# Введение

Кроссплатформенные десктоп приложения – это приложения которые используют веб технологии для реализации настольных приложений. Работают с html и css.

Такие приложения могут взаимодействовать с операционной системой также как C#???

## Среды разработки кросс платформенных приложений:

Electron (JavaScript/TypeScript) — для приложений с веб-интерфейсом (например, VS Code, Slack).

Qt (C++/Python) — мощный фреймворк для создания нативных приложений.

Flutter (Dart) — набирает популярность для создания кроссплатформенных настольных приложений.

Avalonia (C#) — для разработки настольных приложений с использованием .NET.

Tauri (Rust) — легкая альтернатива Electron с меньшим потреблением ресурсов.

# Пакеты для разработки

Для корректной работы фронтэнда нужно убедится что базовые вещи работали

JavaScript

Node js

npm

## npm

node -v

npm –v

npm install -g npm@latest

### npm install -g npm@latest не работает

> PS D:\#work\zadaniya\5\_administrate\_BD> npm -v

10.9.2

> PS D:\#work\zadaniya\5\_administrate\_BD> npm install -g npm@latest

npm error code SELF\_SIGNED\_CERT\_IN\_CHAIN

npm error errno SELF\_SIGNED\_CERT\_IN\_CHAIN

npm error request to https://registry.npmjs.org/npm failed, reason: self-signed certificate in certificate chain

npm error A complete log of this run can be found in: C:\Users\Rud\_MS\AppData\Local\npm-cache\\_logs\2025-03-07T07\_45\_27\_507Z-debug-0.log

> PS D:\#work\zadaniya\5\_administrate\_BD> npm cache clean

npm error As of npm@5, the npm cache self-heals from corruption issues

npm error by treating integrity mismatches as cache misses. As a result,

npm error data extracted from the cache is guaranteed to be valid. If you

npm error want to make sure everything is consistent, use `npm cache verify`

npm error instead. Deleting the cache can only make npm go slower, and is

npm error not likely to correct any problems you may be encountering!

npm error

npm error On the other hand, if you're debugging an issue with the installer,

npm error or race conditions that depend on the timing of writing to an empty

npm error cache, you can use `npm install --cache /tmp/empty-cache` to use a

npm error temporary cache instead of nuking the actual one.

npm error

npm error If you're sure you want to delete the entire cache, rerun this command

npm error with --force.

npm error A complete log of this run can be found in: C:\Users\Rud\_MS\AppData\Local\npm-cache\\_logs\2025-03-07T07\_47\_58\_811Z-debug-0.log

Вот что помогло:

> PS D:\#work\zadaniya\5\_administrate\_BD> npm config set strict-ssl false

> PS D:\#work\zadaniya\5\_administrate\_BD> npm install -g npm@latest

added 1 package in 5s

24 packages are looking for funding

run `npm fund` for details

> PS D:\#work\zadaniya\5\_administrate\_BD> npm -v

11.2.0

# Tauri 2.0

<https://v2.tauri.app/>

Фреймворк для создания кросплатформенных приложений.

Поддерживает бэкенд и фронтэнд. Поэтому можно писать код на разных языках в независимых ячейках.

## Документация tauri 2.0

<https://v2.tauri.app/start/>

## Начало

### Установка нужного

<https://v2.tauri.app/start/prerequisites/>

### Стек для приложения

Когда происходит создание шаблона приложения в выбранной папке  
  
то мы можем выбрать

#### 1) Сброщик фронтенда

– платформа которая собирает сервер и компилирует приложение

<https://vite.dev/>

#### 2) Менеджер пакетов

– система которая контролирует обновление пакетов и зависимостей

#### 3) Фреймворк разработки приложения

– просто фреймворк

? Choose your UI template ›

Vanilla

Vue

❯ Svelte (https://svelte.dev/)

React

Solid

Angular

Preact

Сравнение

<https://www.youtube.com/watch?v=MnpuK0MK4yo>

#### Мой выбор

Я выбрал

Vite – самый популярный вроде, да и в целом пофик какой сборщик

React – популярный фреймворк, позволяющий разрабатывать приложения на TS

TypeScript – надежный язык поверх JS

### Создание, компиляция и запуск шаблона

Tauri позволяет один раз скомпилировать приложение через терминал и после этого запустить его на ОС.

