

Budowanie aplikacji desktopowych w Electron JS

Mateusz Jabłoński



Kim jestem?

- ◆ programista od 2011 roku
- ◆ głównie: Javascript / Typescript / Java / dawniej: PHP
- ◆ szkoleniowiec / trener / mentor od 2016 roku
- ◆ prywatnie tata i mąż
- ◆ ostatnio również aktor w lokalnym teatrze i gracz papierowych RPG

Zaufaj naszemu doświadczeniu

18 lat

na rynku usług IT

29 560+

przeszkolonych osób

500+

klientów biznesowych

4 877+

zorganizowanych szkoleń i warsztatów

GWARANCJA JAKOŚCI USŁUG



98%

zadowolonych klientów*

* Średnia z ankiet poszkoleniowych przeprowadzanych wśród uczestników naszych szkoleń.

Edukacja na najwyższym poziomie

sages

Wiedza specjalistyczna dla branży IT

Oferujemy szeroki katalog szkoleń z technologii mainstreamowych i specjalistycznych, wschodzących i legacy. Zajęcia prowadzimy w trybie warsztatowym, a programy są oparte o praktyczne know-how. Specjalizujemy się w prowadzeniu dedykowanych szkoleń technologicznych, których agendę dostosowujemy do potrzeb naszych klientów i oczekiwania uczestników.

Wybitni eksperci

Od początku naszego istnienia przeszkołiliśmy dziesiątki tysięcy osób, co pomogło absolwentom podnieść konkurencyjność na rynku pracy i jakość projektów realizowanych na co dzień. Nasze szkolenia prowadzą najlepsi trenerzy, a nasze produkty są oparte na najnowocześniejszej technologii. Ich niezawodność i dopasowanie do potrzeb klientów są możliwe dzięki zespołowi składającemu się z wybitnych ekspertów i ekspertek, którzy/e są na pierwszej linii teorii i praktyki tworzenia i wdrażania innowacji technologicznych.

NASZE POZOSTAŁE MARKI EDUKACYJNE

STACJA.IT

kodo/amacz
by sages

Najlepsze standardy usług edukacyjnych

Stosujemy Standard Usługi Szkoleniowej Polskiej Izby Firm Szkoleniowych, a nasze usługi realizowane są na najwyższym poziomie, o czym świadczą stale powracający klienci oraz wdrożony certyfikat ISO 9001. Metodologia prowadzonych przez nas zajęć oparta jest na współczesnych narzędziach i dostosowana do potrzeb i oczekiwania klientów.

Ponadto jesteśmy firmą wpisaną do rejestru instytucji szkoleniowych w Wojewódzkim Urzędzie Pracy w Warszawie pod nr 2.14/00133/2019.

Zaufali nam najlepsi

Wśród naszych klientów są takie firmy jak Alior Bank, OLX Group, Bank Zachodni WBK, Orange Polska, Lufthansa i wiele innych.

STUDIA PODYPLOMOWE

Wspieramy organizację zaawansowanych kierunków studiów podyplomowych. Realizujemy zajęcia na kierunkach: Data Science, Big Data i Wizualna analityka danych, AI & Data Driven Business oraz User Experience Design – projektowanie doświadczeń cyfrowych.



Instytut Informatyki
Wydział Elektroniki i Technik Informacyjnych
Politechniki Warszawskiej



AKADEMIA
LEONA KOŽMIŃSKIEGO

Uwaga

Wszelkie materiały (treści tekstowe, wideo, ilustracje, zdjęcia itp.) wchodzące w skład szkoleń, kursów i webinarów organizowanych przez Sages są objęte prawem autorskim i podlegają ochronie na mocy Ustawy o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (tekst ujednolicony: Dz.U. 2006 nr 90 poz. 631). Kopiowanie, przetwarzanie, rozpowszechnianie tych materiałów w całości lub w części jest zabronione.



Ustalenia

- ▶ Cel i agenda
- ▶ Wzajemne oczekiwania
- ▶ Pytania i dyskuje
- ▶ Elastyczność
- ▶ Otwartość i uczciwość

Agenda, czyli co nas czeka?

1. Wprowadzenie
2. Struktura projektu i podstawy
3. Integracja z Node.js i systemem
4. Interfejs użytkownika
5. Dystrybucja aplikacji
6. Zaawansowane funkcje i dobre praktyki

github.com/matwjablonski/electron-js-1125

wprowadzenie

Czym jest ElectronJS?

- ▶ Electron został stworzony przez Cheng Zhao i zespół GitHub w 2013 roku (pierwotnie nazywał się Atom Shell i miał stanowić podstawę dla edytora kodu Atom)
- ▶ miał stanowić alternatywę dla takich frameworków jak: `node-webkit` and `Chromium Embedded Framework`
- ▶ jest to framework open-source umożliwiający tworzenie aplikacji desktopowych przy użyciu technologii webowych (HTML, CSS, JavaScript)
- ▶ łączy w sobie silnik renderujący Chromium oraz środowisko uruchomieniowe Node.js
- ▶ pozwala na tworzenie aplikacji wieloplatformowych (Windows, macOS, Linux) z jednym kodem źródłowym
- ▶ popularne aplikacje zbudowane w Electronie to m.in. `Visual Studio Code`, `Slack`, `Discord`, `GitHub Desktop`

w czym tkwi siła Electrona?

- ◀ umożliwia programistom webowym tworzenie aplikacji desktopowych bez konieczności nauki nowych języków programowania czy technologii
- ◀ dostęp do natywnych funkcji systemu operacyjnego (system plików, powiadomienia, schowek, itp.) poprzez API Node.js
- ◀ duża społeczność i bogaty ekosystem bibliotek oraz narzędzi wspierających rozwój aplikacji w Electronie
- ◀ szybki rozwój i aktualizacje dzięki wykorzystaniu silnika Chromium, co pozwala na korzystanie z najnowszych funkcji webowych

Problemy i wyzwania

- ◀ aplikacje Electron mogą być zasobożerne (duże zużycie pamięci RAM i CPU) w porównaniu do natywnych aplikacji
- ◀ rozmiar aplikacji może być większy ze względu na dołączenie całego silnika Chromium
- ◀ bezpieczeństwo aplikacji wymaga dodatkowej uwagi, zwłaszcza przy korzystaniu z Node.js w rendererze
- ◀ konieczność dbania o aktualizacje zarówno Electrona, jak i zależności, aby uniknąć luk bezpieczeństwa

