

The background is a dark blue gradient with faint, light blue geometric patterns. On the left side, there are several concentric circles and arcs, some with degree markings ranging from 40 to 260. These markings are arranged in a way that suggests a circular scale or a compass. The overall aesthetic is technical and modern.

TAURI: CROSS-PLATFORM DESKTOP APPLICATIONS WITH RUST AND WEB TECHNOLOGIES

LIEL FRIDMAN

TAURI IN A NUTSHELL

- A library for desktop (and since v2.0 - mobile) application development
- Uses web technologies
 - But supports front end in Rust as well (Dioxus, Leptos, Yew)
 - Backend code is Rust, but third-party binaries can be embedded as [sidecars](#)
- Written in Rust
- Developers can write backend logic in Rust, but don't have to



TAURI

APPS SHOWCASE

I'm not affiliated with any of these. I just thought they serve as cool examples.

- [pgMagic](#) – a PostgreSQL client that supports natural language
- [RustDesk](#) – open-source remote access and support software

TAURI VS ELECTRON (1)

- Rust based vs Node.js based
- A different philosophy of using the browser
 - Tauri uses the [OS-provided WebView](#)
 - Electron bundles Chromium
 - Impacts size

TAURI VS ELECTRON (2)

- Security

- Electron apps can be very secure
- But it's harder to misuse Tauri
 - IPC via message passing
 - Permissions mechanism
 - Therefore, a compromised frontend won't be able to do as much damage
- Caveat: Rust code is **not isolated**

CLI AND DEVELOPMENT SETUP

Two primary tools:

- Create-tauri-app – for scaffolding new Tauri projects
- Tauri CLI – for manual setup and other tasks (installed locally on projects, but can also be installed globally)

A Rust toolchain is necessary, as well as Node.js and your favorite build tool (if you use JS/TS for the frontend) or the .NET equivalent.

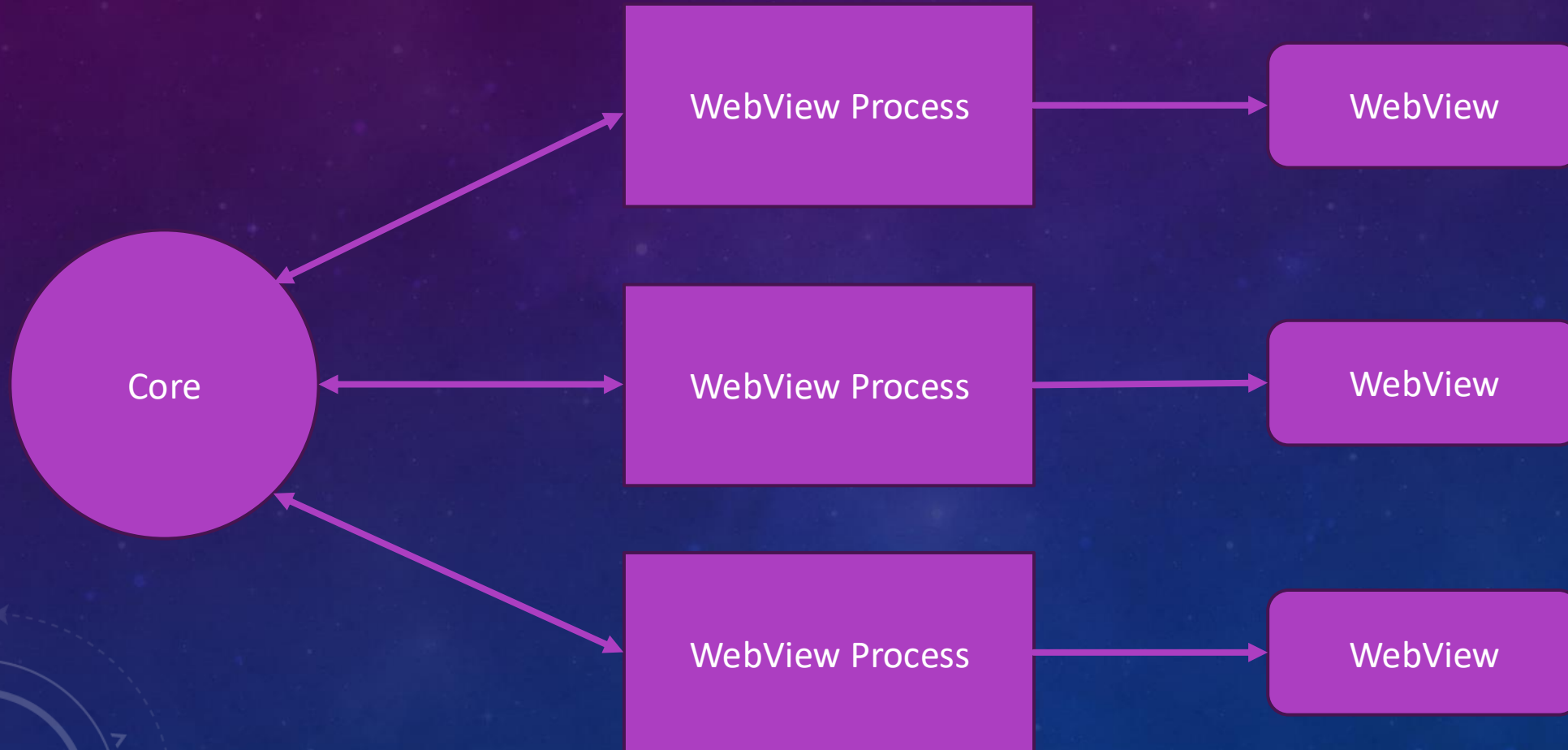
Regarding IDEs, the recommended setup is VS Code, Neovim or JetBrains IDEs.