После чего можно будет параллельно с работающим приложением открыть его проект в Visual Studio и редактировать его. Изменения будут в реальном времени появляться в приложении

#### 1) Подключение из командной строки к папке:

d:

cd D:\#work\zadaniya\5\_administrate\_BD\crossplatform\_tauri\_app

#### 2) Создание шаблона проекта:

##### Стандартный шаблон с сайта таури

irm https://create.tauri.app/ps | iex

6NEWmoduleViewer

**Важно:**

1. По условию Сборщик – Vite
2. Идентификатор приложения должен соответствовать правилам

Идентификаторы (ID) для приложений, особенно для Android, должны соответствовать определенным правилам. Эти правила важны, чтобы избежать конфликтов между приложениями и обеспечить уникальность каждого приложения в экосистеме (например, в Google Play Store).

Правила для идентификаторов

1. Формат идентификатора

Идентификатор должен быть в формате доменного имени (обратного DNS). Например:

com.example.myapp

org.mycompany.myapp

io.github.username.project

Код

irm https://create.tauri.app/ps | iex

app

123

TypeScript / JavaScript

npm

React

TypeScript

cd app

npm install

##### Шаблон Vite

npm create vite@latest

#### 3) Запуск

npm run tauri dev

После этого приложение скомпилируется и запустится.

### Приложение

Весь код приложения помещен в src

#### Повторный запуск для разработки

Точно также нужно зайти в ту же директорию и

npm run tauri dev

#### Фронтенд (src)

##### Main.tsx

Цикл приложения

##### App.tsx

Логика окна

##### App.css

Внешний вид окна

Визуальные настройки окна

#### Бэкенд (src-turi)

Бэкенд пишется на Rust в Tauri

##### build.rs

build.rs — это скрипт сборки (build script), который выполняется до компиляции основного кода бэкенда. Он используется для настройки или генерации кода, который будет использоваться во время компиляции.

##### src/main.rs

src/main.rs — это точка входа в бэкендную часть приложения Tauri. Здесь находится основной код, который выполняется при запуске приложения.

#### Билд релиза

npm run tauri build

для успешного билда и создания установщика нужно чтобы программа работала в режиме dev + установить wix toolkit именно для создания установщика

d:

Get-ExecutionPolicy (должно быть НЕ restricted)

Set-ExecutionPolicy AllSigned

A

Установка chkolatey:

Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))

choco upgrade chocolatey

choco install wixtoolset

## Работа

### Back <–> Front (коммуникация)

#### Готовность к коммуникации

Чтобы сообщить бекенду, что фронт готов к получению запросов нужно добавить событие на фронте, которое бы отправляло бекенду информацию о том, что фронт загрузился и готов.

import { listen, emit } from '@tauri-apps/api/event';

import { useState, useEffect } from "react";

function App() {

useEffect(() => {

*// Уведомляем бэкенд о готовности фронтенда*

emit("frontend\_ready", {}).catch(err => console.error("Failed to emit frontend\_ready", err));

}, []);

……

…….

На бекенде слушаем это сообщение и ждём до тех пор

tauri::Builder::default()

.setup(|app| {

let handle = app.app\_handle(); // создаём указатель на объект нашего приложения

let handle\_clone = handle.clone();

handle.listen("frontend\_ready", move |\_| { // ждём пока фронт полностью отрендерится

let query = r#"

SELECT one."\*SAP\_nomer", one.ENTRY, one."Description"

FROM E3\_ADMIN."ComponentData" one

JOIN E3\_ADMIN."ComponentAttribute" two ON two.ID = one.ID

WHERE two."AttributeValue" = 'Разрешено использование'

"#;

let json = get\_json\_db\_response(query);

handle\_clone.emit("test", json).unwrap();

//println!("{}", &json);

start\_sending\_data2(handle\_clone.clone());

});

//start\_sending\_data(handle.clone()); // передаём новый указатель на объект приложения

//start\_sending\_data2(handle.clone());

Ok(())

})

.run(tauri::generate\_context!())

.expect("error while running tauri application");

}

#### Front calling <- back function

##### Бэкенд:

#[tauri::command] - позволяет вызвать команду из фронта

fn symcount() -> int {

Ok() => 5

}

fn main() {

tauri::Builder::default()

.invoke\_handler(tauri::generate\_handler![symcount])

.setup(|app| {

Ok(())

})

.run(tauri::generate\_context!())

.expect("Ошибка при запуске приложения Tauri");

}

##### Фронт:

import { invoke } from "@tauri-apps/api/core";

function App() {

const result = await invoke(" symcount ");

