAddInvoiceDocuments"; import { TextInput } from "../TextInput/TextInput"; import scss from "./FormAddInvoice.module.scss"; import { Tooltip } from "react-tooltip"; import { FaInfoCircle } from "react-icons/fa"; import { addInvoiceDetails } from "../../../electron/dataBase/dbFunction"; import { ButtonCancel } from "../ButtonCancel/ButtonCancel"; interface FormAddInvoiceProps { addInvoiceData: InvoiceSave; setAddInvoiceData: React.Dispatch<React.SetStateAction>; }

export const FormAddInvoice: React.FC = ({ addInvoiceData, setAddInvoiceData, }) => { const [inputInvoiceName, setInputInvoiceName] = useState(""); const [inputInvoiceNameError, setInputInvoiceNameError] = useState(""); const [dateTimePickerReceiptDate, setDateTimePickerReceiptDate] = useState(null); const [dateTimePickerDeadlineDate, setDateTimePickerDeadlineDate] = useState(null); const [dateTimePickerPaymentDate, setDateTimePickerPaymentDate] = useState(null); const [documentComponents, setDocumentComponents] = useState([ nanoid(), ]); const [isSaveButtonEnabled, setIsSaveButtonEnabled] = useState(false);

// Formatowanie daty do YYYY-MM-DD const formatDate = (date: Date | null): string | null => { if (!date) return null; return date.toISOString().split("T")[0]; // YYYY-MM-DD };

// Aktualizacja dat w addInvoiceData useEffect(() => { setAddInvoiceData((prev) => ({ ...prev, invoice: { ...prev.invoice, ReceiptDate: formatDate(dateTimePickerReceiptDate) || "", DeadlineDate: formatDate(dateTimePickerDeadlineDate), PaymentDate: formatDate(dateTimePickerPaymentDate) || null, }, })); }, [ dateTimePickerReceiptDate, dateTimePickerDeadlineDate, dateTimePickerPaymentDate, setAddInvoiceData, ]); // Walidacja pól i unikalności dokumentów const validateForm = (): boolean => { // Sprawdzenie invoiceName const isInvoiceNameValid = inputInvoiceName.trim() !== "";

// Sprawdzenie ReceiptDate const isReceiptDateValid = !!dateTimePickerReceiptDate;

// Sprawdzenie dokumentów const areDetailsValid = addInvoiceData.details.every((detail) => { return ( detail.DocumentId !== 0 && detail.Quantity > 0 && detail.Price >= 0 && (!detail.MainTypeId || detail.MainTypeId > 0) && (!detail.TypeId || detail.TypeId > 0) && (!detail.SubtypeId || detail.SubtypeId > 0) ); });

// Sprawdzenie unikalności dokumentów const documentKeys = addInvoiceData.details.map( (detail) => `${detail.DocumentId}-${detail.MainTypeId ?? "null"}-${ detail.TypeId ?? "null" }-${detail.SubtypeId ?? "null"}` ); const uniqueDocuments = new Set(documentKeys).size === documentKeys.length;

return ( isInvoiceNameValid && isReceiptDateValid && areDetailsValid && uniqueDocuments ); }; // Aktualizacja stanu przycisku Zapisz useEffect(() => { setIsSaveButtonEnabled(validateForm()); }, [inputInvoiceName, dateTimePickerReceiptDate, addInvoiceData.details]);

const handleSingleInputChange = ( event: React.ChangeEvent ) => { const currentValue = event.target.value; const currentName = event.target.name; let errorTextInput = "";

if (currentName === "invoiceName") { setInputInvoiceName(currentValue); setAddInvoiceData((prev) => ({ ...prev, invoice: { ...prev.invoice, InvoiceName: currentValue }, })); if (!currentValue) { errorTextInput = "Musisz wypełnić te pole"; } setInputInvoiceNameError(errorTextInput); } };