See the [documentation](#) for more information.

PROJECT STRUCTURE

- Designed to be minimally invasive and allow existing frontend code to work with Tauri
- Rust code and Tauri configuration live in the src-tauri folder
- Create-tauri-app installs the tauri CLI into the project, but also allows separate frontend development

PROCESS MODEL



COMMANDS (FRONTEND -> RUST) (1)

- Every call is a command
- Permissions system (more on that shortly)
- Flexible payload (only needs to implement `Serde::Deserialize`)

COMMANDS (FRONTEND -> RUST) (2)

```
#[tauri::command]
fn login(user: String, password: String) -> Result<String, String> {
    if user == "tauri" && password == "tauri" {
        // resolve
        Ok("logged_in".to_string())
    } else {
        // reject
        Err("invalid credentials".to_string())
    }
}
```

EVENTS AND CHANNELS (1)

- For small events: Events
 - Also ideal for multi consumer and multi producer system
 - Unlike commands, no strong type support. Payloads are always JSON strings
 - No permissions/capabilities
- For low latency: Channels

EVENTS AND CHANNELS (2)

Global events

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

#[tauri::command]
fn download(app: AppHandle, url: String) {
    app.emit("download-started", &url).unwrap();
    for progress in [1, 15, 50, 80, 100] {
        app.emit("download-progress", progress).unwrap();
    }
    app.emit("download-finished", &url).unwrap();
}
```

EVENTS AND CHANNELS (3)

Webview events (specific to one view)

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

#[tauri::command]
fn login(app: AppHandle, user: String, password: String) {
    let authenticated = user == "tauri-apps" && password == "tauri";
    let result = if authenticated { "loggedIn" } else { "invalidCredentials" };
    app.emit_to("login", "login-result", result).unwrap();
}
```

EVENTS AND CHANNELS (4)

Listening to events on the frontend (global events)

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

type DownloadStarted = {
  url: string;
  downloadId: number;
  contentLength: number;
};

listen<DownloadStarted>('download-started', (event) => {
  console.log(
    `downloading ${event.payload.contentLength} bytes from ${event.payload.url}`
  );
});
```

EVENTS AND CHANNELS (5)

Listening to events on the frontend (webview-specific events)

```
import { getCurrentWebviewWindow } from '@tauri-apps/api/webviewWindow';

const appWebview = getCurrentWebviewWindow();
appWebview.listen<string>('logged-in', (event) => {
  localStorage.setItem('session-token', event.payload);
});
```

Unlisten

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

const unlisten = await listen('download-started', (event) => {});
unlisten();
```

EVENTS AND CHANNELS (6)