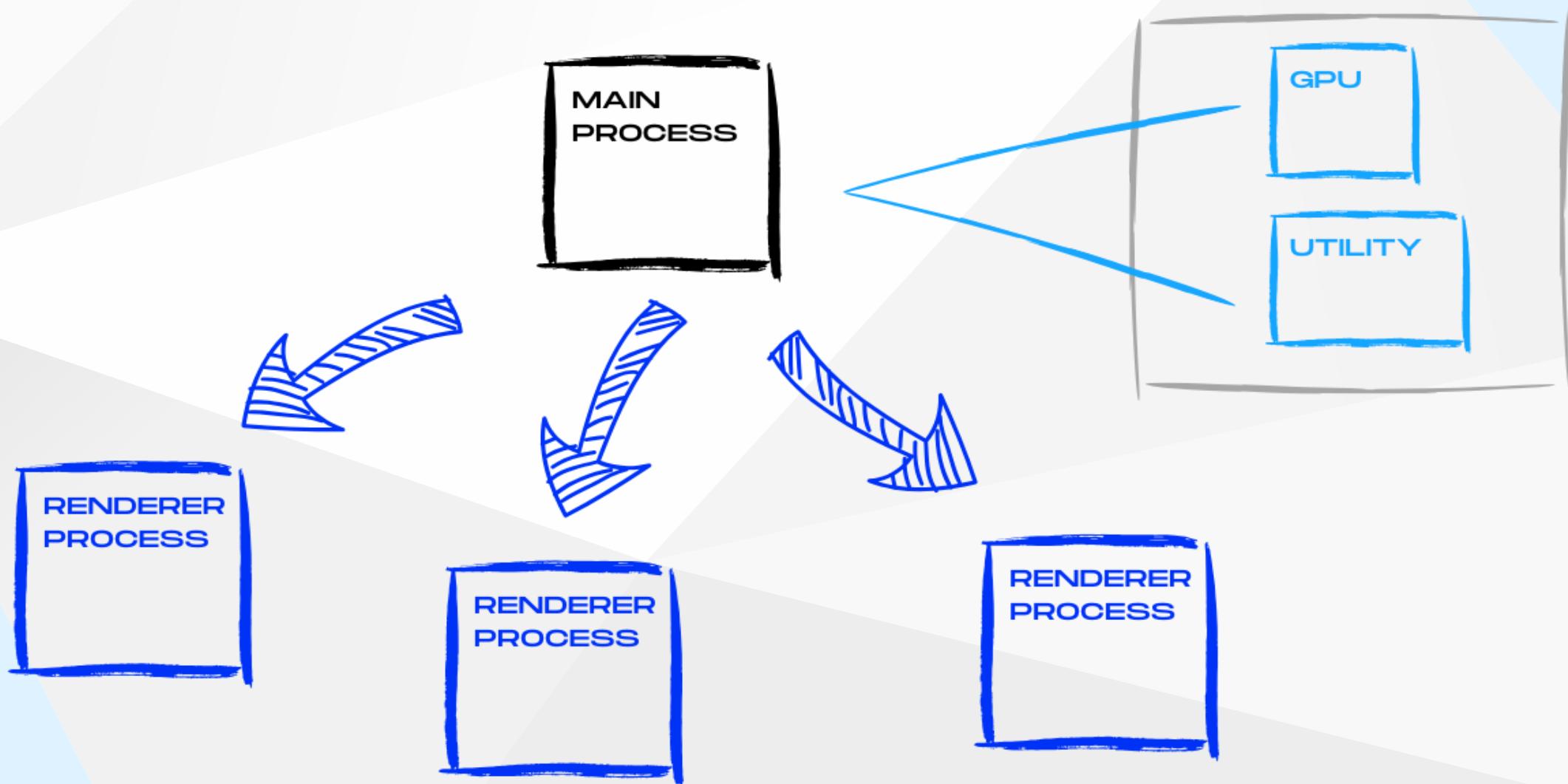
Problem, który rozwiązuje

- ◀ umożliwia tworzenie aplikacji desktopowych przy użyciu znanych technologii webowych
- ◀ eliminuje potrzebę nauki nowych języków programowania czy frameworków natywnych
- ◀ pozwala na szybkie prototypowanie i rozwój aplikacji dzięki wykorzystaniu istniejących narzędzi i bibliotek webowych
- ◀ ułatwia tworzenie aplikacji wieloplatformowych z jednym kodem źródłowym



Jak działa Electron pod spodem?

- ▶ Electron składa się z dwóch głównych procesów: **Main Process** i **Renderer Process**
- ▶ Chromium jest używane do renderowania interfejsu użytkownika w procesie renderera, podczas gdy Node.js działa w procesie głównym, umożliwiając dostęp do natywnych funkcji systemu operacyjnego
- ▶ komunikacja między tymi dwoma procesami odbywa się za pomocą mechanizmu IPC (Inter-Process Communication)
- ▶ aplikacja Electron uruchamia się od procesu głównego, który tworzy okna renderera i zarządza ich cyklem życia



IPC – komunikacja między procesami

- IPC (Inter-Process Communication) to mechanizm umożliwiający komunikację między procesem głównym a procesami renderera w aplikacji Electron
- pozwala na wysyłanie i odbieranie wiadomości między tymi dwoma procesami
- w Electronie dostępne są dwa główne moduły do obsługi IPC:
 - `ipcMain` – używany w procesie głównym do nasłuchiwanego na wiadomości od procesów renderera
 - `ipcRenderer` – używany w procesie renderera do wysyłania wiadomości do procesu głównego
- IPC jest kluczowe dla bezpiecznej i efektywnej komunikacji między warstwą UI a logiką aplikacji

Przykładowy kod:

```
// Przykład użycia IPC w Electronie
// W procesie głównym (main.js)
const { ipcMain } = require('electron');
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg); // Wyświetla wiadomość od renderera
  event.reply('asynchronous-reply', 'pong'); // Odpowiada do renderera
});

// W procesie renderera (renderer.js)
const { ipcRenderer } = require('electron');
ipcRenderer.send('asynchronous-message', 'ping'); // Wysyła wiadomość do głównego
ipcRenderer.on('asynchronous-reply', (event, arg) => {
  console.log(arg); // Wyświetla odpowiedź od głównego
});
```

Architektura aplikacji

Main process:

- ▶ tworzy okna aplikacji `BrowserWindow`
- ▶ zarządza cyklem życia aplikacji (start, zamykanie, restart, obsługa zdarzeń systemowych)
- ▶ odpowiada za IPC
- ▶ może wykonywać operacje wrażliwe lub wymagające uprawnień (np. odczyt/zapis plików)

Renderer process:

- ▶ wyświetla UI w Chromium
- ▶ obsługuje interakcje użytkownika (formularze, kliknięcia, eventy DOM)
- ▶ może korzystać z Node.js w trybie preload (wartwa pośrednia pomiędzy rendererem a Node.js)
- ▶ izolowany od systemu w celu zwiększenia bezpieczeństwa (`contextIsolation`)

... a technicznie?

Main Process

- ▶ działa jako jeden pojedynczy proces Node.js
- ▶ wykonuje kod z pliku startowego (typowo main.js)
- ▶ często zachowuje się jak backend w architekturze Electron

Renderer Process

- ▶ każde okno `BrowserWindow` działa w oddzielnym rendererze (jak oddzielna karta w Chrome)
- ▶ może ich być wiele jednocześnie
- ▶ zachowuje się jak frontend aplikacji

Kluczowe różnice między Main a Renderer

Obszar	Main Process	Renderer Process
Rola	Logika aplikacji, zarządzanie oknami, dostęp systemowy	UI, logika frontendowa
Środowisko	Node.js (pełny dostęp)	Chromium (DOM + JS)
Dostęp do systemu	Pełny	Ograniczony, sandbox
Ilość procesów	Zwykle jeden	Jeden na każde okno
Komunikacja	ipcMain	ipcRenderer
Zagrożenia / bezpieczeństwo	Ryzyko przy błędnym kodzie systemowym	Izolacja przed exploitami z UI
Typowe zadania	tworzenie okien, obsługa plików, integracje systemowe	wyswietlanie UI, obsługa użytkownika

Przykładowy plik main.js

Przykładowy kod:

```
const { app, BrowserWindow } = require('electron');

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  });

  win.loadFile('index.html');
}

app.whenReady().then(() => {
  createWindow();
});
```

Przykładowy plik preload.js

Przykładowy kod:

```
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('electronAPI', {
  sendMessage: (message) => ipcRenderer.send('asynchronous-message', message),
  onReply: (callback) => ipcRenderer.on('asynchronous-reply', callback)
});
```

Przykładowy kod w rendererze

Przykładowy kod:

```
<!DOCTYPE html>
<html>
  <head><title>My Electron App</title></head>
  <body>
    <h1>Hello, Electron!</h1>
    <button id="sendBtn">Send Message</button>
    <script>
      const sendBtn = document.getElementById('sendBtn');
      sendBtn.addEventListener('click', () => {
        window.electronAPI.sendMessage('ping');
      });

      window.electronAPI.onReply((event, arg) => {
        console.log(arg);
      });
    </script>
  </body>
</html>
```

Struktura projektu i podstawy

Konfiguracja środowiska i struktura projektu

Wymagania wstępne:

- ◆ Node.js (LTS)
- ◆ `npm` lub `pnpm` - do zarządzania pakietami
- ◆ edytor kodu, np. `VSCode`
- ◆ podstawy JS/TS + HTML/CSS

Electron Forge – oficjalne narzędzie do tworzenia aplikacji Electron

To najwygodniejsze i rekomendowane narzędzie do tworzenia nowych projektów. Powstał w latach 2018-2020 jako następca Electron Packager i Electron Builder.

