



STŘEDNÍ ŠKOLA PRŮMYSLOVÁ
A UMĚLECKÁ, OPAVA

ZÁVĚREČNÁ STUDIJNÍ PRÁCE

dokumentace

Mobilní aplikace pro ovládání funkcí počítače

Jiří Gebauer



FOSS-Deck

Obor: 18-20-M/01 INFORMAČNÍ TECHNOLOGIE
se zaměřením na počítačové sítě a programování

Třída: IT4

Školní rok: 2025/2026

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně a uvedl veškeré použité informační zdroje.

Souhlasím, aby tato studijní práce byla použita k výukovým účelům na Střední průmyslové a umělecké škole v Opavě, Praskova 399/8.

V Opavě 6. 1. 2026

podpis autora práce

ABSTRAKT

Tato práce se zabývá návrhem a realizací systému pro vzdálené ovládání vybraných funkcí osobního počítače pomocí mobilního zařízení. Řešení je založeno na klient-server architektuře, kde serverová část běží na osobním počítači a klientská část je realizována jako webová aplikace optimalizovaná pro mobilní zařízení. Komunikace mezi klientem a serverem probíhá prostřednictvím protokolu WebSocket. Serverová aplikace napsaná v Rustu zajišťuje ovládání systémových funkcí operačního systému, ovládání hlasitosti zvuku, ovládání multimediálního přehrávání a dalších vybraných akcí. Klientská aplikace vytvořená pomocí frameworku Tauri poskytuje uživatelské rozhraní ve formě konfigurovatelné sady ovládacích prvků, které umožňují jednoduché a intuitivní ovládání počítače na dálku. Součástí řešení je také mechanismus párování zařízení a autentizace, který zajišťuje základní úroveň zabezpečení komunikace.

Klíčová slova

klient, server, WebSocket, Rust, Tauri, párování, autentizace

ABSTRACT

This thesis deals with the design and implementation of a system for remote control of selected personal computer functions using a mobile device. The solution is based on a client-server architecture, where the server part runs on a personal computer and the client part is implemented as a web application optimized for mobile devices. Communication between the client and server takes place via the WebSocket protocol. The server application, written in Rust, provides control of operating system functions, audio volume control, multimedia playback control, and other selected actions. The client application, created using the Tauri framework, provides a user interface in the form of a configurable set of controls that enable simple and intuitive remote control of the computer. The solution also includes a device pairing and authentication mechanism that provides a basic level of communication security.

Keywords

client, server, WebSocket, Rust, Tauri, pairing, authentication

OBSAH

ÚVOD.....	5
1 ARCHITEKTURA A POUŽITÉ TECHNOLOGIE.....	6
1.1 KLIENT-SERVER ARCHITEKTURA A KOMUNIKAČNÍ MODEL	6
1.2 WEBSOCKET KOMUNIKACE	6
1.3 POUŽITÉ TECHNOLOGIE	6
2 ZPŮSOBY ŘEŠENÍ A POUŽITÉ POSTUPY.....	7
2.1 ZAČÁTEK PROJEKTU	7
2.2 PRVNÍ BUILD PRO TELEFON.....	7
2.3 PŘIDÁNÍ PÁROVÁNÍ	7
2.4 ÚPRAVA UI	8
2.5 PŘIDÁNÍ DALŠÍCH FUNKCÍ	9
2.6 BUILD APLIKACÍ	9
3 IMPLEMENTACE JEDNÉ FUNKCE NA PC	10
3.1 FORMÁT PŘÍKAZU	10
3.2 PŘIJETÍ ZPRÁVY	10
3.3 KONTROLA AUTENTIZACE A PŘEDÁNÍ DO APLIKAČNÍ LOGIKY	10
3.4 IMPLEMENTACE TOGGLEMicMUTE V HANDLERU PŘÍKAZŮ.....	11
3.5 PŘÍSTUP K MIKROFONU VE WINDOWS	11
4 IMPLEMENTACE JEDNÉ FUNKCE NA MOBILU	12
4.1 STAV KLIENTA.....	12
4.2 DEFINICE AKCE A „OPTIMISTIC UI“	12
4.3 ODESLÁNÍ PŘÍKAZU NA SERVER.....	13
4.4 PŘÍJEM ODPOVĚDÍ: SYNCHRONIZACE STAVU Z PC	13
4.5 HEARTBEAT: PRAVIDELNÉ DOTAZOVÁNÍ NA STAV.....	13
ZÁVĚR	14
SEZNAM POUŽITÝCH INFORMAČNÍCH ZDROJŮ	15