Channels (fast, ordered)

```
use tauri::{AppHandle, ipc::Channel};
use serde::Serialize;

#[derive(Clone, Serialize)]
#[serde(rename_all = "camelCase", tag = "event", content = "data")]
enum DownloadEvent<'a> {
    #[serde(rename_all = "camelCase")]
    Started {
        url: &'a str,
        download_id: usize,
        content_length: usize,
    },
    #[serde(rename_all = "camelCase")]
    Progress {
        download_id: usize,
        chunk_length: usize,
    },
    #[serde(rename_all = "camelCase")]
    Finished {
        download_id: usize,
    },
}
```

EVENTS AND CHANNELS (7)

Channels (fast, ordered)

```
#[tauri::command]
fn download(app: AppHandle, url: String, on_event: Channel<DownloadEvent>) {
    let content_length = 1000;
    let download_id = 1;

    on_event.send(DownloadEvent::Started {
        url: &url,
        download_id,
        content_length,
    }).unwrap();

    for chunk_length in [15, 150, 35, 500, 300] {
        on_event.send(DownloadEvent::Progress {
            download_id,
            chunk_length,
        }).unwrap();
    }

    on_event.send(DownloadEvent::Finished { download_id }).unwrap();
}
```

EVENTS AND CHANNELS (9)

Channels (fast, ordered) – Frontend Side

```
import { invoke, Channel } from '@tauri-apps/api/core';

type DownloadEvent =
  // redacted for simplicity
  };

const onEvent = new Channel<DownloadEvent>();
onEvent.onmessage = (message) => {
  console.log(`got download event ${message.event}`);
};

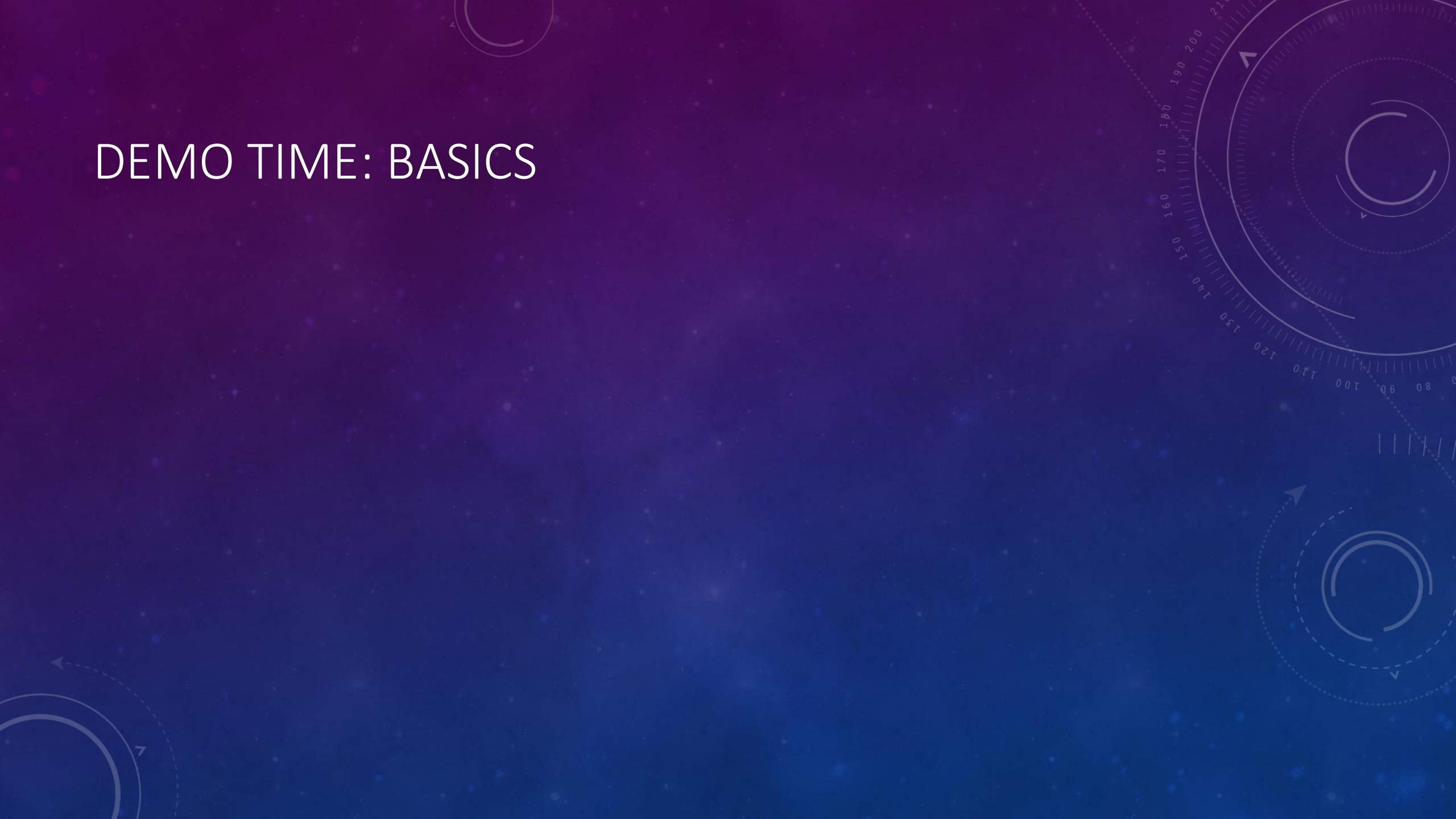
await invoke('download', {
  url: 'https://raw.githubusercontent.com/tauri-apps/tauri/dev/crates/tauri-schema-generator/schemas/config.schema.json',
  onEvent,
});
```