Przykładowy kod:

```
npx create-electron-app my-app
```

Automatycznie wygeneruje:

- ▶ main.js
- ▶ preload.js
- ▶ gotowe skrypty w package.json

Wsparcie dla:

- ◀ buildów (Windows/macOS/Linux)
- ◀ auto-reload
- ◀ TypeScript (opcjonalnie)
- ◀ Webpack / Vite (opcjonalnie)

Electron Quick Start – oficjalny szablon GitHub

Przykładowy kod:

```
npx degit electron/electron-quick-start my-app  
cd my-app  
npm install
```

Generuje najprostszy możliwy układ:

- ▶ main.js
- ▶ index.html
- ▶ renderer.js

Vite + Electron

Najbardziej nowoczesne podejście do tworzenia aplikacji Electron z wykorzystaniem Vite jako bundlera.

Przykładowy kod:

```
npm create electron-vite
```

Electron Builder – stare, ale działa

Najstarsze narzędzie do tworzenia aplikacji Electron z wieloma opcjami konfiguracyjnymi. Dziś wypierane przez Electron Forge oraz Vite + Electron.

Przykładowy kod:

```
npx create-electron-app my-app --template=webpack
```

Struktura projektu – przykładowy układ

Przykładowy kod:

```
/project
└── package.json
└── main.js
└── preload.js
└── renderer/
    └── index.html
    └── index.js
└── assets/
    └── icon.png
```

Struktura większych projektów

Przykładowy kod:

```
/project
  |- main/      // logika procesu głównego
  |- renderer/   // UI / frontend
  |- preload/    // mosty IPC
  |- common/     // współdzielone typy, utils
  |- build/      // config do builda
  |- assets/     // zasoby statyczne
```

package.json

Przykładowy kod:

```
{  
  "name": "my-electron-app",  
  "version": "1.0.0",  
  "main": "main.js",  
  "scripts": {  
    "start": "electron .",  
    "build": "electron-builder"  
  },  
  "devDependencies": {  
    "electron": "^25.0.0",  
    "electron-builder": "^23.0.0"  
  }  
}
```

Electromon – narzędzie do automatycznego restartu aplikacji podczas developmentu

- ▶ przydatne do szybkiego testowania zmian w kodzie bez konieczności ręcznego restartu aplikacji
- ▶ monitoruje zmiany w plikach i automatycznie restartuje Electron
- ▶ wykorzystuje `chokidar` do obserwacji plików
- ▶ w bardziej zaawansowanych setupach (`Vite` , `Forge`) toolchainy same zapewniają hot reload – wtedy electronmon nie jest potrzebny

Przykładowy kod:

```
npm install --save-dev electromon
```

Electron nie wymusza framework. Mamy w tym zakresie pełną dowolność.

20 minut

zadanie nr 1

Hello Electron

- ▶ stwórz nowy projekt Electron wykorzystując oficjalny szablon GitHub
- ▶ wyświetl okno z napisem "Hello Electron"
- ▶ spróbuj wysłać wiadomość z Renderer do Main i odbierz odpowiedź

Przykładowy kod:

```
npx degit electron/electron-quick-start my-app  
cd my-app  
npm install
```



Tworzenie okna BrowserWindow

Przykładowy kod:

```
const mainWindow = new BrowserWindow({
  width: 800,
  height: 600,
  webPreferences: {
    preload: path.join(__dirname, 'preload.js'),
    contextIsolation: true,
    nodeIntegration: false,
  },
});
mainWindow.loadFile('renderer/index.html');
```

Ustawienia okien – najważniejsze opcje

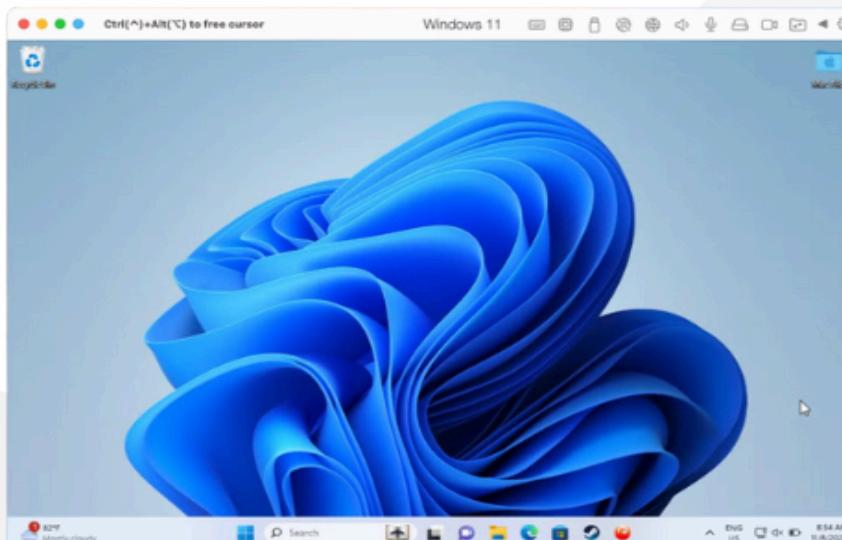
Opcja	Opis
<code>width , height</code>	szerokość i wysokość okna w pikselach
<code>resizable</code>	czy okno może być zmieniane rozmiarowo (domyślnie <code>true</code>)
<code>fullscreen</code>	czy okno ma się otwierać w trybie pełnoekranowym
<code>title</code>	tytuł okna (wyświetlany na pasku tytułu)
<code>icon</code>	ikona okna (ścieżka do pliku z ikoną)
<code>webPreferences</code>	ustawienia dotyczące renderera (np. preload, contextIsolation)
<code>show</code>	czy okno ma być od razu widoczne po utworzeniu (domyślnie <code>true</code>)

webPreferences – kluczowe opcje

Opcja	Opis
<code>preload</code>	ścieżka do pliku preload.js, który ładuje się przed rendererem
<code>contextIsolation</code>	czy izolować kontekst renderera od Node.js (zalecane <code>true</code>)
<code>nodeIntegration</code>	czy umożliwić dostęp do Node.js w rendererze (zalecane <code>false</code>)
<code>enableRemoteModule</code>	czy włączyć moduł remote (zalecane <code>false</code>)
<code>devTools</code>	czy włączyć narzędzia deweloperskie (domyślnie <code>true</code>)
<code>sandbox</code>	czy uruchomić renderer w trybie sandbox (zalecane <code>true</code>)

WYGLĄD

width
height
minWidth
maxHeight
frame
icon
visualEffectState



ZACHOWANIE

resizable
movable
fullscreen
minimizable
maximizable
closable
alwaysOnTop



BEZPIECZEŃSTWO

sandbox
webSecurity
enableRemoteModule
contextIsolation
nodeIntegration
preload



Okna modalne

- okno modalne to okno, które wymaga interakcji użytkownika przed powrotem do głównego okna aplikacji
- tworzymy je, ustawiając opcję `modal: true` i przekazując referencję do okna nadrzędnego (`parent`) podczas tworzenia nowego okna `BrowserWindow`

Przykładowy kod:

```
const mainWindow = new BrowserWindow({ /* opcje głównego okna */ });
const modalWindow = new BrowserWindow({
  parent: mainWindow,
  modal: true,
  width: 400,
  height: 300,
  webPreferences: {
    preload: path.join(__dirname, 'preload.js'),
    contextIsolation: true,
    nodeIntegration: false,
  },
});
modalWindow.loadFile('modal.html');
```

Preload scripts

W preload możesz:

- rejestrować IPC API
- udostępniać "bezpieczne" funkcje Rendererowi
- mieć dostęp do Node.js (w kontrolowanych warunkach)
- Sandbox i bezpieczeństwo: Renderer nie ma bezpośredniego dostępu do Node

Przykładowy kod:

```
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('appApi', {
  getVersion: () => ipcRenderer.invoke('get-version'),
});
```

UWAGA: Udostępniamy API "z białej listy" – tylko to, co zdefiniujemy w preload. **contextIsolation** chroni Renderer przed bezpośredniem dostępem do Node.js.