const handleAddDocument = () => { const newId = nanoid(); setDocumentComponents((prev) => [...prev, newId]); setAddInvoiceData((prev) => ({ ...prev, details: [ ...prev.details, { InvoiceId: undefined, DocumentId: 0, MainTypeId: null, TypeId: null, SubtypeId: null, Quantity: 0, Price: 0, }, ], })); }; const handleRemoveDocument = (id: string) => { if (documentComponents.length <= 1) { return; // Blokuj usuwanie, jeśli jest tylko jeden komponent } const indexToRemove = documentComponents.indexOf(id); if (indexToRemove === -1) return; // Zabezpieczenie przed błędnym ID setDocumentComponents((prev) => prev.filter((componentId) => componentId !== id) ); setAddInvoiceData((prev) => ({ ...prev, details: prev.details.filter((\_, i) => i !== indexToRemove), })); }; // const handleRemoveDocument = (id: string) => { // if (documentComponents.length <= 1) { // return; // Blokuj usuwanie, jeśli jest tylko jeden komponent // } // setDocumentComponents((prev) => // prev.filter((componentId) => componentId !== id) // ); // setAddInvoiceData((prev) => ({ // ...prev, // details: prev.details.filter((\_, i) => documentComponents[i] !== id), // })); // }; const handleSaveInvoice = async () => { if (!isSaveButtonEnabled) return;

try { const invoice: InvoiceTable = { InvoiceName: addInvoiceData.invoice.InvoiceName, ReceiptDate: addInvoiceData.invoice.ReceiptDate, DeadlineDate: addInvoiceData.invoice.DeadlineDate, PaymentDate: addInvoiceData.invoice.PaymentDate, IsDeleted: 0, };

const invoiceDetails: InvoiceDetailsTable[] = addInvoiceData.details.map( (detail) => ({ DocumentId: detail.DocumentId, MainTypeId: detail.MainTypeId, TypeId: detail.TypeId, SubtypeId: detail.SubtypeId, Quantity: detail.Quantity, Price: detail.Price, }) );

const result = await addInvoiceDetails(invoice, invoiceDetails); if (result.changes && result.lastID) { alert(`Faktura zapisana pomyślnie! ID: ${result.lastID}`); // Reset formularza po zapisie setInputInvoiceName(""); setDateTimePickerReceiptDate(null); setDateTimePickerDeadlineDate(null); setDateTimePickerPaymentDate(null); setDocumentComponents([nanoid()]); setAddInvoiceData({ invoice: { InvoiceId: undefined, InvoiceName: "", ReceiptDate: "", DeadlineDate: null, PaymentDate: null, IsDeleted: 0, }, details: [ { InvoiceId: undefined, DocumentId: 0, MainTypeId: null, TypeId: null, SubtypeId: null, Quantity: 0, Price: 0, }, ], }); } else { throw new Error("Nie udało się dodać faktury."); } } catch (error) { console.error("Błąd podczas zapisywania faktury:", error); alert( `Wystąpił błąd podczas zapisywania faktury: ${(error as Error).message}` ); } }; return (

Początek formularza

**Dodaj nową fakturę:**

Data wpływu:

Termin płatności:

Data płatności:

**Dodaj dokumenty:**

{documentComponents.map((id, index) => ( handleRemoveDocument(id)} isLast={index === documentComponents.length - 1} isOnly={documentComponents.length === 1} index={index} /> ))}

Faktura: 1234,00 zł

Dół formularza

); };

function tooltipInfoFormAddInvoice() { const text = `Formularz wyboru daty. Umożliwia wybór początkowej i końcowej daty wpływu faktury oraz opcjonalne wyświetlenie usuniętych elementów. Wybierz daty, kliknij przycisk "Pokaż", aby zastosować zmiany. Uwaga! Data początkowa nie może być późniejsza niż data końcowa. Przycisk "Dodaj fakturę" służy do otwarcia okna, w którym można dodać fakturę.`; return text.replace(/\n/g, "  
"); }

src\ui\components\FormAddInvoiceDocuments\FormAddInvoiceDocuments.tsx import { use, useEffect, useMemo, useState } from "react"; import { useTableDictionaryDocuments } from "../../hooks/useTableDictionaryDocuments"; import { useConnectedTableDictionary } from "../../hooks/useConnectedTableDictionary"; import { MdOutlinePostAdd } from "react-icons/md"; import scss from "./FormAddInvoiceDocuments.module.scss"; import Select, { SingleValue } from "react-select"; import { DbTables } from "../../../electron/dataBase/enum"; import { TextInput } from "../TextInput/TextInput"; import { customStylesComboBox, ComboBoxOption } from "../ComboBox/ComboBox"; import { SingleInput } from "../SingleInput/SingleInput"; import { ButtonCancel } from "../ButtonCancel/ButtonCancel"; import { useMainDataContext } from "../Context/useOptionsImage"; import { calculateTotalAmount } from "../GlobalFunctions/GlobalFunctions";