LISTENING TO EVENTS IN RUST

```
use tauri::Listener;

#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
    tauri::Builder::default()
        .setup(|app| {
            app.listen("download-started", |event| {
                if let Ok(payload) = serde_json::from_str::<DownloadStarted>(&event.payload()) {
                    println!("downloading {}", payload.url);
                }
            });
            Ok(())
        })
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

DEMO TIME: BASICS



PERMISSIONS (1)

- IPC is the only way of the UI to communicate with the application core
 - Done via message passing, and each message is also known as a command
 - Permissions give **explicit privileges** to commands
 - Can be scoped. For example, filesystem permissions can be restricted to the home folder
 - Example in the next slide

PERMISSIONS (2)

```
[[permission]]
identifier = "my-identifier"
description = "This describes the impact and more."
commands.allow = [
    "read_file"
]
```

```
[[scope.allow]]
my-scope = "$HOME/*"
```

```
[[scope.deny]]
my-scope = "$HOME/secret"
```

Allowed variables

CAPABILITIES

- Build upon the permissions system
- A set of permissions mapped to application windows by their label
 - Label $\neq$ title
 - Still need to carefully manage window creation permissions
- Can also be platform-specific
- Can be defined in JSON or in TOML

```
{  
  "$schema": "../gen/schemas/desktop-schema.json",  
  "identifier": "desktop-capability",  