}

return (

<main>

<button onClick={ symcount } > query from back </button>

result

</main>);

}

##### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### Передача аргументов из фронта в бэк

Фронт

const query1 = "SELECT COUNT(\*) FROM E3\_ADMIN.\"ComponentData\" ";

const response: string = await invoke("dsaemdbquery", { query: query1 });

бэк:  
fn dsaemdbquery(query: &str) -> String {

// реализация //

}

#### Back sending -> front listening

##### Бэк:

use tauri::{AppHandle, Manager, Emitter};

fn main() {

tauri::Builder::default()

.setup(|app| {

let handle = app.app\_handle(); // создаём указатель на объект нашего приложения

let info = 55;

handle.emit("update\_value", info).unwrap(); // отправляем информацию на фронт

Ok(())

})

.run(tauri::generate\_context!())

.expect("error while running tauri application");

}

##### Фронт:

import { listen, UnlistenFn } from '@tauri-apps/api/event';

import { useState, useEffect } from "react";

function App() {

const [variable, setVariable] = useState<string>('Waiting for data...');

useEffect(() => {

let unlisten: UnlistenFn | undefined;

listen<string>('update\_value', (event) => {

setVariable(event.payload); // Обновляем состояние переменной

})

.then((unlistenFn) => {

unlisten = unlistenFn; // Сохраняем функцию для отписки

});

return () => {

if (unlisten) {

unlisten();

}

};

}, []);

return (

<main className= "grid-container">

<div className= "grid-item">

<div className= "grid-item2">

Переменная из Rust:

<p>{variable}</p>

</div>

</main>

);

}

##### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### Асинхронное обновление информации на фронте

###### бэк

#![cfg\_attr(not(debug\_assertions), windows\_subsystem = "windows")]

use oracle::Connection;

use serde::Serialize;

use tauri::{AppHandle, Manager, Emitter};

use std::thread;

use tokio::time::Duration;

fn start\_sending\_data(app\_handle: AppHandle) {

thread::spawn(move || {

let mut counter = 0;

loop {

counter += 1;

app\_handle.emit("update\_value3", counter).unwrap();

thread::sleep(Duration::from\_secs(3));

}

});

}

fn main() {

setup\_oracle\_client();

tauri::Builder::default()

.setup(|app| {

let handle = app.app\_handle(); // создаём указатель на объект нашего приложения

start\_sending\_data(handle.clone()); // передаём новый указатель на объект приложения

Ok(())

})

.run(tauri::generate\_context!())

.expect("error while running tauri application");

}

###### Фронт

import { useState, useEffect } from "react";

import { listen, UnlistenFn } from '@tauri-apps/api/event';

function App() {

const [variable, setVariable] = useState<string>('Waiting for data...');

const listen\_backend = <T,>(

eventName: string,

callback: (payload: T) => void,

dependencies: any[] = []

) => {

useEffect(() => {

let unlisten: UnlistenFn | undefined;

listen<T>(eventName, (event) => {

callback(event.payload);

}).then((unlistenFn) => {

unlisten = unlistenFn;

});

return () => {

unlisten?.();

};

}, dependencies);

};

listen\_backend<number>('update\_value3', (payload) => {

setVariable(payload.toString());

});

return (

<main>

<p>Counter: {variable}</p>

</main>

);

}

export default App;

#### http JSON коммуникация

Основной вид коммуникации между разными сервисами соединенными через интернет это JSON формат. Любой запрос в сеть и любой ответ будет в JSON формате.

Это потому что фронтенд и бэкенд и любые другие сервисы связанные через интернет коммуницируют через http протокол, который может отправлять данные ТОЛЬКО в текстовом формате

На фронт нельзя отправить структуру данных или что то ещё, только string,

Поэтому все структуры данных принято переводит в текстовый формат и отправлять на фронт по сети.