ÚVOD

Cílem této práce bylo vytvořit mobilní aplikaci jako alternativu k zařízení Stream Deck od firmy Elgato. K tomuto tématu jsem se rozhodl z toho důvodu, že oficiální mobilní aplikace tohoto zařízení má některé funkce dostupné pouze po zaplacení a zároveň jsem nenašel jinou funkční alternativu, která by byla volně dostupná a splňovala mé požadavky.

Navržené řešení se skládá ze dvou aplikací. První z nich je serverová aplikace, která běží na počítači a vykonává jednotlivé funkce na základě příkazů zaslaných z mobilního telefonu. Druhou částí je klientská aplikace, která běží na mobilním zařízení a slouží jako uživatelské rozhraní. Obě aplikace spolu komunikují v rámci stejné lokální sítě.

Serverovou část projektu jsem se rozhodl implementovat v programovacím jazyce Rust, jelikož jsem s tímto jazykem měl již předchozí zkušenosti a považuji jej za programovací jazyk s dobrým výhledem do budoucna. Mobilní aplikace byla vytvořena pomocí frameworku Tauri, jehož backend je rovněž napsán v jazyce Rust. Frontend mobilní aplikace je realizován pomocí technologií HTML a JavaScript, se kterými mám rovněž předchozí zkušenosti.

1 ARCHITEKTURA A POUŽITÉ TECHNOLOGIE

1.1 Klient-server architektura a komunikační model

Aplikace je rozdělena na dvě hlavní části – server a klienta. Serverová část zajišťuje hlavní logiku aplikace, přístup k systémovým prostředkům a zpracování požadavků, zatímco klientská část slouží jako uživatelské rozhraní, prostřednictvím kterého uživatel se systémem komunikuje. Obě části spolu komunikují v rámci stejné lokální sítě.

1.2 WebSocket komunikace

Pro komunikaci mezi klientem a serverem byl zvolen protokol WebSocket. Jedná se o aplikační protokol, který umožňuje navázání trvalého obousměrného spojení mezi klientem a serverem. Na rozdíl od klasické HTTP komunikace není nutné opakovaně vytvářet nové požadavky. Server tak může aktivně zasílat informace klientovi bez nutnosti, aby o ně klient opakovaně žádal. V projektu je WebSocket využit jako hlavní komunikační kanál, přes který jsou přenášeny příkazy z mobilní aplikace do serverové části a zpětně informace o aktuálním stavu systému.

1.3 Použité technologie

Serverová část aplikace je implementována v programovacím jazyce Rust. Tento jazyk byl zvolen především kvůli své bezpečnosti, vysokému výkonu a modernímu přístupu ke správě paměti bez nutnosti garbage collectoru. Pro implementaci síťové komunikace je využit framework Warp, který umožňuje jednoduchou tvorbu HTTP a WebSocket serverů.

Klientská část aplikace je realizována pomocí frameworku Tauri, který umožňuje vytvářet multiplatformní aplikace s využitím webových technologií. Backend Tauri aplikace je rovněž napsán v jazyce Rust, zatímco frontend je tvořen pomocí HTML, CSS a JavaScriptu.

Pro ukládání lokálních dat na straně klienta, například autentizačních tokenů nebo nastavení aplikace, je využito rozhraní LocalStorage.

2 ZPŮSOBY ŘEŠENÍ A POUŽITÉ POSTUPY

2.1 Začátek projektu

Nejdříve jsem vytvořil projekt pro počítač skrze RustRover a poté pomocí npm vytvořil šablonu pro Tauri projekt, přičemž jsem se rozhodl využít šablonu s JavaScriptem. První verze programu byl pouze otevřený port na počítači, na který se po zadání správné IP adresy připojil telefon, jenž pak mohl ovládat zvuk počítače.