interface FormAddInvoiceDocumentsProps { addInvoiceData: InvoiceSave; setAddInvoiceData: React.Dispatch<React.SetStateAction>; onAddDocument: () => void; onRemoveDocument: () => void; isLast: boolean; isOnly: boolean; index: number; } export const FormAddInvoiceDocuments: React.FC< FormAddInvoiceDocumentsProps > = ({ addInvoiceData, setAddInvoiceData, onAddDocument, onRemoveDocument, isLast, isOnly, index, }) => { //useState combobox const [selectedDocument, setSelectedDocument] = useState(null); const [selectedMainType, setSelectedMainType] = useState(null); const [selectedType, setSelectedType] = useState(null); const [selectedSubtype, setSelectedSubtype] = useState( null ); const [isMainTypeExistsBool, setIsMainTypeExistsBool] = useState(false); const [isTypeExistsBool, setIsTypeExistsBool] = useState(false); const [isSubtypeExistsBool, setIsSubtypeExistsBool] = useState(false);

//useState textbox const [inputInvoiceQuantity, setInputInvoiceQuantity] = useState(""); const [inputInvoiceQuantityError, setInputInvoiceQuantityError] = useState(""); const [inputInvoicePrice, setInputInvoicePrice] = useState(""); const [inputInvoicePriceError, setInputInvoicePriceError] = useState(""); const [isPriceManuallyEdited, setIsPriceManuallyEdited] = useState(false); // Nowy stan //All documents name const { allDocumentsData } = useMainDataContext(); const { data: dataAllDocumentsName, loading: loadingAllDocumentsName, error: errorAllDocumentsName, } = allDocumentsData;

const handleSingleInputChange = ( event: React.ChangeEvent ) => { const currentValue = event.target.value; const currentName = event.target.name; let errorTextInput = "";

if (currentName === "quantity") { const isValidInteger = /^\d\*$/.test(currentValue); if (isValidInteger) { // Usuń wiodące zera i sparsuj do liczby const parsedValue = currentValue ? parseInt(currentValue, 10) : NaN; const cleanedValue = isNaN(parsedValue) ? "" : parsedValue.toString();

setInputInvoiceQuantity(cleanedValue);

if (!cleanedValue) { errorTextInput = "Musisz wypełnić to pole"; } else if (parsedValue <= 0) { errorTextInput = "Wprowadź liczbę większą od 0"; } } else { errorTextInput = "Dozwolone są tylko liczby całkowite"; } setInputInvoiceQuantityError(errorTextInput); } if (currentName === "price") { const isValidPrice = /^\d\*\.?\d\*$/.test(currentValue); // Dopuszcza liczby zmiennoprzecinkowe setInputInvoicePrice(currentValue); setIsPriceManuallyEdited(true); // Ustaw flagę, że cena została ręcznie edytowana if (!currentValue) { errorTextInput = "Musisz wypełnić to pole"; } else if (currentValue.includes(",")) { errorTextInput = "Zamiast przecinka użyj kropki"; } else if (!isValidPrice) { errorTextInput = "Wprowadź poprawną liczbę"; } setInputInvoicePriceError(errorTextInput); } }; const handleKeyDownQuantityInput = ( event: React.KeyboardEvent ) => { if ( event.key.match(/[^0-9]/) && !["Backspace", "Delete", "ArrowLeft", "ArrowRight", "Tab"].includes( event.key ) ) { event.preventDefault(); } }; const handleKeyDownPriceInput = ( event: React.KeyboardEvent ) => { if ( event.key === "," || (event.key.match(/[^0-9.]/) && !["Backspace", "Delete", "ArrowLeft", "ArrowRight", "Tab"].includes( event.key )) ) { event.preventDefault(); } }; //Dane tabel pobrane z hooka //dictionaryDocumentTable const { data: dictionaryDocumentTable, loading: loadingDictionaryDocumentTable, error: errorDictionaryDocumentTable, } = useConnectedTableDictionary( DbTables.DictionaryDocuments ); //dictionaryMainTypeTable const { data: dictionaryMainTypeTable, loading: loadingDictionaryMainTypeTable, error: errorDictionaryMainTypeTable, } = useConnectedTableDictionary( DbTables.DictionaryMainType, selectedDocument?.value ); //dictionaryTypeTable const { data: dictionaryTypeTable, loading: loadingDictionaryTypeTable, error: errorDictionaryTypeTable, } = useConnectedTableDictionary( DbTables.DictionaryType, selectedDocument?.value, selectedMainType?.value ); //dictionarySubtypeTable const { data: dictionarySubtypeTable, loading: loadingDictionarySubtypeTable, error: errorDictionarySubtypeTable, } = useConnectedTableDictionary( DbTables.DictionarySubtype, selectedDocument?.value, selectedMainType?.value, selectedType?.value );