Przykład użycia w Rendererze

- ▶ Renderer NIE zna `ipcRenderer`
- ▶ komunikuje się tylko przez API `exposeInMainWorld`

Przykładowy kod:

```
document
  .getElementById('versionBtn')
  .addEventListener('click', async () => {
    const version = await window.appApi.getVersion();
    console.log(version);
});
```

Preload – więcej o bezpieczeństwie

- preload działa w kontekście Node.js, ale Renderer jest izolowany
- `contextIsolation: true` – chroni Renderer przed dostępem do Node.js
- `nodeIntegration: false` – Renderer nie ma dostępu do Node.js
- preload powinien być krótki i bezpieczny – minimalizuj kod w preload
- unikaj bezpośredniego przekazywania obiektów Node.js do Renderera
- waliduj dane przesyłane między Main, Preload i Rendererem

Content Security Policy (CSP)

- ▶ CSP pomaga zapobiegać atakom XSS
- ▶ definiujemy ją w nagłówkach HTTP lub meta tagach HTML

Przykładowa polityka CSP:

Przykładowy kod:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline';">
```

Menu aplikacji

- Electron umożliwia tworzenie natywnych menu systemowych
- tworzymy je przez `Menu.buildFromTemplate()`, gdzie przekazujemy tablicę obiektów definiujących strukturę menu

Przykładowy kod:

```
const template = [
  { label: "File", submenu: [{ role: "quit" }] },
];
Menu.setApplicationMenu(Menu.buildFromTemplate(template));
```

- можемy definiować różne role dla pozycji menu, takie jak `copy`, `paste`, `quit`, `toggleDevTools` i wiele innych
- role automatycznie dostosowują się do systemu operacyjnego, zapewniając spójne doświadczenie użytkownika

Skróty klawiszowe

- ▶ w Electronie możemy definiować skróty klawiszowe na dwa sposoby: za pomocą modułu `Menu` lub `globalShortcut`
- ▶ globalne skróty: `globalShortcut.register()`
- ▶ skróty rejestrujemy tylko po `app.whenReady()`
- ▶ należy pamiętać o `globalShortcut.unregisterAll()` przy zamykaniu aplikacji

Przykładowy kod:

```
const { globalShortcut } = require('electron');
app.whenReady().then(() => {
  globalShortcut.register('CommandOrControl+X', () => {
    console.log('Skrót klawiszowy aktywowany!');
  });
});
```

Tray

- ▶ Tray to ikona aplikacji wyświetlana w obszarze powiadomień systemu operacyjnego (system tray)
- ▶ umożliwia szybki dostęp do funkcji aplikacji bez konieczności otwierania głównego okna
- ▶ tworzymy ją za pomocą modułu **Tray** z Electron API

Przykładowy kod:

```
const { Tray, Menu } = require('electron');
let tray = null;

app.whenReady().then(() => {
  tray = new Tray('path/to/icon.png');
  const contextMenu = Menu.buildFromTemplate([
    { label: 'Show App', click: () => { /* pokaż okno aplikacji */ } },
    { label: 'Quit', role: 'quit' },
  ]);
  tray.setToolTip('My Electron App');
  tray.setContextMenu(contextMenu);
});
```

Różnice Windows, macOS i Linux

Windows

- ▶ obsługuje pełny tray z menu kontekstowym
- ▶ ikona może być kolorowa lub monochromatyczna
- ▶ wspiera animowane GIF-y
- ▶ można używać `.ico`

macOS

- ▶ Tray = Status Bar Item (globalna ikona na pasku systemowym)
- ▶ zalecane szablony monochromatyczne (template images)
- ▶ włącza automatycznie tryb jasny/ciemny
- ▶ nie wspiera GIF-ów – animacja wymaga manualnej zmiany ikon

Linux

- ◀ zależne od środowiska graficznego ([GNOME](#) , [KDE](#) , [XFCE](#))
- ◀ część środowisk wymaga dodatkowych bibliotek (AppIndicator)
- ◀ zdarza się, że tray jest domyślnie wyłączony
- ◀ format [PNG](#) najbezpieczniejszy

Przykładowy kod:

```
let icon = null;

if (process.platform === 'win32') {
  icon = 'icons/win/icon.ico';
} else if (process.platform === 'darwin') {
  icon = 'icons/mac/iconTemplate.png'; // dostosowuje się do dark/light mode
} else {
  icon = 'icons/linux/icon.png';
}

tray = new Tray(icon);
```

Animowane Tray Icons (np. synchronizacja)

Windows / Linux

- ▶ można użyć GIF
- ▶ albo serii PNG jako animacji

macOS

- ▶ nie obsługuje GIF
- ▶ należy ręcznie zmieniać ikony w interwale

Przykład animowanej ikony (wszędzie)

Przykładowy kod:

```
const frames = [
  'icons/sync/frame1.png',
  'icons/sync/frame2.png',
  'icons/sync/frame3.png',
];

let current = 0;
let interval = null;

function startSyncAnimation() {
  interval = setInterval(() => {
    tray.setImage(frames[current]);
    current = (current + 1) % frames.length;
  }, 200);
}

function stopSyncAnimation() {
  clearInterval(interval);
  tray.setImage('icons/sync/done.png');
}
```

Animacja GIF – tylko na Windows/Linux

Przykładowy kod:

```
if (process.platform !== 'darwin') {  
  tray.setImage('icons/sync.gif');  
}
```

.send vs .invoke

ipcRenderer.send(channel, data)

- ▶ komunikacja jednokierunkowa (od Renderera do Main)
- ▶ nie zwraca wartości
- ▶ odpowiedź wymaga `ipcMain.on()` i osobnego kanału do odesłania danych
- ▶ trudniejszy w kontroli przepływu i błędów
- ▶ typowy do: powiadomień, logów, zdarzeń bez oczekiwania na odpowiedź

Przykładowy kod:

```
// renderer
ipcRenderer.send('log-event', { message: 'clicked' });

// main
ipcMain.on('log-event', (event, data) => {
  console.log(data.message);
});
```

ipcRenderer.invoke(channel, data)

- ◆ komunikacja dwukierunkowa (od Renderera do Main i z powrotem)
- ◆ zwraca Promise z odpowiedzią
- ◆ prostszy w obsłudze błędów (try/catch)
- ◆ odpowiedź zwracana przez `ipcMain.handle()`
- ◆ typowy do: zapytań o dane, operacji wymagających odpowiedzi, operacji asynchronicznych

Przykładowy kod:

```
// renderer
const value = await ipcRenderer.invoke('get-user', 42);