2.2 První build pro telefon

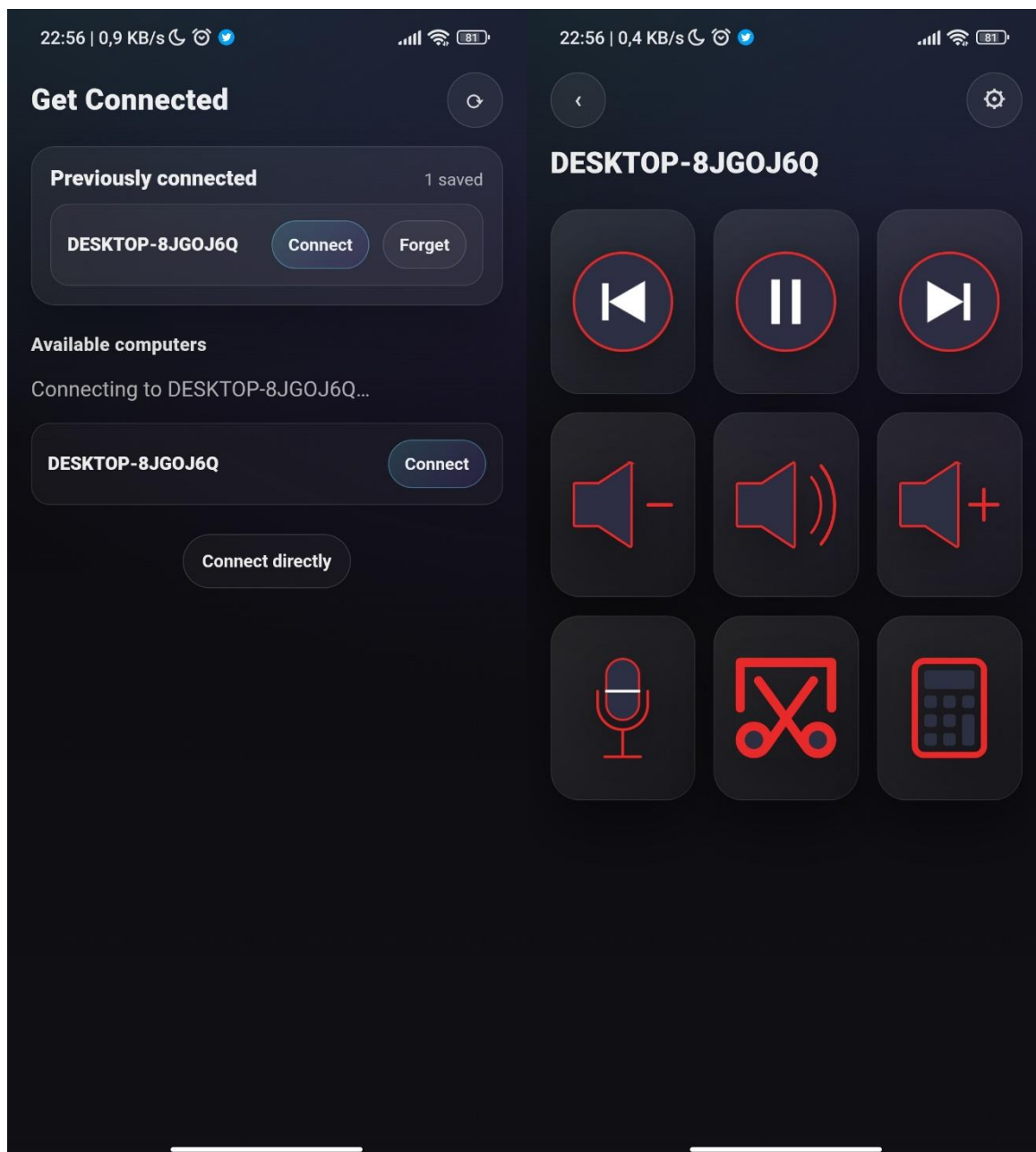
Chtěl jsem přidat možnost skenovat zařízení v okolí, a jelikož jsem do té doby používal pouze emulátor, který se mohl pouze připojit přes lokální adresu 10.0.2.2, tak jsem musel buildnout aplikaci na telefon. Tauri to naštěstí udělalo jednoduché, kdy stačilo pouze na telefonu, který kabelem připojíte k počítači, povolit ladění přes USB a automaticky to začalo instalovat přímo do telefonu. Mohl jsem tak přidat funkci do aplikace, která se umí vyhledat počítače, jenž mají zapnutou viditelnost. V GUI počítače je možné buď zapnout pouze server, takže se na něj dá připojit pouze napřímo, nebo se k tomu dá zapnout i viditelnost, takže se počítač bude zobrazovat v mobilní aplikaci.