##### Приятый формат JSON

Принято отправлять данные на фронт в текстовом формате JSON

const json: string = “

{

"success": true,

"message": "Запрос выполнен успешно",

"data": [

{

"id": "1",

"name": "John",

"email": "john@example.com"

}

]

}

”

Универсальная структура хранящая условно любую информацию

pub struct DbResult0 {

pub success: bool,

pub message: String,

pub data: Option<Vec<HashMap<String, String>>>,

}

##### 1. Рабочие функции бэкенда возвращают нужную структуру

Если бекенд выполнил нужные операции и нужно получить итоговый результат или структуру данных, то этот результат может быть в виде обычной переменной или структуры данных,

Но для удобства часто такой результат сразу в JSON подобном виде и формируют (но который содержит структуры данных) :

pub fn dsaemdbquerry0(query: &str) -> DbResult0 {

…

DbResult0{

success: true,

message: "Запрос выполнен успешно".to\_string(),

data, // Структура данных

}

}

##### 2. Свертывание в string JSON

Поэтому нужно cвернуть эту структуру в такой формат

{

"success": true,

"message": "Запрос выполнен успешно",

"data": [

{

"id": "1",

"name": "John",

"email": "john@example.com"

}

]

}

Это возомжно сделать с помощью

serde\_json::to\_string\_pretty(&result).unwrap()

Эта функция создаёт string объект c нужной структурой

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Например

let query = r#"

SELECT COUNT(\*) FROM E3\_ADMIN."ComponentData"

"#;

let result = dsaemdbquerry0(query);

serde\_json::to\_string\_pretty(&result).unwrap()

println!("{}", json);

Вывод будет такой

{

"success": true,

"message": "Запрос выполнен успешно",

"data": [

{

"id": "1",

"name": "John",

"email": "john@example.com"

}

]

}

##### Отправка по http

Из бэкенда создаём событие «**test»**, которое один раз отправляет json на фронт

let result = dsaemdbquerry0(query);

let json = serde\_json::to\_string\_pretty(&result).unwrap();

app\_handle.emit("test", json).unwrap();

##### 2.Принять на фронте

Создадим на фронте прослушивание конкретного события по имени «**test»**, которое принимает string

function App() {

  // Универсальный хук для подписки на события Tauri

  const listen\_backend = <T,>(

    eventName: string,

    callback: (payload: T) => void,

    dependencies: any[] = []

  ) => {

    useEffect(() => {

      let unlisten: UnlistenFn | undefined;

      listen<T>(eventName, (event) => {

        callback(event.payload);

      }).then((unlistenFn) => {

        unlisten = unlistenFn;

      });

      return () => {

        unlisten?.();

      };

    }, dependencies);

  };

  listen\_backend<string>('test', (jsonString) => {

const json: string = jsonString

}

##### 1. Развернуть JSON в структуру данных

После получения string JSON файла нужно снова развернуть его в структуру данных, чтобы можно было эффективно работать с данными из этого json

Развертывание делается с помощью парсинга JSON файлов

listen\_backend<string>('test', (jsonString) => {

const payload: **Format** = JSON.parse(jsonString) }

После парсинга мы получим сформированную структуру, к объектам которой можно обращаться через точку итд

Структура данных в которую мы соберем json зависит от определенного нами формата **Format**, который представляет собой interface

Для простых запросов из базы данных следующий интерфейс более менее универсальный:

interface Format {

success: boolean;

message: string;

data?: Array<Record<string, string>>;

}

Итого пример получения файлов из событий :

1) Cоздаём интефейс для парсинга json

2) Прослушиваем нужное событие

3) Получаем json

4) Парсим json в структуру данных

5) Записывем структуру данных в глобальную переменную

interface DbResult0 {

success: boolean;

message: string;

data?: Array<Record<string, string>>;

}

function App() {

const [message3, setMessage3] = useState<string>();

// Универсальный хук для подписки на события Tauri

const listen\_backend = <T,>(

eventName: string,

callback: (payload: T) => void,

dependencies: any[] = []

) => {

useEffect(() => {

let unlisten: UnlistenFn | undefined;

listen<T>(eventName, (event) => {

callback(event.payload);

}).then((unlistenFn) => {

unlisten = unlistenFn;

});

return () => {

unlisten?.();

};

}, dependencies);

};

listen\_backend<string>('test', (jsonString) => {

const payload: DbResult0 = JSON.parse(jsonString);

const firstItem = payload.data[0];

setMessage3(firstItem["ID"]);

});

##### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### Структура данных для передачи

Какую структуру данных мы использует в бэкенде, чтобы сформировать json – это не важно, главное чтобы финальный json был адекватный для чтения.