//dane do combobox //dictionaryDocumentTable const optionsDictionaryDocumentTable = useMemo(() => { if (!dictionaryDocumentTable) { return []; // Zwróć pustą tablicę, jeśli brak danych } return dictionaryDocumentTable.map((doc) => ({ value: doc.DocumentId, label: doc.DocumentName, })); }, [dictionaryDocumentTable]); //dictionaryMainTypeTable const optionsDictionaryMainTypeTable = useMemo(() => { if (!dictionaryMainTypeTable) { return []; // Zwróć pustą tablicę, jeśli brak danych } return dictionaryMainTypeTable.map((doc) => ({ value: doc.MainTypeId, label: doc.MainTypeName, })); }, [dictionaryMainTypeTable]); //dictionaryTypeTable const optionsDictionaryTypeTable = useMemo(() => { if (!dictionaryTypeTable) { return []; // Zwróć pustą tablicę, jeśli brak danych } return dictionaryTypeTable.map((doc) => ({ value: doc.TypeId, label: doc.TypeName, })); }, [dictionaryTypeTable]); //dictionarySubtypeTable const optionsDictionarySubtypeTable = useMemo(() => { if (!dictionarySubtypeTable) { return []; // Zwróć pustą tablicę, jeśli brak danych } return dictionarySubtypeTable.map((doc) => ({ value: doc.SubtypeId, label: doc.SubtypeName, })); }, [dictionarySubtypeTable]);

//set single item in combobox const getSingleDefaultOption = ( options: T[] ): T | undefined => { return options.length === 1 ? options[0] : undefined; }; useEffect(() => { const defaultMainType = getSingleDefaultOption( optionsDictionaryMainTypeTable ); if (defaultMainType && !selectedMainType) { setSelectedMainType(defaultMainType); } }, [optionsDictionaryMainTypeTable, selectedMainType]);

useEffect(() => { const defaultType = getSingleDefaultOption(optionsDictionaryTypeTable); if (defaultType && !selectedType) { setSelectedType(defaultType); } }, [optionsDictionaryTypeTable, selectedType]);

useEffect(() => { const defaultSubtype = getSingleDefaultOption( optionsDictionarySubtypeTable ); if (defaultSubtype && !selectedSubtype) { setSelectedSubtype(defaultSubtype); } }, [optionsDictionarySubtypeTable, selectedSubtype]);

//removing elements from the combobox when changing useEffect(() => { setSelectedMainType(null); setSelectedType(null); setSelectedSubtype(null); setIsPriceManuallyEdited(false); // Resetuj flagę przy zmianie dokumentu }, [selectedDocument]); useEffect(() => { setSelectedType(null); setSelectedSubtype(null); setIsPriceManuallyEdited(false); // Resetuj flagę przy zmianie typu głównego }, [selectedMainType]); useEffect(() => { setSelectedSubtype(null); setIsPriceManuallyEdited(false); // Resetuj flagę przy zmianie typu }, [selectedType]);

//Setting the price and checking the existence of types useEffect(() => { const isMainTypeExists = checkComboBoxExistence( [selectedDocument], optionsDictionaryMainTypeTable ); setIsMainTypeExistsBool(isMainTypeExists);

const isTypeExists = checkComboBoxExistence( [selectedDocument, selectedMainType], optionsDictionaryTypeTable ); setIsTypeExistsBool(isTypeExists);

const isSubtypeExists = checkComboBoxExistence( [selectedDocument, selectedMainType, selectedType], optionsDictionarySubtypeTable ); setIsSubtypeExistsBool(isSubtypeExists);

if (!isPriceManuallyEdited) { const price = getPrice( dataAllDocumentsName, selectedDocument, selectedMainType, isMainTypeExists, selectedType, isTypeExists, selectedSubtype, isSubtypeExists ); setInputInvoicePrice(price); }