2.3 Přidání párování

Jelikož jsem chtěl zabezpečit připojení, tak aby se nemohl jen tak někdo připojit, tak jsem přidal nutnost zadat párovací kód, který se vygeneruje na obrazovce počítače. Později jsem přidal automatickou autentizaci, kdy telefony které se už jednou spárovaly budou pomocí unikátního tokenu automaticky spárovány bez nutnosti znovu zadat párovací kód. Zapamatované telefony je možné odebrat v GUI pomocí tlačítka „Revoke“

2.4 Úprava UI

Jelikož je hlavně důležitý vzhled mobilní aplikace, tak jsem ji upravil, aby to již nebyla pouze bílá stránka se třemi tlačítky a logem pro debuggování. Aplikace má dvě stránky, první ve které se páruje s počítačem a druhá ve které je již spárovaná a ukazuje tlačítka na ovládání počítače.

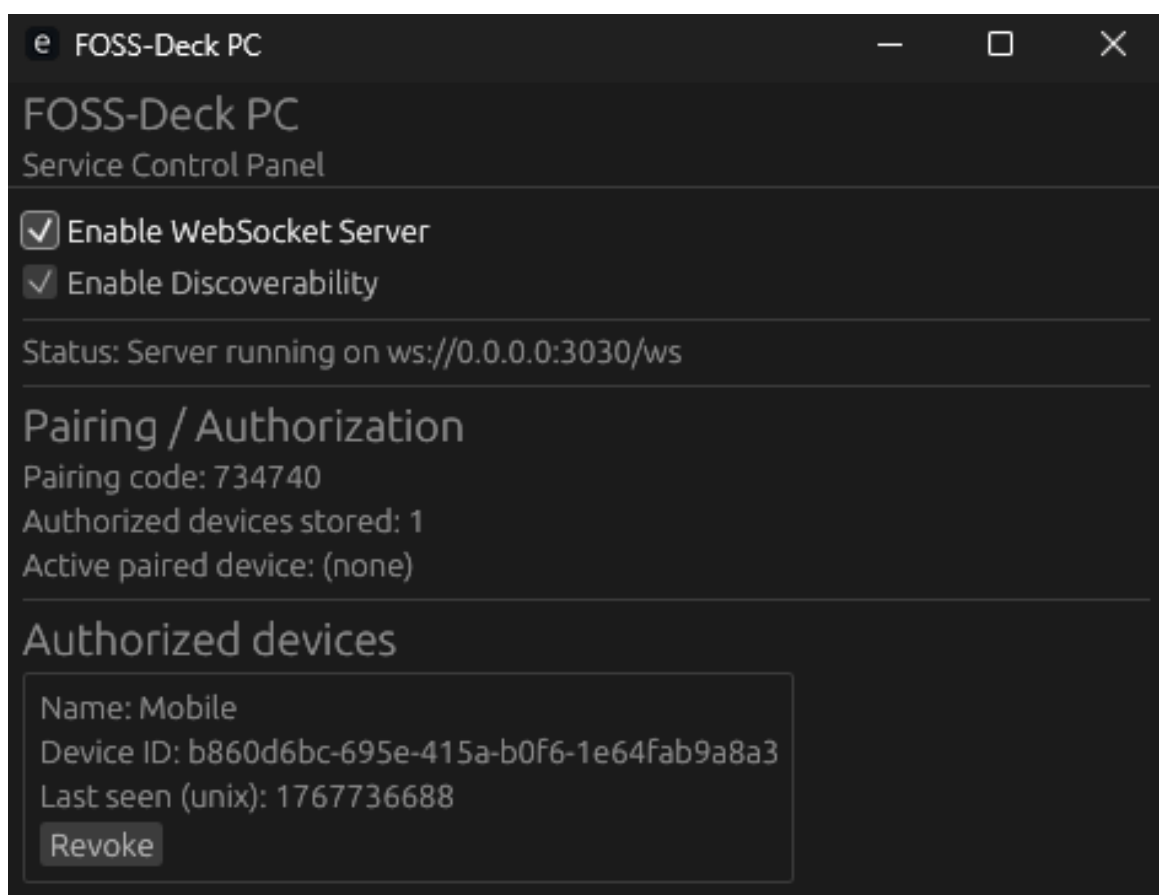


2.5 Přidání dalších funkcí

Doted' bylo možné ovládat pouze hlasitost zvuku, a tak jsem přidal další funkce, které bych osobně používal. Možnost přepínání multimédií, ovládání mikrofonu, screenshot obrazovky a otevření kalkulačky.

2.6 Build aplikací

Na závěr již zbývalo pouze buildnout release verzi aplikací. To bylo u počítačové aplikace jednoduché, stačilo použít `cargo build --release` a ve složce `/target` bylo optimalizované `.exe`. U mobilní aplikace to bylo složitější, pomocí „keytool“ jsem si vytvořil unikátní podpis, jehož cestu jsem poté přidal do tauri configu, který pak veškeré buildy tvořil s již podepsaným `.apk`. Tyto spustitelné programy jsem poté releasnul na Github repositáři tohoto projektu.



3 IMPLEMENTACE JEDNÉ FUNKCE NA PC

3.1 Formát příkazu

Server očekává JSON s polem cmd, které je mapované přes serde na enum příkazů:

```
pub enum WsCommand {  
    ToggleMicMute,  
}
```

Praktický příklad zprávy z klienta:

```
{"cmd": "toggle_mic_mute"}
```

(soubor commands.rs)

3.2 Přijetí zprávy

Ve WebSocket vrstvě se textová zpráva přečte a deserializuje na WsCommand:

```