Я использую такую структуру, которая на фронте распознается как: Array<Record<string, string>>

{

"success": true,

"message": "Запрос выполнен успешно",

"data": [

{

"id": "1",

"name": "John",

"email": "john@example.com"

}

]

}

Однако на фронте очень важно то в какую структуру мы развернём json, тк простой доступ к значениям json важен

Я использую такую структуру но если данные простые она может не подходить.

interface DbResult0 {

success: boolean;

message: string;

data?: Array<Record<string, string>>;

}

Можно использовать абстрактные подходы:

interface ApiResponse<T = any> {

success: boolean;

message: string;

data: T;

}

// Пример для массива объектов

type User = {

id: string;

name: string;

email: string;

};

const response: ApiResponse<User[]> = {

success: true,

message: "Запрос выполнен успешно",

data: [

{ id: "1", name: "John", email: "john@example.com" },

{ id: "2", name: "Jane", email: "jane@example.com" }

]

};

// Пример для одиночного объекта

const singleResponse: ApiResponse<User> = {

success: true,

message: "Запрос выполнен успешно",

data: { id: "1", name: "John", email: "john@example.com" }

};

### Бэкенд main.rs

#### Запуск приложения

Код ниже представляет собой точку входа в программу. Он создаёт экземпляр приложения таури, запускает асинхронные процессы и что то ещё может и сразу после этого завершает мейн.

Через точку мы обновляем экземпляр приложения, добавляя какой то функционал в объект tauri::Builder::default().

fn main() {

tauri::Builder::default()

.invoke\_handler(tauri::generate\_handler![test\_oracle\_connection, symcount])

.setup(|app| {

Ok(())

})

.run(tauri::generate\_context!())

.expect("Ошибка при запуске приложения Tauri");

}

Аналог этого кода :

let app = tauri::Builder::default();

app.invoke\_handler(tauri::generate\_handler![test\_oracle\_connection, symcount]);

app.setup(|app| {

Ok(())

});

app.run(tauri::generate\_context!());

app.expect("Ошибка при запуске приложения Tauri");

#### Подключение других файлов к main.rs

Если файл events.rs находится в той же директории:

mod events;

Если файл в другой директории:

### Фронт App.tsx (React)

#### Инфо

##### ReactNode

ReactNode — это тип, предоставляемый React, который описывает любой допустимый узел (node), который можно отрендерить в компоненте. Это может быть:

Строка или число :

const element: ReactNode = "Привет, мир!";

JSX-элемент :

const element: ReactNode = <div>Привет, мир!</div>;

Массив JSX-элементов :

const elements: ReactNode = [<div key="1">Первый</div>, <div key="2">Второй</div>];

null, undefined или boolean :

Эти значения также считаются допустимыми узлами, но они не отображаются на странице:

const element: ReactNode = null;

Фрагменты (Fragments) :

const element: ReactNode = (

<>

<div>Часть 1</div>

<div>Часть 2</div>

</>

);

Компоненты React :

Любой функциональный или классовый компонент также является ReactNode:

const MyComponent = () => <div>Компонент</div>;

const element: ReactNode = <MyComponent />;

**Таким Образом:**

interface ModuleInfo {

id: string;

preview: ReactNode;

}

ModuleInfo[] = [

{

id: "camera",

preview: (

<div>

<FiCamera size={40} />

<p>Камера: подключена</p>

</div>

),

},

{ ... },

{ ... } ]

Здесь я храню массив из структур ModuleInfo, где каждый элемент имеет имя и reactNode

##### Children

##### Html

В JSX, если компонент или HTML-элемент не имеет дочерних элементов (т.е. он пустой), вы можете использовать самозакрывающийся синтаксис . Это означает, что вместо:

<Query\_preview></Query\_preview> 🡪 <Query\_preview />

#### Условный рендеринг

Рендер html по некоторому условию:

const [isWorkspaceVisible, setIsWorkspaceVisible] = useState(false);

const handleClick = () => {

setIsWorkspaceVisible((prev) => !prev);

};

<div>

{/\* Preview блок \*/}

<div className="tile" onClick={handleClick}>

{preview}

<p>{isWorkspaceVisible.toString()}</p>

</div>

{/\* Workspace блок (отображается только если isWorkspaceVisible === true) \*/}

{isWorkspaceVisible && (

<div className="workspace">

{workspace}

</div>

)}

</div>

#### div как панель на весь экран

#### Отображение структур данных

К примеру у нас есть такая структура которую нужно отобразить на экране

const [message3, setMessage3] = useState<Array<Record<string, string>>>();

