System Tray

Native application system tray.

Setup

Configure the `systemTray` object on `tauri.conf.json`:

```json
{
  "tauri": {
    "systemTray": {
      "iconPath": "icons/icon.png",
      "iconAsTemplate": true
    }
  }
}
```

The `iconAsTemplate` is a boolean value that determines whether the image represents a Template Image on macOS.

Linux Setup

On Linux, you need to install one of `libayatana-appindicator` or `libappindicator3` packages. Tauri determines which package to use at runtime, with `libayatana` being the preferred one if both are installed.

By default, the Debian package (`.deb` file) will add a dependency on `libayatana-appindicator3-1`. To create a Debian package targetting `libappindicator3`, set the `TAURI_TRAY` environment variable to `libappindicator3`.

The AppImage bundle automatically embeds the installed tray library, and you can also use the `TAURI_TRAY` environment variable to manually select it.

INFO

`libappindicator3` is unmaintained and does not exist on some distros like `debian11`, but `libayatana-appindicator` does not exist on older releases.

Creating a system tray

To create a native system tray, import the `SystemTray` type:

```rust
use tauri::SystemTray;
```

Initialize a new tray instance:

```rust
let tray = SystemTray::new();
```

Configuring a system tray context menu

Optionally you can add a context menu that is visible when the tray icon is right-clicked. Import the `SystemTrayMenu`, `SystemTrayMenuItem` and `CustomMenuItem` types:

```rust
use tauri::{CustomMenuItem, SystemTrayMenu, SystemTrayMenuItem};
```

Create the `SystemTrayMenu`:

```rust
// here `"quit".to_string()` defines the menu item id, and the second parameter is the menu
item label.
let quit = CustomMenuItem::new("quit".to_string(), "Quit");
let hide = CustomMenuItem::new("hide".to_string(), "Hide");
let tray_menu = SystemTrayMenu::new()
  .add_item(quit)
  .add_native_item(SystemTrayMenuItem::Separator)
  .add_item(hide);
```

Add the tray menu to the `SystemTray` instance:

```rust
let tray = SystemTray::new().with_menu(tray_menu);
```

Configure the app system tray

The created `SystemTray` instance can be set using the `system_tray` API on the `tauri::Builder` struct:

```rust
use tauri::{CustomMenuItem, SystemTray, SystemTrayMenu};
```

```
fn main() {
  let tray_menu = SystemTrayMenu::new(); // insert the menu items here
  let system_tray = SystemTray::new()
    .with_menu(tray_menu);
  tauri::Builder::default()
    .system_tray(system_tray)
    .run(tauri::generate_context!())
    .expect("error while running tauri application");
}
```

Listening to system tray events

Each `CustomMenuItem` triggers an event when clicked. Also, Tauri emits tray icon click events. Use the `on_system_tray_event` API to handle them:

```
use tauri::{CustomMenuItem, SystemTray, SystemTrayMenu, SystemTrayEvent};
use tauri::Manager;

fn main() {
  let tray_menu = SystemTrayMenu::new(); // insert the menu items here
  tauri::Builder::default()
    .system_tray(SystemTray::new().with_menu(tray_menu))
    .on_system_tray_event(|app, event| match event {
      SystemTrayEvent::LeftClick {
        position: _,
        size: _,
        ..
      } => {
        println!("system tray received a left click");
      }
      SystemTrayEvent::RightClick {
        position: _,
        size: _,
        ..
      } => {
        println!("system tray received a right click");
      }
      SystemTrayEvent::DoubleClick {
        position: _,
        size: _,
        ..
      } => {
        println!("system tray received a double click");
      }
      SystemTrayEvent::MenuItemClick { id, .. } => {
        match id.as_str() {
          "quit" => {
            std::process::exit(0);
          }
          "hide" => {
            let window = app.get_window("main").unwrap();
            window.hide().unwrap();
          }
          _ => {}
        }
      }
      _ => {}
    })
    .run(tauri::generate_context!())
    .expect("error while running tauri application");
}
```

Updating system tray

The `AppHandle` struct has a `tray_handle` method, which returns a handle to the system tray allowing updating tray icon and context menu items:

Updating context menu items

```
use tauri::{CustomMenuItem, SystemTray, SystemTrayMenu, SystemTrayEvent};
use tauri::Manager;

fn main() {
  let tray_menu = SystemTrayMenu::new(); // insert the menu items here
  tauri::Builder::default()
    .system_tray(SystemTray::new().with_menu(tray_menu))
    .on_system_tray_event(|app, event| match event {
      SystemTrayEvent::MenuItemClick { id, .. } => {
        // get a handle to the clicked menu item
        // note that `tray_handle` can be called anywhere,
        // just get an `AppHandle` instance with `app.handle()` on the setup hook
        // and move it to another function or thread
        let item_handle = app.tray_handle().get_item(&id);
        match id.as_str() {
          "hide" => {
            let window = app.get_window("main").unwrap();
            window.hide().unwrap();
            // you can also `set_selected`, `set_enabled` and `set_native_image` (macOS only).
            item_handle.set_title("Show").unwrap();
          }
          _ => {}
        }
      }
      _ => {}
    })
    .run(tauri::generate_context!())
    .expect("error while running tauri application");
}
```

Updating tray icon

Note that you need to add `icon-ico` or `icon-png` feature flag to the tauri dependency in your Cargo.toml to be able to use `Icon::Raw`

```
app.tray_handle().set_icon(tauri::Icon::Raw(include_bytes!
("../path/to/myicon.ico").to_vec())).unwrap();
```

Preventing the App from Closing

By default, Tauri closes the application when the last window is closed. You can simply call `api.prevent_close()` to prevent this.

Depending on your needs you can use one of the two following options:

Keep the Backend Running in the Background

If your backend should run in the background, you can call `api.prevent_exit()` like so:

```
tauri::Builder::default()
  .build(tauri::generate_context!())
  .expect("error while building tauri application")
  .run(|_app_handle, event| match event {
    tauri::RunEvent::ExitRequested { api, .. } => {
      api.prevent_exit();
    }
    _ => {}
  });
```

Keep the Frontend Running in the Background

If you need to keep the frontend running in the background, this can be achieved like this:

```
tauri::Builder::default().on_window_event(|event| match event.event() {
  tauri::WindowEvent::CloseRequested { api, .. } => {
    event.window().hide().unwrap();
    api.prevent_close();
  }
```

```
    _ => {}
  })
  .run(tauri::generate_context!())
  .expect("error while running tauri application");
```

Edit this page

*Last updated on **Jun 30, 2023***