let reply = match serde_json::from_str::<WsCommand>(text) {  
    Ok(cmd) => { /* ... */ }  
    Err(e) => { /* ... */ }  
};
```

(soubor ws.rs)

3.3 Kontrola autentizace a předání do aplikační logiky

Po úspěšné autentizaci (řešeno v ws.rs) se příkaz předá do funkce, která řeší „device control“ příkazy

```
if !authenticated {  
    json!({"type": "error", "reason": "not_authenticated"})  
} else {  
    { pairing.lock().unwrap().mark_seen(); }  
    match handle_command(cmd) {  
        Ok(v) => v,  
        Err(_) => json!({"type": "error", "reason": "command_failed"})  
    }  
}
```

(soubor ws.rs)

3.4 Implementace ToggleMicMute v handleru příkazů

V `handle_command` se aktuální stav mikrofonu zjistí, invertuje a zapíše zpět. Nakonec server vrátí JSON se stavem, aby si klient mohl UI synchronizovat:

```
WsCommand::ToggleMicMute => {
    let mic_muted = audio::get_mic_mute()?;
    audio::set_mic_mute(!mic_muted)?;
    let (vol, muted) = audio::get_volume_and_mute()?;
    let mic_muted = audio::get_mic_mute()?;
    Ok(json!({"type":"ok","action":"toggle_mic_mute",
            "volume":vol,"muted":muted,"mic_muted":mic_muted}))
}
```

(*soubor commands.rs*)

3.5 Přístup k mikrofonu ve Windows

Na úrovni OS se používá výchozí **capture** zařízení (eCapture) a COM rozhraní `IAudioEndpointVolume`. Mikrofon se ztlumí / povolí přes `SetMute`:

```
fn default_capture_endpoint() -> Result<IMMDevice> {
    let enumerator: IMMDeviceEnumerator =
        CoCreateInstance(&MMDeviceEnumerator, None, CLSCTX_ALL)?;
    Ok(enumerator.GetDefaultAudioEndpoint(eCapture, eConsole)?)
}

rust
Zkopírovat kód
pub fn set_mic_mute(mute: bool) -> Result<()> {
    let ep = mic_endpoint_volume()?;
    ep.SetMute(BOOL::from(mute), &GUID::zeroed())?;
    Ok(())
}

rust
Zkopírovat kód
pub fn get_mic_mute() -> Result<bool> {
    let ep = mic_endpoint_volume()?;
    Ok(ep.GetMute()?.as_bool())
}
```