##### Простой способ

<div className="row2">

<ul>

{message3?.map((item, index) => (

<li key={index}>

{Object.entries(item).map(([key, value]) => (

<div key={key}>

{key}: {value}

</div>

))}

</li>

))}

</ul>

</div>

##### Таблица из структуры Array<Record<string, string>>

<div className="grid-item2">

{message3?.length ? (

<table className="tab">

<thead>

<tr>

{Object.keys(message3[0]).map((key) => (

<th key={key}>{key}</th>

))}

</tr>

</thead>

<tbody>

{message3.map((item, index) => (

<tr key={index}>

{Object.values(item).map((value, valueIndex) => (

<td key={valueIndex}>{value}</td>

))}

</tr>

))}

</tbody>

</table>

) : (

<p>No data available</p>

)}

</div>

#### Модульная разработка

src/

├── moduels/ // папка для модулей

│ ├── ModuleGrid.tsx // создаём сетку из инстансев модулей

| |---- ModuleTile.tsx // создаём универсальную обёртку для любых модулей для стилей

│ └── modules.tsx // создаём различные реализации разных модулей

├─- App.tsx // главный компоненты который запускает main

└── App.css // стили

мы создаём модули какие нам надо различные:

**modules.tsx:**

import { ReactNode } from "react";

import { FiCamera, FiCpu, FiDatabase } from "react-icons/fi";

interface ModuleInfo {

id: string;

preview: ReactNode;

}

export const modules: ModuleInfo[] = [

{

id: "camera",

preview: (

<div>

<FiCamera size={40} />

<p>Камера: подключена</p>

</div>

),

},

{

id: "processor",

preview: (

<div>

<FiCpu size={40} />

<p>Загрузка CPU: 25%</p>

</div>

),

},

{

id: "database",

preview: (

<div>

<FiDatabase size={40} />

<p>БД: 3 активных подключения</p>

<button>Обновить</button>

</div>

),

},

];

В мейне мы получаем созданные нами модули и передаём их в элемент модуль грид:

**App.tsx:**

import "./App.css";

import ModuleGrid from "./modules/ModuleGrid";

import { modules } from "./modules/modules.tsx";

function App() {

return (

<main className="container">

<ModuleGrid modules={modules} />

</main>

);

}

export default App;

Далее мы получаем в модуль грид несколько модулей и с помощью map по очереди обрабатываем модули: каждый отдельный модуль передаём в модуль тайл ( имя и preview, которое является ReactNode - ом):

**ModuleGrid.tsx:**

import React from "react";

import ModuleTile from "./ModuleTile";

interface Props {

modules: { id: string; preview: React.ReactNode }[];

}

const ModuleGrid: React.FC<Props> = ({ modules }) => {

return (

<div className="grid">

{modules.map((mod) => (

<ModuleTile key={mod.id}>{mod.preview}</ModuleTile>

))}

</div>

);

};

export default ModuleGrid;

Далее мы здесь в качестве children получаем наш preview (который является ReactNode) и включаем его в html код. и возвращаем это код в app.tsx

**ModuleTile.tsx:**

import React from "react";

interface ModuleTileProps {

children: React.ReactNode;

}

const ModuleTile: React.FC<ModuleTileProps> = ({ children }) => {

return (

<div className="tile">

{children}

</div>

);

};

export default ModuleTile;

## Запросы в БД Rust

#### Oracle DB

##### Установка библиотек для совершения запросов

<https://www.oracle.com/database/technologies/instant-client/winx64-64-downloads.html#ic_winx64_inst>

Скачать файл

Распаковать

И внести путь в path переменных среды

Проверить подключение через простой запрос

+ проверить в cargo.toml подключение к библиотеке oracle

[dependencies]

tauri = { version = "2", features = [] }

tauri-plugin-opener = "2"

serde = { version = "1", features = ["derive"] }

serde\_json = "1"

oracle = "0.5"

+Добавить в код подключение к библиотекам OCI в main.rs (в бэкенде)

use oracle::{Connection, Error};

use serde::Serialize;

use tauri::Manager;

use std::path::Path;

##### EZConnect

Простая проверка подключения к БД

Простой способ совершения запросов в БД

###### 1) бэкенд

В самом верху main.rs

use oracle::{Connection, Error};

use serde::Serialize;

use tauri::Manager;

// Структура для возврата результата в JSON

#[derive(Serialize)]

struct DbResult {

success: bool,

message: String,

}

Функции запроса

#[tauri::command]

fn check\_oracle\_connection() -> DbResult {

let username = "E3\_ADMIN"; // Замените на ваше имя пользователя

let password = "ddbadmine3"; // Замените на ваш пароль

let connect\_string = "pme3app1:1521/E3P2 "; // Например, "localhost:1521/XE"