// main
ipcMain.handle('get-user', async (event, id) => {
  return db.users.find(id);
});
```

Kiedy używać?

- ◀ `send()` - gdy nie potrzebujesz odpowiedzi lub wysyłasz event
- ◀ `invoke()` - gdy potrzebujesz wyniku albo gdy logika jest asynchroniczna

25 minut

zadanie nr 2

Zmień strukturę projektu tak, aby:

- ▶ okienko miało wymiary 800×500
- ▶ preload ma udostępniać funkcję `getAppInfo()`, która zwraca obiekt z informacjami o aplikacji (np. nazwa, wersja)
- ▶ renderer wywołuje funkcję `getAppInfo()` i wyświetla wynik
- ▶ dodaj proste menu do aplikacji
- ▶ dodaj skrót klawiszowy do otwierania DevTools (np. `Ctrl+Shift+I` lub `Cmd+Option+I` na Macu)
- ▶ dodaj ikonę tray z jedną pozycją menu (np. "Otwórz aplikację")



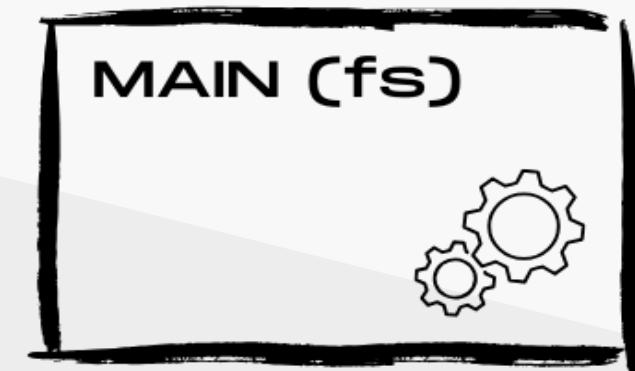
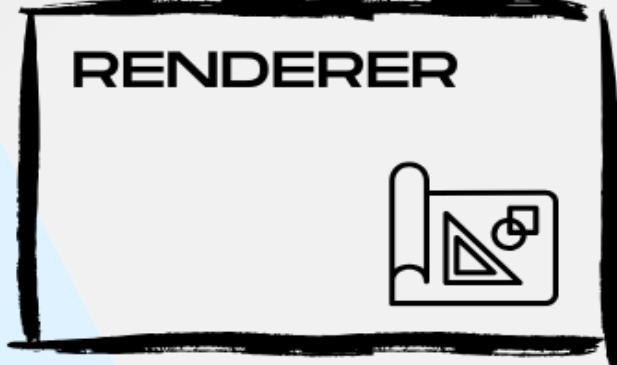
Integracja z Node.js i systemem operacyjnym

Dostęp do systemu plików (fs)

- ▶ Electron (Main Process) ma pełen dostęp do Node.js
- ▶ można korzystać bezpośrednio z `fs`, `path`, `os`, `child_process`

WAŻNE

Dostęp do plików powinien być po stronie Main - Renderer jedynie просит poprzez IPC



Odczyt pliku (fs.readFile)

Przykładowy kod:

```
// main.js
ipcMain.handle("read-file", async (_, filePath) => {
  const data = await fs.promises.readFile(filePath, "utf-8");
  return data;
});
```

Przykładowy kod:

```
// preload.js
const { contextBridge, ipcRenderer } = require("electron");
contextBridge.exposeInMainWorld("fileApi", {
  readFile: (path) => ipcRenderer.invoke("read-file", path)
});
```

Przykładowy kod:

```
// renderer.js
const content = await window.fileApi.readFile("/tmp/example.txt");
```

zapis pliku (fs.writeFile)

Przykładowy kod:

```
// main.js
ipcMain.handle("write-file", async (_, filePath, text) => {
  await fs.promises.writeFile(filePath, text, "utf-8");
});
```

Przykładowy kod:

```
// preload.js
contextBridge.exposeInMainWorld("fileApi", {
  writeFile: (path, text) => ipcRenderer.invoke("write-file", path, text)
});
```

Przykładowy kod:

```
// renderer.js
await window.fileApi.writeFile("/tmp/example.txt", "Hello, Electron!");
```

zalety podejścia z Preload i IPC

Takie podejście:

- ◀ jest bezpieczne
- ◀ pozwala kontrolować do czego renderer ma dostęp
- ◀ działa na wszystkich OS

Przykłady typowych zastosowań fs w Electron

- ▶ zapisywanie konfiguracji użytkownika
- ▶ wczytywanie plików JSON
- ▶ ładowanie i zapisywanie dokumentów aplikacji
- ▶ tworzenie katalogów aplikacji
- ▶ `cache` i `temp` danych aplikacji

WSKAZÓWKA

Używaj `app.getPath` zamiast ręcznych ścieżek

Co zwraca app.getPath?

- ◀ `appData` - katalog aplikacji (Windows)
- ◀ `home` - katalog domowy użytkownika
- ◀ `userData` - katalog do przechowywania danych aplikacji
- ◀ `temp` - katalog tymczasowy systemu
- ◀ `desktop` - katalog pulpitu użytkownika

Przykładowy kod:

```
const { app } = require("electron");
const path = require("path");

const userDataPath = app.getPath("userData");
const configPath = path.join(userDataPath, "config.json");
```

Dialogi systemowe

Electron oferuje natywne okna:

- ◆ open-dialog (wybór pliku/folderu)
- ◆ save-dialog (zapis pliku)
- ◆ message-box (informacje, ostrzeżenia)

Działa tylko w **Main Process**.

Przykładowy kod:

```
// main.js
const { dialog, ipcMain } = require('electron');

ipcMain.handle("open-dialog", async () => {
  const result = await dialog.showOpenDialog({
    properties: ["openFile"],
    filters: [{ name: "Text Files", extensions: ["txt"] }]
  });
  return result.filePaths;
});
```

Przykładowy kod:

```
// preload.js
const { contextBridge, ipcRenderer } = require("electron");
contextBridge.exposeInMainWorld("dialogApi", {
  openFile: () => ipcRenderer.invoke("open-dialog")
});
```

Przykładowy kod:

```
// renderer.js
const filePaths = await window.dialogApi.openFile();
console.log(filePaths);
```

Przykładowy kod:

```
// main.js
ipcMain.handle("save-dialog", async () => {
  return dialog.showSaveDialog({
    defaultPath: "document.txt"
  });
});
```

Przykładowy kod:

```
// preload.js
contextBridge.exposeInMainWorld("dialogApi", {
  saveFile: () => ipcRenderer.invoke("save-dialog")
});
```

Przykładowy kod:

```
// renderer.js
const { filePath } = await window.dialogApi.saveFile();
console.log(filePath);
```

Message Box

Można używać zamiast HTMLowych dialogów. Wygląd jest natywny dla OS.

Przykładowy kod:

```
// main.js
ipcMain.handle("show-message-box", async () => {
  return dialog.showMessageBox({
    type: "info", // info / warn / confirm / error
    title: "Information",
    message: "This is an info message",
    buttons: ["OK"]
  });
});
```

Przykładowy kod:

```
// preload.js
contextBridge.exposeInMainWorld("dialogApi", {
  showMessage: () => ipcRenderer.invoke("show-message-box")
});
```

Przykładowy kod:

```
// renderer.js
const response = await window.dialogApi.showMessage();
console.log(response);
```

Obsługa przycisków w Message Box

Przykładowy kod:

```
dialog
  .showMessageBox({
    type: "warning",
    message: "Czy na pewno chcesz usunąć?",
    buttons: ["Tak", "Nie"],
  })
  .then(result => {
    if (result.response === 0) {
      // kliknięto TAK
    } else {
      // kliknięto NIE
    }
  });
}
```

Moduł clipboard

- ◆ służy do interakcji ze schowkiem systemowym
- ◆ pozwala na kopiowanie i wklejanie tekstu, obrazów oraz plików

Przykładowy kod:

```
const { clipboard } = require("electron");

clipboard.readText(); // odczytywanie tekstu ze schowka

clipboard.writeText("Hello from Electron!"); // kopiowanie tekstu do schowka
```

Zastosowanie:

- ◆ kopiowanie do schowka
- ◆ "Copy to clipboard" w UI
- ◆ integracja z edytorami

Moduł shell

- ◆ umożliwia otwieranie plików i URLi za pomocą domyślnych aplikacji systemowych

Przykładowy kod:

```
const { shell } = require("electron");

shell.openExternal("https://electronjs.org"); // otwieranie URL w domyślnej przeglądarce
shell.openPath("/Users/user/file.pdf"); // otwieranie pliku za pomocą domyślnej aplikacji
```

Zastosowanie:

- ◆ otwieranie dokumentów PDF
- ◆ otwieranie folderów w eksploratorze
- ◆ otwieranie linków poza aplikacją

Moduł notifications

Przykładowy kod:

```
const { Notification } = require("electron");
new Notification({
  title: "Zakończono!",
  body: "Proces eksportu został ukończony."
}).show();
```

Plusy:

- ▶ wyglądają jak natywne notyfikacje OS
- ▶ nie wymagają żadnych zależności
- ▶ działają z **Main** lub **Renderer** (lepiej z **Main**)

25 minut

zadanie nr 3

- ▶ do aplikacji dodaj funkcjonalność zapisywania i odczytywania plików tekstowych z systemu plików użytkownika
- ▶ dodaj przycisk "Otwórz plik", który otworzy okno dialogowe wyboru pliku i wczyta jego zawartość do aplikacji
- ▶ dodaj przycisk "Zapisz plik", który otworzy okno dialogowe zapisu pliku i zapisze aktualną zawartość aplikacji do wybranego pliku
- ▶ pokaż powiadomienie systemowe po pomyślnym zapisaniu pliku



Interfejs użytkownika

Electron + frameworki frontendowe

Electron działa w oparciu o Chromium, więc każdy framework webowy działa natywnie.

- ▶ React
- ▶ Vue
- ▶ Svelte
- ▶ Angular
- ▶ SolidJS
- ▶ Astro
- ▶ Vanilla JS



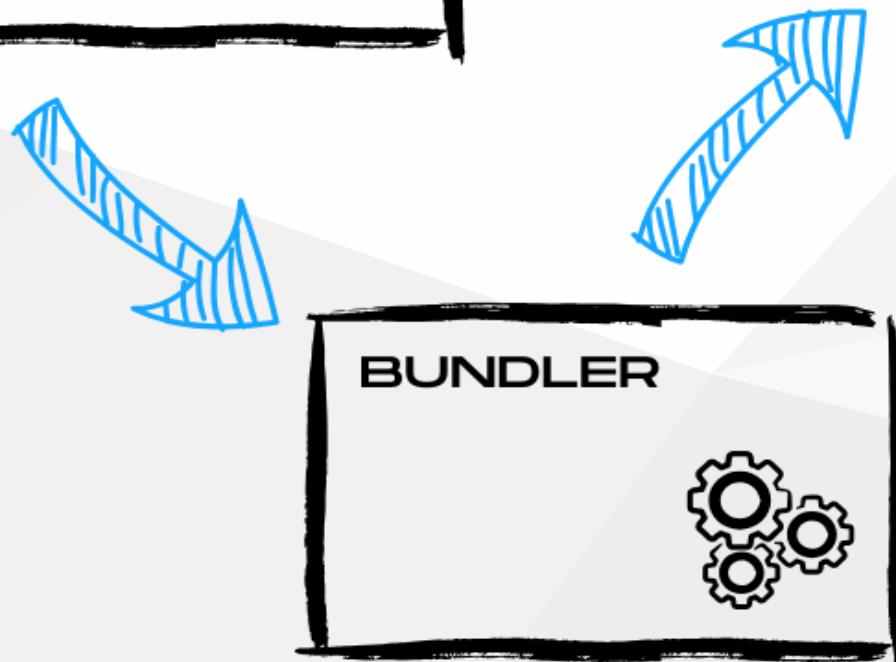
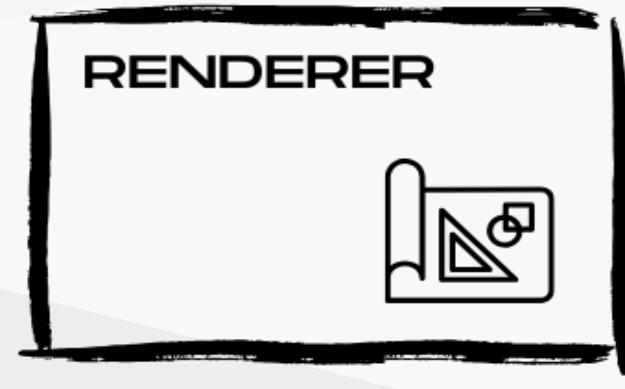
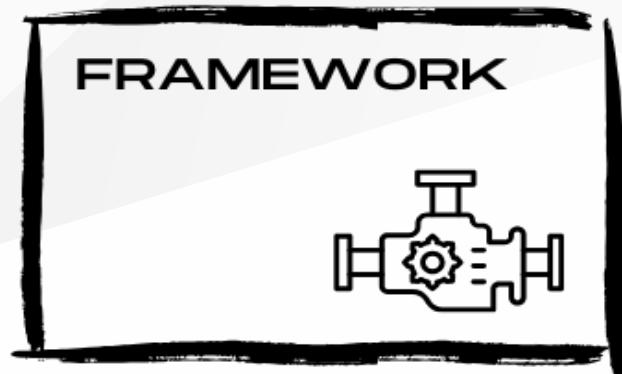
Architektura – Electron + React/Vue/Svelte

UI jest po stronie **Renderer Process**

- ◆ **Build** frameworka generuje bundle (np. `dist`)
- ◆ **BrowserWindow** ładuje `index.html`

Przykładowy kod:

```
// main.js
mainWindow.loadURL("http://localhost:5173"); // dev (Vite)
mainWindow.loadFile("dist/index.html");      // prod
```



Electron + React (najpopularniejsza kombinacja)

Zalety:

- ◆ świetny ekosystem
- ◆ gotowe UI libraries
- ◆ duże projekty komercyjne (VSCode, Discord)

Minimalny setup:

- ◆ Vite + React
- ◆ preload API dla bezpieczeństwa
- ◆ komunikacja z Main przez IPC

WAŻNE

React działa w Renderer, ale NIE ma dostępu do Node (sandbox).

Electron + Vue

Zalety:

- ▶ prosta integracja
- ▶ lekki runtime
- ▶ dobra dokumentacja dla Electron + Vue + Vite

Częste projekty:

- ▶ panele sterowania
- ▶ edytory i narzędzia graficzne

PLUSY: Reaktywność, prosty stan

Electron + Svelte / SvelteKit

Zalety:

- ▶ minimalny overhead
- ▶ super szybki development
- ▶ idealne do mniejszych i średnich aplikacji

WSKAZÓWKA

Svelte generuje bardzo małe bundle, a to przekłada się na niższe zużycie RAM

Porównanie frameworków

Framework	Zalety	Wady
React	największy ekosystem, MUI, Next UI	największy bundle
Vue	intuicyjny, szybki	mniejszy ekosystem desktopowy
Svelte	lekki, szybki, minimalny	mniejsze community
Vanilla JS	zero zależności	trudniejszy rozwój UI

Stylowanie w Electron – podstawy

- ◆ **Renderer** to po prostu strona webowa - stylowanie działa identycznie jak w przeglądarce
- ◆ możesz używać CSS, Sass, Tailwind, Material UI itp.
- ◆ możesz też korzystać z **CSS-in-JS** , **Styled Components** itp.
- ◆ należy pamiętać, że Electron używa Chromium, więc nie wszystkie funkcje CSS mogą być dostępne

WSKAZÓWKA

Unikaj inline CSS. Jest to trudne testowanie i skalowanie.

Używanie TailwindCSS

Dlaczego Tailwind?

- ◀ szybka budowa UI
- ◀ minimalny runtime
- ◀ świetnie nadaje się do Electron (mało narzutu)

Używanie Material UI (React)

Zalety:

- gotowe komponenty
- zgodne z Google Material Design
- bardzo szybkie prototypowanie

Minusy:

- największy rozmiar bundle
- wymaga React - większe zużycie RAM

Przykładowy kod:

```
import Button from "@mui/material/Button";  
  