// Aktualizacja addInvoiceData.details setAddInvoiceData((prev) => { const newDetails = [...prev.details]; newDetails[index] = { InvoiceId: undefined, DocumentId: selectedDocument?.value ?? 0, MainTypeId: selectedMainType?.value ?? null, TypeId: selectedType?.value ?? null, SubtypeId: selectedSubtype?.value ?? null, Quantity: inputInvoiceQuantity ? parseInt(inputInvoiceQuantity) : 0, Price: inputInvoicePrice ? parseFloat(inputInvoicePrice) : 0, }; return { ...prev, details: newDetails }; }); }, [ dataAllDocumentsName, selectedDocument, selectedMainType, optionsDictionaryMainTypeTable, selectedType, optionsDictionaryTypeTable, selectedSubtype, optionsDictionarySubtypeTable, inputInvoiceQuantity, inputInvoicePrice, isPriceManuallyEdited, index, setAddInvoiceData, ]);

const areFieldsFilled = () => { const isDocumentSelected = !!selectedDocument; const isQuantityValid = inputInvoiceQuantity && parseInt(inputInvoiceQuantity) > 0; const isPriceValid = inputInvoicePrice && /^\d\*\.?\d\*$/.test(inputInvoicePrice);

// Sprawdź, czy wymagane pola zależne są wypełnione const isMainTypeFilled = !isMainTypeExistsBool || !!selectedMainType; const isTypeFilled = !isTypeExistsBool || !!selectedType; const isSubtypeFilled = !isSubtypeExistsBool || !!selectedSubtype;

return ( isDocumentSelected && isQuantityValid && isPriceValid && isMainTypeFilled && isTypeFilled && isSubtypeFilled ); }; return (

<Select //Dokument Combobox value={selectedDocument} // <-- zamiast tylko defaultValue defaultValue={getSingleDefaultOption(optionsDictionaryDocumentTable)} onChange={(option) => setSelectedDocument(option as ComboBoxOption)} options={optionsDictionaryDocumentTable} // Użyj danych z hooka isSearchable={false} placeholder="Wybierz..." styles={customStylesComboBox} menuPortalTarget={document.body} // Portal, który zapewnia renderowanie listy na poziomie document.body menuPosition="fixed" // Zapewnia, że pozycjonowanie menu jest "fixed" menuShouldBlockScroll={true} // Opcjonalnie: blokuje scroll podczas otwartego menu className={scss["select-document-container"]} classNamePrefix={scss["select-document"]} /> {isMainTypeExistsBool && ( <Select //MainType Combobox value={selectedMainType} defaultValue={getSingleDefaultOption( optionsDictionaryMainTypeTable )} onChange={(option) => setSelectedMainType(option as ComboBoxOption)} options={optionsDictionaryMainTypeTable} // Użyj danych z hooka isSearchable={false} placeholder="Wybierz..." styles={customStylesComboBox} menuPortalTarget={document.body} // Portal, który zapewnia renderowanie listy na poziomie document.body menuPosition="fixed" // Zapewnia, że pozycjonowanie menu jest "fixed" menuShouldBlockScroll={true} // Opcjonalnie: blokuje scroll podczas otwartego menu className={scss["select-maintype-container"]} /> )} {isTypeExistsBool && ( <Select //Type Combobox value={selectedType} defaultValue={getSingleDefaultOption(optionsDictionaryTypeTable)} onChange={(option) => setSelectedType(option as ComboBoxOption)} options={optionsDictionaryTypeTable} // Użyj danych z hooka isSearchable={false} placeholder="Wybierz..." styles={customStylesComboBox} menuPortalTarget={document.body} // Portal, który zapewnia renderowanie listy na poziomie document.body menuPosition="fixed" // Zapewnia, że pozycjonowanie menu jest "fixed" menuShouldBlockScroll={true} // Opcjonalnie: blokuje scroll podczas otwartego menu className={scss["select-type-container"]} /> )} {isSubtypeExistsBool && ( <Select //Subtype Combobox value={selectedSubtype} defaultValue={getSingleDefaultOption(optionsDictionarySubtypeTable)} onChange={(option) => setSelectedSubtype(option as ComboBoxOption)} options={optionsDictionarySubtypeTable} // Użyj danych z hooka isSearchable={false} placeholder="Wybierz..." styles={customStylesComboBox} menuPortalTarget={document.body} // Portal, który zapewnia renderowanie listy na poziomie document.body menuPosition="fixed" // Zapewnia, że pozycjonowanie menu jest "fixed" menuShouldBlockScroll={true} // Opcjonalnie: blokuje scroll podczas otwartego menu className={scss["select-subtype-container"]} /> )}