(*soubor audio.rs*)

4 IMPLEMENTACE JEDNÉ FUNKCE NA MOBILU

4.1 Stav klienta

Klient si drží lokální stav v `state.audio`, včetně stavu mikrofonu:

```
audio: {  
  muted: false,  
  volume: 1.0,  
  playing: true,  
  micMuted: false,  
},
```

(soubor *state.js*)

4.2 Definice akce a „optimistic UI“

Akce `toggle_mic_mute` je definovaná v mapě `ACTIONS`. Důležité:

- `enabled()` povolí tlačítko jen když je klient spárovaný (`state.isPaired`)
- v `run()` se nejdřív **optimisticky** přepne `state.audio.micMuted` a hned se překreslí UI
- potom se odešle příkaz přes WS

```
toggle_mic_mute: {  
  id: "toggle_mic_mute",  
  title: "Mic",  
  icon: () => state.audio.micMuted ? "assets/mic_muted.png" : "as-  
sets/mic.png",  
  enabled: () => state.isPaired,  
  run: () => {  
    state.audio.micMuted = !state.audio.micMuted; // optimistic  
    renderTiles();  
    sendCmd({ cmd: "toggle_mic_mute" });  
  },  
},
```

(soubor *actions.js*)

4.3 Odeslání příkazu na server

Odesílání je centralizované v `sendCmd()`:

```
export function sendCmd(obj) {  
  if (!state.ws || state.ws.readyState !== WebSocket.OPEN) return;  
  state.ws.send(JSON.stringify(obj));  
}
```

(soubor *ws.js*)

4.4 Příjem odpovědí: synchronizace stavu z PC

Klient zpracovává příchozí WS zprávy v `onmessage`. Pro stav je klíčový typ `status`:

```
if (obj.type === "status") {  
  if (typeof obj.muted === "boolean") state.audio.muted = obj.muted;  
  if (typeof obj.volume === "number") state.audio.volume = obj.volume;  
  if (typeof obj.mic_muted === "boolean")  
    state.audio.micMuted = obj.mic_muted;  
  renderTiles();  
  return;  
}
```

(soubor *ws.js*)

To je mechanismus, kterým se „optimistic UI“ případně opraví na reálný stav (pokud by server akci neprovedl nebo měl jiný stav).

4.5 Heartbeat: pravidelné dotazování na stav

Po úspěšném spárování se každých 5 sekund posílá `get_status`, aby se UI udržovalo synchronizované:

```
export function startHeartbeat() {  
  stopHeartbeat();  
  sendCmd({ cmd: "get_status" });  
  state.heartbeatTimer = setInterval(() => {  
    if (state.ws && state.ws.readyState === WebSocket.OPEN && state.isPaired) {  
      sendCmd({ cmd: "get_status" });  
    }  
  }, 5000);  
}
```

ZÁVĚR

Výsledkem práce je funkční aplikace umožňující vzdálené ovládání základních funkcí počítače.

Pro další zlepšení by bylo dobré přidat možnost tvořit vlastní akce, jako např. spouštění vlastních aplikací, či jejich ovládání, vylepšení UI na počítači, vytvoření verze pro linux.

Kód je k nalezení zde: <https://github.com/007king700/FOSS-Deck>

SEZNAM POUŽITÝCH INFORMAČNÍCH ZDROJŮ

- [1] *Tauri*. Online. Dostupné z: <https://v2.tauri.app/>. [cit. 2026-01-06].
- [2] *Stack overflow*. Online. [Stackoverflow.com/questions](https://stackoverflow.com/questions). [cit. 2026-01-06].
- [3] *ChatGPT*. Online. Dostupné z: <https://chatgpt.com/>. [cit. 2026-01-06].