Tauri Plugins

Plugins allow you to hook into the Tauri application lifecycle and introduce new commands.

Using a Plugin

To use a plugin, just pass the plugin instance to the App's `plugin` method:

```
fn main() {
  tauri::Builder::default()
    .plugin(my_awesome_plugin::init())
    .run(tauri::generate_context!())
    .expect("failed to run app");
}
```

Writing a Plugin

Plugins are reusable extensions to the Tauri API that solve common problems. They are also a very convenient way to structure your code base!

If you intend to share your plugin with others, we provide a ready-made template! With Tauri's CLI installed just run:

npm Yarn pnpm Cargo

```
$ cargo tauri plugin init --name awesome
```

API package

By default consumers of your plugin can call provided commands like this:

```
import { invoke } from '@tauri-apps/api'
invoke('plugin:awesome|do_something')
```

where `awesome` will be replaced by your plugin name.

This isn't very convenient, however, so it's common for plugins to provide a so-called *API package*, a JavaScript package that provides convenient access to your commands.

> An example of this is the tauri-plugin-store, which provides a convenient class structure for accessing a store. You can scaffold a tauri plugin with an attached javascript API package like this:

npm Yarn pnpm Cargo

```
$ cargo tauri plugin init --name awesome --api
```

Writing a Plugin

Using the `tauri::plugin::Builder` you can define plugins similar to how you define your app:

```rust
use tauri::{
  plugin::{Builder, TauriPlugin},
  Runtime,
};

// the plugin custom command handlers if you choose to extend the API:

#[tauri::command]
// this will be accessible with `invoke('plugin:awesome|initialize')`.
// where `awesome` is the plugin name.
fn initialize() {}

#[tauri::command]
// this will be accessible with `invoke('plugin:awesome|do_something')`.
fn do_something() {}

pub fn init<R: Runtime>() -> TauriPlugin<R> {
  Builder::new("awesome")
    .invoke_handler(tauri::generate_handler![initialize, do_something])
    .build()
}
```

Plugins can set up and maintain state, just like your app can:

```rust
use tauri::{
  plugin::{Builder, TauriPlugin},
  AppHandle, Manager, Runtime, State,
};

#[derive(Default)]
struct MyState {}

#[tauri::command]
// this will be accessible with `invoke('plugin:awesome|do_something')`.
fn do_something<R: Runtime>(_app: AppHandle<R>, state: State<'_, MyState>) {
  // you can access `MyState` here!
}

pub fn init<R: Runtime>() -> TauriPlugin<R> {
  Builder::new("awesome")
    .invoke_handler(tauri::generate_handler![do_something])
    .setup(|app_handle| {
      // setup plugin specific state here
      app_handle.manage(MyState::default());
      Ok(())
    })
    .build()
}
```

Conventions

- The crate exports an `init` method to create the plugin.
- Plugins should have a clear name with `tauri-plugin-` prefix.
- Include `tauri-plugin` keyword in `Cargo.toml` / `package.json`.
- Document your plugin in English.
- Add an example app showcasing your plugin.

Advanced

Instead of relying on the `tauri::plugin::TauriPlugin` struct returned by `tauri::plugin::Builder::build`, you can implement the `tauri::plugin::Plugin` yourself. This allows you to have full control over the associated data.

Note that each function on the `Plugin` trait is optional, except the `name` function.

```rust
use tauri::{plugin::{Plugin, Result as PluginResult}, Runtime, PageLoadPayload, Window, Invoke,
AppHandle};
```

```rust
struct MyAwesomePlugin<R: Runtime> {
  invoke_handler: Box<dyn Fn(Invoke<R>) + Send + Sync>,
  // plugin state, configuration fields
}

// the plugin custom command handlers if you choose to extend the API.
#[tauri::command]
// this will be accessible with `invoke('plugin:awesome|initialize')`.
// where `awesome` is the plugin name.
fn initialize() {}

#[tauri::command]
// this will be accessible with `invoke('plugin:awesome|do_something')`.
fn do_something() {}

impl<R: Runtime> MyAwesomePlugin<R> {
  // you can add configuration fields here,
  // see https://doc.rust-lang.org/1.0.0/style/ownership/builders.html
  pub fn new() -> Self {
    Self {
      invoke_handler: Box::new(tauri::generate_handler![initialize, do_something]),
    }
  }
}

impl<R: Runtime> Plugin<R> for MyAwesomePlugin<R> {
  /// The plugin name. Must be defined and used on the `invoke` calls.
  fn name(&self) -> &'static str {
    "awesome"
  }

  /// The JS script to evaluate on initialization.
  /// Useful when your plugin is accessible through `window`
  /// or needs to perform a JS task on app initialization
  /// e.g. "window.awesomePlugin = { ... the plugin interface }"
  fn initialization_script(&self) -> Option<String> {
    None
  }

  /// initialize plugin with the config provided on `tauri.conf.json > plugins >
$yourPluginName` or the default value.
  fn initialize(&mut self, app: &AppHandle<R>, config: serde_json::Value) -> PluginResult<()> {
    Ok(())
  }

  /// Callback invoked when the Window is created.
  fn created(&mut self, window: Window<R>) {}

  /// Callback invoked when the webview performs navigation.
```

```
  fn on_page_load(&mut self, window: Window<R>, payload: PageLoadPayload) {}

  /// Extend the invoke handler.
  fn extend_api(&mut self, message: Invoke<R>) {
    (self.invoke_handler)(message)
  }
}
```

Edit this page

*Last updated on **Sep 28, 2023***