// Попытка подключения к базе данных

match Connection::connect(username, password, connect\_string) {

Ok(conn) => {

// Если подключение успешно, выполним простой запрос

match conn.query\_row("SELECT 1 FROM dual", &[]) {

Ok(row) => {

let value: i32 = row.get(0).unwrap(); // Чтение результата

DbResult {

success: true,

message: format!("Подключение успешно! Результат запроса: {}", value),

}

}

Err(e) => DbResult {

success: false,

message: format!("Ошибка выполнения запроса: {}", e),

},

}

}

Err(e) => DbResult {

success: false,

message: format!("Ошибка подключения к базе данных: {}", e),

},

}

}

В функции мейн

fn main() {

tauri::Builder::default()

.invoke\_handler(tauri::generate\_handler![test\_oracle\_connection]) // Регистрация команды

.setup(|app| {

// Здесь можно добавить инициализацию, если нужно

Ok(())

})

.run(tauri::generate\_context!())

.expect("Ошибка при запуске приложения Tauri");

}

###### 2) Фронт

function App() {

const [message, setMessage] = useState<string | null>(null);

const [loading, setLoading] = useState(false);

async function testConnection() {

setLoading(true); // Начинаем загрузку

setMessage(null); // Сбрасываем предыдущее сообщение

try {

// Вызываем Tauri-команду для проверки подключения

const result = await invoke<{ success: boolean; message: string }>("test\_oracle\_connection");

setMessage(result.message); // Устанавливаем сообщение с результатом

} catch (e) {

setMessage("Ошибка при вызове команды"); // Обрабатываем ошибку

} finally {

setLoading(false); // Заканчиваем загрузку

}

}

return …..

##### Работа с запросами

###### Совершение запроса

let username = "E3\_ADMIN"; // Замените на ваше имя пользователя

let password = "ddbadmine3"; // Замените на ваш пароль

let connect\_string = "pme3app1:1521/E3P2 "; // Например, "localhost:1521/XE"

match Connection::connect(username, password, connect\_string) {

Ok(conn) => {

match conn.query(query, &[]) {

Ok(rows) => {

for row in rows {

###### Вывод информации из запроса

for row in rows {

let row = row.unwrap();

for info in row.column\_info() {

println!(

"{:-30} {:-8} {}",

info.name(),

if info.nullable() { "" } else { "NOT NULL" },

info.oracle\_type()

);

}

}

###### Самый простой запрос

#![cfg\_attr(not(debug\_assertions), windows\_subsystem = "windows")]

use oracle::Connection;

#[tauri::command]

fn dsaemdbquery(query: &str) -> String {

let username = "E3\_ADMIN";

let password = "ddbadmine3";

let connect\_string = "pme3app1:1521/E3P2";

// Подключаемся к базе

let conn = match Connection::connect(username, password, connect\_string) {

Ok(c) => c,

Err(e) => return format!("Connection failed: {}", e),

};

// Выполняем запрос

match conn.query\_row(query, &[]) {

Ok(row) => row.get(0).unwrap\_or\_else(|\_| "No data".to\_string()),

Err(e) => format!("Query failed: {}", e),

}

}

###### Запрос с одной ячейкой ответа

use oracle::{Connection, Row, RowValue};

use serde::Serialize;

use tauri::{AppHandle, Manager, Emitter};

use std::thread;

use tokio::time::Duration;

use std::collections::HashMap;

fn dsaemdbquerry0(querry: &str) -> DbResult {

let username = "E3\_ADMIN"; // Замените на ваше имя пользователя

let password = "ddbadmine3"; // Замените на ваш пароль

let connect\_string = "pme3app1:1521/E3P2 "; // Например, "localhost:1521/XE"

match Connection::connect(username, password, connect\_string) {

Ok(conn) => {

match conn.query\_row(querry, &[]) {

Ok(row) => {

let value: String = row.get(1).unwrap(); // Чтение результата

DbResult {

success: true,

message: value.to\_string(),

}

}

Err(e) => DbResult {

success: false,

message: format!("Ошибка выполнения запроса: {}", e),

},

}

}

Err(e) => DbResult {

success: false,

message: format!("Ошибка подключения к базе данных: {}", e),

},

}

}

###### Запрос с ответ из нескольких колонок и нескольких строк

pub fn dsaemdbquerry0(query: &str) -> DbResult0 {

let username = "E3\_ADMIN";

let password = "ddbadmine3";

let connect\_string = "pme3app1:1521/E3P2";