<Button variant="contained">Zapisz</Button>
```

Stylowanie – najlepsze praktyki

- ▶ używaj **CSS modules** lub **Styled Components** do izolacji stylów
- ▶ unikaj globalnych stylów, aby zapobiec konfliktom
- ▶ testuj wygląd na różnych rozdzielczościach ekranu
- ▶ korzystaj z narzędzi do analizy wydajności CSS, aby unikać nadmiernego obciążenia
- ▶ używaj CSS variables (np. do themingu)
- ▶ renderer = UI - unikać Node.js
- ▶ przy większych projektach - **UI kit** + **Storybook**

25 minut

zadanie nr 4

- ▶ dodaj konfigurację Tailwinda do projektu Electron
- ▶ ostyluj UI Tailwindem
- ▶ dodaj tryb **dark/light** (CSS variables)

Przykładowy kod:

```
npm install -D tailwindcss postcss autoprefixer @tailwindcss/cli  
npx tailwindcss init -p
```



Dystrybucja aplikacji

Opcje budowania aplikacji Electron

Electron oferuje dwa główne narzędzia:

- ◆ **electron-packager**

- ◆ proste narzędzie do pakowania aplikacji (generuje folder z plikami aplikacji, tzw. surowe artefakty)
- ◆ brak wsparcia dla instalatorów i auto-update (bez pluginów)

- ◆ **electron-builder**

- ◆ bardziej zaawansowane narzędzie z obsługą instalatorów i aktualizacji
- ◆ tworzy instalatory
- ◆ tworzy paczki **.dmg** , **.exe** , **.AppImage** , **.deb**
- ◆ pełne wsparcie **auto-updater**
- ◆ ogrom możliwości konfiguracyjnych

Instalacja electron-builder

Przykładowy kod:

```
npm install --save-dev electron-builder
```

Dodanie skryptu do package.json:

Przykładowy kod:

```
"scripts": {  
  "build": "electron-builder"  
}
```

WAŻNE

Electron-builder sam wykrywa OS i generuje właściwy typ instalatora.

Minimalna konfiguracja electron-builder

W pliku package.json:

Przykładowy kod:

```
"build": {  
  "appId": "com.example.myapp",  
  "productName": "MyApp",  
  "directories": {  
    "buildResources": "assets"  
  }  
}
```

Wyjaśnienia:

- ◆ **appId** - wymagane na macOS
- ◆ **productName** - nazwa aplikacji w instalatorze
- ◆ **buildResources** - ikony i grafiki

Tworzenie buildów – praktyka

Windows (exe + nsis installer):

Przykładowy kod:

```
npm run build -- --win
```

macOS (.dmg + .app):

Przykładowy kod:

```
npm run build -- --mac
```

Linux (.AppImage + .deb):

Przykładowy kod:

```
npm run build -- --linux
```

WSKAZÓWKA

Budowanie macOS wymaga macOS (Apple narzuca ograniczenia).

Tworzenie instalatorów – instalator NSIS (Windows)

Najpopularniejszy instalator dla Windows.

Przykładowy kod:

```
"build": {  
  "nsis": {  
    "oneClick": false,  
    "perMachine": true,  
    "allowToChangeInstallationDirectory": true  
  }  
}
```

Zalety:

- ◆ instalacja jednym kliknięciem (lub wizard)
- ◆ autostart, skróty, odinstalowywanie

Tworzenie instalatorów – instalator DMG (macOS)

Zalety:

- ▶ natywne doświadczenie macOS
- ▶ podpisywanie aplikacji (code signing)
- ▶ można dodać własne tło DMG

WAŻNE

Aplikacje muszą być podpisane do dystrybucji poza dev. Do podpisania aplikacji wymagane jest konto Apple Developer (konto płatne).

Tworzenie instalatorów – Linux (.AppImage/.deb)

Najpopularniejsze formaty:

- ◀ **.AppImage** – działa praktycznie na każdym Linux (nie wymaga instalacji, ale nie jest oficjalnym formatem)
- ◀ **.deb** – Ubuntu/Debian
- ◀ **.snap** – Ubuntu
- ◀ **.rpm** – Fedora/RedHat

Zalety Electron-builder:

Automatycznie tworzy wszystkie paczki jeśli nie ograniczymy konfiguracji.

Auto-update – jak działa?

`electron-builder` posiada moduł `electron-updater`, który:

- ▶ obsługuje aktualizacje różnicowe
- ▶ wspiera Windows, macOS, Linux
- ▶ wymaga hostingu dla paczek aktualizacji

Schemat działania:

1. Aplikacja sprawdza nową wersję
2. Pobiera paczkę update
3. Restartuje aplikację
4. Aktualizacja instaluje się automatycznie

Konfiguracja auto-updater

Przykładowy kod:

```
npm install electron-updater
```

Przykładowy kod:

```
import { autoUpdater } from "electron-updater";
autoUpdater.checkForUpdatesAndNotify();
```

WAŻNE

Auto-updater wymaga serwera do hostowania plików aktualizacji (np. GitHub Releases, własny serwer HTTP).

Kanały aktualizacji

Electron wspiera tzw. release channels:

- ▶ stable
- ▶ beta
- ▶ alpha
- ▶ nightly

Przykład w package.json:

Przykładowy kod:

```
"publish": [{"  
  "provider": "github",  
  "owner": "my-org",  
  "repo": "my-repo"  
}]
```

Jak działa auto-update krok po kroku?

1. Robimy commit z nową funkcją
2. Pobijamy wersję w package.json
3. Uruchamiamy build
4. Wrzucamy paczki na GitHub Releases
5. Aplikacja wykrywa nową wersję
6. Aplikacja pokazuje powiadomienie
7. Po restarcie – działa najnowsza wersja

20 minut

zadanie nr 5

- ▶ zbuduj i spakuj aplikację Electron na wybraną platformę (Windows, macOS, Linux)
- ▶ użyj do tego `electron-builder` lub `electron-packager`
- ▶ przygotuj instalator dla aplikacji



Zaawansowane funkcje i dobre praktyki

Dlaczego bezpieczeństwo jest krytyczne?

- ◀ zestaw Node.js + Chromium jest potężnym połączeniem, ale niesie ze sobą ryzyko
- ◀ ataki typu XSS, RCE i inne mogą zagrozić Twojej aplikacji
- ◀ użytkownicy ufają aplikacjom desktopowym bardziej niż webowym – zawierają często wrażliwe dane
- ◀ naruszenie bezpieczeństwa może prowadzić do utraty danych, reputacji i zaufania użytkowników

Najczęstsze błędy

- ◀ włączony `nodeIntegration`
- ◀ `eval()` w Renderer
- ◀ ładowanie stron z internetu

contextIsolation

Najważniejsza flaga bezpieczeństwa w Electron.

Działanie:

- ◆ oddziela window renderera od window preloada
- ◆ uniemożliwia modyfikowanie API przez złośliwy kod

Przykładowy kod:

```
webPreferences: {  
  contextIsolation: true  
}
```

WAŻNE

Zawsze ustawiaj `contextIsolation: true`. Renderer będzie miał dostęp tylko do tego, co świadomie wystawisz.

sandbox

Sandbox aktywuje tryb podobny do Chrome.

Przykładowy kod:

```
webPreferences: {  
  sandbox: true  
}
```

Zalety:

- ◀ Renderer działa jak "strona w przeglądarce"
- ◀ brak bezpośredniego Node.js
- ◀ zdecydowanie bezpieczniejsze

Preload jako bezpieczna warstwa

Preload.js to jedyny bezpieczny most między frontendem a Node.

W preload możesz:

- ▶ wystawiać ograniczone API do Renderer
- ▶ używać `contextBridge` do bezpiecznej komunikacji
- ▶ wykorzystywać IPC do komunikacji z Main Process

Przykładowy kod:

```
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('api', {
  send: (channel, data) => {
    const validChannels = ['toMain'];
    if (validChannels.includes(channel)) {
      ipcRenderer.send(channel, data);
    }
  },
});
```

WAŻNE

Nigdy nie udostępniaj pełnego `require` w Rendererze! Nie udostępniaj pełnych funkcji `fs`, `shell`.

Principle of Least Privilege

- ▶ jest to zasada minimalnych uprawnień
- ▶ każda część aplikacji powinna mieć tylko te uprawnienia, które są absolutnie niezbędne do jej działania
- ▶ ogranicz dostęp do systemu plików, sieci i innych zasobów
- ▶ regularnie przeglądaj i aktualizuj uprawnienia

Przykłady Principle of Least Privilege w praktyce (Electron)

Renderer nie powinien mieć dostępu do Node.js

Przykładowy kod:

```
contextIsolation: true,  
nodeIntegration: false,  
sandbox: true
```

Renderer dostaje tylko bezpieczne metody

zamiast pełnego `fs`, wystaw tylko to, co potrzebne:

Przykładowy kod:

```
contextBridge.exposeInMainWorld('fileAPI', {  
  readFile: (path) => ipcRenderer.invoke('read-file', path)  
});
```

Main wykonuje operacje na plikach, a Renderer tylko "prosi"

Renderer nie może przypadkiem skasować katalogu użytkownika

Principle of Least Privilege ma kolosalne znaczenie w Electronie, ponieważ łączy on potęgę Node.js z interfejsem webowym. Principle of Least Privilege rozdziela te dwa światy, minimalizując ryzyko ataków i nadużyć.

Content Security Policy (CSP)

CSP chroni Renderer przed atakami XSS.

Przykładowy kod:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline';">
```

Najważniejsze zasady:

- ◆ nie zezwalamy na unsafe-eval
- ◆ nie ładujemy JS z internetu
- ◆ nie używamy inline scriptów

Typowe problemy z wydajnością Electron

- ▶ każde okno to osobny proces Chromium
- ▶ zbyt duże bundle
- ▶ renderery otwarte w tle
- ▶ nieusunięte event listenery
- ▶ leaky IPC (niezamknięte kanały komunikacji)

Jak poprawić wydajność?

- ◀ **Lazy loading** okien - twórz okna tylko wtedy, gdy potrzebne
- ◀ używanie **Vite** / **SWC** / **esbuild** - zmniejsza rozmiar bundla
- ◀ wyłącz background throttling - jeśli aplikacja wykonuje operacje w tle:

Przykładowy kod:

```
backgroundThrottling: false
```

- ◀ minimalizuj IPC - IPC jest wolniejsze niż zwykłe wywołania JS

- ◆ używaj offscreen rendering (dla mediów, canvas)

WSKAZÓWKA

Offscreen rendering to tryb, w którym zawartość okna jest renderowana poza ekranem, co pozwala na bardziej efektywne zarządzanie zasobami graficznymi i może poprawić wydajność aplikacji, zwłaszcza podczas pracy z mediami lub grafiką.

Offscreen rendering uruchamia się przez ustawienie `offscreen: true` w `webPreferences` okna:

Przykładowy kod:

```
webPreferences: {  
  offscreen: true  
}
```

Monitorowanie zużycia pamięci

1. Electron + Chrome DevTools/Performance

2. Main Process:

Przykładowy kod:

```
const { app } = require('electron');

app.getAppMetrics()
```

Renderer:

Chrome Performance timeline, Memory view.

WSKAZÓWKA

Unikaj trzymania dużych obiektów w globalnych zmiennych.

Najczęstsze błędy i pułapki

120

Błąd 1 – włączony nodeIntegration

Przykładowy kod:

```
nodeIntegration: true
```

- ◆ pozwala wykonywać dowolny kod systemowy w Renderer
- ◆ krytyczne zagrożenie bezpieczeństwa

Rozwiążanie:

- ◆ `nodeIntegration: false`
- ◆ używaj `preload`

Błąd 2 – IPC bez walidacji

- ▶ renderer wysyła niezweryfikowane dane
- ▶ main wykonuje operacje bez kontroli

Rozwiążanie:

- ▶ każde wywołanie IPC traktować jak API
- ▶ walidować wejście (Zod, yup)
- ▶ mocno ograniczać powierzchnię API

Błąd 3 – ładowanie stron z internetu

Przykładowy kod:

```
loadURL("https://example.com")
```

Dlaczego nie?

Strona może załadować złośliwy skrypt, który zyska dostęp do API preloada.

Rozwiążanie:

- ▶ używać lokalnych plików
- ▶ jeśli musisz ładować z internetu, to:
 - ▶ `sandbox: true`
 - ▶ `contextIsolation: true`
 - ▶ `nodeIntegration: false`

Błąd 4 – Brak CSP i inline scripts

Skutki:

- ▶ łatwe XSS
- ▶ możliwy dostęp do API preloada

Rozwiążanie:

- ▶ zawsze używać CSP
- ▶ przenieść skrypty inline do osobnych plików

Błąd 5 – zbyt duże Main Process

Problem:

Main Process robi za dużo:

- ▶ logika
- ▶ fetch
- ▶ pliki
- ▶ state

Rozwiązanie:

- ▶ DDD lub modułowa architektura
- ▶ przeniesienie logiki do osobnych modułów
- ▶ renderer komunikuje się tylko przez API

Błąd 6 – Brak aktualizacji

Skutki:

- ▶ luki bezpieczeństwa
- ▶ błędy stabilności
- ▶ brak nowych funkcji

Rozwiążanie:

- ▶ regularne aktualizacje
- ▶ automatyczne mechanizmy aktualizacji

Oficjalne źródła dokumentacji

- ▶ <https://www.electronjs.org/docs>
- ▶ <https://www.electron.build>
- ▶ <https://www.electronforge.io/>

WSKAZÓWKA

Dokumentacja Electrona jest dobra, ale rozproszona. Najlepiej zaczynać od API a następnie przejść do tutoriali i przykładów dostępnych na oficjalnej stronie.

Spółeczności

- ▶ Reddit: r/electronjs
- ▶ GitHub Discussions (Electron, Electron-Builder, Vite-Electron)
- ▶ Discord Electron (oficjalny)
- ▶ StackOverflow (bardzo dużo pytań - realne problemy)

15 minut

zadanie nr 6

- ▶ dodaj CSP (Content Security Policy) do swojej aplikacji Electron
- ▶ zmierz czas uruchamiania aplikacji
- ▶ sprawdź zużycie pamięci



129

Dodatkowe pytania?



Dlaczego Sages?

sages.pl



Rozwiązuje my problemy firm

Zapewniamy kompleksowe wsparcie w zakresie IT, poparte wieloletnim doświadczeniem i wysokimi kompetencjami zespołu naszych ekspertów.



Sages w liczbach

Nasze doświadczenie jest poparte 18-letnią działalnością w branży IT. W tym czasie przeszkoliliśmy ponad 29 tys. osób i zorganizowaliśmy prawie 5 tys. szkoleń oraz warsztatów.



Działalność badawcza oraz rozwiązań AI

Działamy na wielu obszarach - wdrażamy stworzony przez Politechnikę Warszawską system Omega-PSIR oraz tworzymy i rozwijamy własne narzędzia oparte na sztucznej inteligencji.

Poznaj te szkolenia!



W ofercie Sages znajdziesz kompleksowe szkolenia w kategoriach **Platforma Java** danych, dzięki którym możesz dalej rozwijać się w wybranym przez siebie kierunku!

SPECJALNA OFERTA

Zniżka 15% tylko dla Stacjowiczów na szkolenia gwarantowane z kategorii **Platforma Java** na hasło

STACJA15

Sprawdź całą ofertę:
sages.pl/szkolenia

Programowanie w języku Java

Sprawdź program
pod QR kodem



Zaawansowane aspekty języka Java



Ankieta

czyli jak mi poszło?

tinyurl.com/yzjy55ym



Dziękuję za uwagę

Mateusz Jabłoński

mail@mateuszjablonski.com

mateuszjablonski.com

STACJA.IT

JABŁOŃSKI

sages