  "windows": ["main"],  
  "platforms": ["linux", "macOS", "windows"],  
  "permissions": ["global-shortcut:allow-register"]  
}
```

IPC: BROWNFIELD

- Default pattern
- No sanitization layer, but still limited ways of misuse

IPC: ISOLATION

- Protects against development threats
- Not enabled by default
- Introduces some overhead
- Isolation application runs separately

PLUGINS

- Provide out-of-the-box functionality
- Can be used directly in JS/TS, no need to write custom Rust code
 - but some can be also used directly from Rust
- Examples:
 - [Log](#) – configurable logging
 - [Deep-link](#) – allows to set the app as the default handler for a URL
 - [Dialog](#) – native system dialogs along with message dialogs
- [List of all official plugins](#)
- [Awesome-Tauri](#)

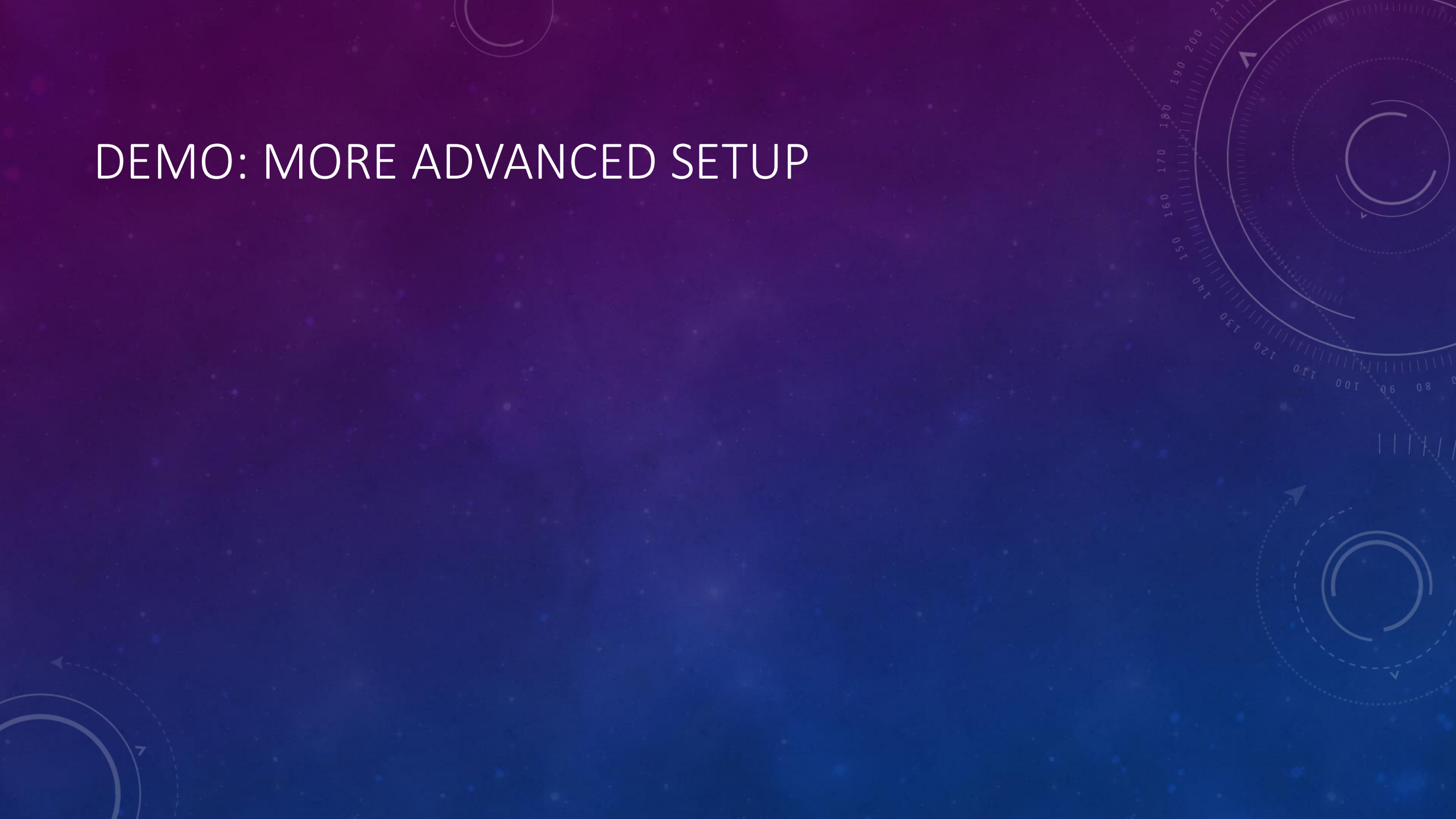
WRAPPING EXISTING PROJECTS IN TAURI

Relatively straightforward: using the tauri CLI, init a project and use `..` as the location of web assets

DISTRIBUTION AND PACKAGING

- Tauri supports packaging out of the box
 - Caveat: Apart from specific (experimental) use cases, you'll need to run the build command on each target platform
 - Run the bundle command in the Tauri CLI
- See the [documentation](#) for more info and caveats

DEMO: MORE ADVANCED SETUP



ADDITIONAL LINKS

- [Official documentation](#)
- [Awesome Tauri](#)
- [GitHub](#)
- [Discord](#)

THANK YOU

Personal contact info:

- [GitHub](#)
- Email: lielft <at> gmail
- [LinkedIn](#)