// Пытаемся подключиться к БД

let conn = match Connection::connect(username, password, connect\_string) {

Ok(conn) => conn,

Err(e) => return DbResult0::error(format!("Ошибка подключения к базе данных: {}", e)),

};

// Выполняем запрос

let rows = match conn.query(query, &[]) {

Ok(rows) => rows,

Err(e) => return DbResult0::error(format!("Ошибка выполнения запроса: {}", e)),

};

// Обрабатываем результаты

let mut data = Vec::new();

for row in rows {

let row = match row {

Ok(row) => row,

Err(e) => return DbResult0::error(format!("Ошибка обработки строки: {}", e)),

};

let mut row\_data = HashMap::new();

for (i, info) in row.column\_info().iter().enumerate() {

let column\_name = info.name().to\_string();

let value: String = row.get(i).unwrap\_or\_default();

row\_data.insert(column\_name, value);

}

data.push(row\_data);

}

// Возвращаем успешный результат

if data.is\_empty() {

DbResult0::success(None)

} else {

DbResult0::success(Some(data))

}

}

#### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### Переносимость приложений с запросами

Компиляция приложения, чтобы других пользователям не пришлось устанавливать библиотеки OCI.

Идея заключается в том, чтобы все файлы OCI перенести в папку приложения, чтобы оно при билде подключило к себе эти файлы автоматически.

##### Oracle DB

1. перенести файлы instantclient-basic-windows.x64-23.7.0.25.01 в src-tauri
2. изменить tauri.conf.json добавив путь к нашим файлам которые мы перенесли

{

"$schema": "https://schema.tauri.app/config/2",

"productName": "appsql",

"version": "0.1.0",

"identifier": "com.appsql.com",

"build": {

"beforeDevCommand": "npm run dev",

"devUrl": "http://localhost:1420",

"beforeBuildCommand": "npm run build",

"frontendDist": "../dist"

},

"app": {

"windows": [

{

"title": "appsql",

"width": 800,

"height": 600

}

],

"security": {

"csp": null

}

},

"bundle": {

"active": true,

"targets": "all",

"icon": [

"icons/32x32.png",

"icons/128x128.png",

"icons/128x128@2x.png",

"icons/icon.icns",

"icons/icon.ico"

],

"resources": [

"Oracle\_OCI\_libs/\*\*/\*"

]

}

}

1. Добавить в код подключение к библиотекам OCI в main.rs (в бэкенде)

use oracle::{Connection, Error};

use serde::Serialize;

use tauri::Manager;

use std::path::Path;

+ добавить функцию устанавливающую переменные среды

fn setup\_oracle\_client() {

let exe\_path = std::env::current\_exe().unwrap();

let client\_path = exe\_path

.parent()

.unwrap()

.join("Oracle\_OCI\_libs");

std::env::set\_var("OCI\_LIB\_DIR", client\_path.to\_str().unwrap());

if cfg!(target\_os = "windows") {

let current\_path = std::env::var("PATH").unwrap\_or\_default();

let new\_path = format!("{};{}",client\_path.to\_str().unwrap(),current\_path);

std::env::set\_var("PATH",new\_path);

}

}

+ добавить в мейн перед запуском настройку переменных среды

fn main() {

setup\_oracle\_client();

tauri::Builder::default()

.invoke\_handler(tauri::generate\_handler![test\_oracle\_connection]) // Регистрация команды

.setup(|app| {

// Здесь можно добавить инициализацию, если нужно

Ok(())

})

.run(tauri::generate\_context!())

.expect("Ошибка при запуске приложения Tauri");

}

+ проверить в cargo.toml подключение к библиотеке oracle

[dependencies]

tauri = { version = "2", features = [] }

tauri-plugin-opener = "2"

serde = { version = "1", features = ["derive"] }

serde\_json = "1"

oracle = "0.5"

1. Билд релиза

Cargo tauri build

Или

npm run tauri build

## Создание библиотек

### Rust crate

Создать папку

Создать Cargo.toml

[package]

name ="src\_modules"

version = "0.1.0"

edition = "2021"

[dependencies]

…

Создать src

Внутри src создать lib.rs

В src создать свои файлы .rs

Указать на них ссылку в lib.rs как pub mod filename;