Razem:{" "} {calculateTotalAmount( [inputInvoiceQuantity], [inputInvoicePrice] )}

{isLast && areFieldsFilled() && ( } classNameButtonContainer={scss["button-add-document"]} /> )}

); };

function getPrice( dataAllDocumentsName: AllDocumentsName[] | null, selectedDocumentComboBox: ComboBoxOption | null, selectedMainTypeComboBox: ComboBoxOption | null, isMainTypeExists: boolean, selectedTypeComboBox: ComboBoxOption | null, isTypeExists: boolean, selectedSubtypeComboBox: ComboBoxOption | null, isSubtypeExists: boolean ): string { if (isMainTypeExists) { if (isTypeExists) { if (isSubtypeExists) { const selectedDocuments = dataAllDocumentsName?.filter( (doc) => doc.DocumentId === selectedDocumentComboBox?.value && doc.MainTypeId === selectedMainTypeComboBox?.value && doc.TypeId === selectedTypeComboBox?.value && doc.SubtypeId === selectedSubtypeComboBox?.value ) || []; if (selectedDocuments.length == 1 && selectedDocuments[0].Price) { const price = selectedDocuments[0].Price.toString(); return price; // Zwraca cenę pierwszego pasującego dokumentu } } else { const selectedDocuments = dataAllDocumentsName?.filter( (doc) => doc.DocumentId === selectedDocumentComboBox?.value && doc.MainTypeId === selectedMainTypeComboBox?.value && doc.TypeId === selectedTypeComboBox?.value ) || []; if (selectedDocuments.length == 1 && selectedDocuments[0].Price) { const price = selectedDocuments[0].Price.toString(); return price; } } } else { const selectedDocuments = dataAllDocumentsName?.filter( (doc) => doc.DocumentId === selectedDocumentComboBox?.value && doc.MainTypeId === selectedMainTypeComboBox?.value ) || []; if (selectedDocuments.length == 1 && selectedDocuments[0].Price) { const price = selectedDocuments[0].Price.toString(); return price; } } } else { const selectedDocuments = dataAllDocumentsName?.filter( (doc) => doc.DocumentId === selectedDocumentComboBox?.value ) || []; if (selectedDocuments.length == 1 && selectedDocuments[0].Price) { const price = selectedDocuments[0].Price.toString(); return price; } } return "0"; }

function checkComboBoxExistence( comboBoxes: (ComboBoxOption | null)[], optionsTable: ComboBoxOption[] | null ): boolean { return ( comboBoxes.every((comboBox) => !!comboBox) && !!optionsTable && optionsTable.length > 0 ); }

po uruchomieniu programu pojawia mi się taki błąd: Module "fs" has been externalized for browser compatibility. Cannot access "fs.existsSync" in client code. See https://vite.dev/guide/troubleshooting.html#module-externalized-for-browser-compatibility for more details. get @ chunk-DF2DBHCE.js?v=d5330809:24 chunk-DF2DBHCE.js?v=d5330809:11 Module "path" has been externalized for browser compatibility. Cannot access "path.existsSync" in client code. See https://vite.dev/guide/troubleshooting.html#module-externalized-for-browser-compatibility for more details. get @ chunk-DF2DBHCE.js?v=d5330809:11 sqlite3.js?v=d5330809:57 Uncaught ReferenceError: process is not defined at node\_modules/bindings/bindings.js (sqlite3.js?v=d5330809:57:14) at \_\_require2 (chunk-KEXKKQVW.js?v=d5330809:15:50) at node\_modules/sqlite3/lib/sqlite3-binding.js (sqlite3.js?v=d5330809:185:22) at \_\_require2 (chunk-KEXKKQVW.js?v=d5330809:15:50) at node\_modules/sqlite3/lib/sqlite3.js (sqlite3.js?v=d5330809:257:19) at \_\_require2 (chunk-KEXKKQVW.js?v=d5330809:15:50) at sqlite3.js?v=d5330809:424:16 react-dom\_client.js?v=d5330809:5589 An error occurred in one of your React components.

Consider adding an error boundary to your tree to customize error handling behavior. Visit https://react.dev/link/error-boundaries to learn more about error boundaries.