

TAURI Docs - English



Table of contents

• .TEMPLATES - Cli	??
• API - Cli	??
• API - Config	11
• API JS - README	41
• API JS - App	42
• API JS - Cli	45
• API JS - Clipboard	48
• API JS - Dialog	50
• API JS - Event	58
• API JS - Fs	64
• API JS - GlobalShortcut	77
• API JS - Http	81
• API JS - Mocks	96
• API JS - Notification	100
• API JS - Os	103
• API JS - Path	107
• API JS - Process	123
• API JS - Shell	125
• API JS - Tauri	142
• API JS - Updater	145
• API JS - Window	149
• GUIDES BUILDING - README	221
• GUIDES BUILDING - _tauri-build	222
• GUIDES BUILDING - App-size	223
• GUIDES BUILDING - Cross-platform	229
• GUIDES BUILDING - Linux	233
• GUIDES BUILDING - Macos	235
• GUIDES BUILDING - Resources	237
• GUIDES BUILDING - Sidecar	239
• GUIDES BUILDING - Windows	242
• GUIDES DEBUGGING - Application	249
• GUIDES DEBUGGING - Vs-code	252
• GUIDES DEVELOPMENT - Development-cycle	254
• GUIDES DEVELOPMENT - Updating-dependencies	255

• GUIDES DISTRIBUTION - Publishing	257
• GUIDES DISTRIBUTION - Sign-linux	258
• GUIDES DISTRIBUTION - Sign-macos	259
• GUIDES DISTRIBUTION - Sign-windows	263
• GUIDES DISTRIBUTION - Updater	267
• GUIDES - Faq	274
• GUIDES FEATURES - README	276
• GUIDES FEATURES - Cli	277
• GUIDES FEATURES - Command	280
• GUIDES FEATURES - Events	285
• GUIDES FEATURES - Icons	288
• GUIDES FEATURES - Menu	290
• GUIDES FEATURES - Multiwindow	293
• GUIDES FEATURES - Plugin	296
• GUIDES FEATURES - Splashscreen	300
• GUIDES FEATURES - System-tray	302
• GUIDES FEATURES - Window-customization	306
• GUIDES GETTING-STARTED - Prerequisites	308
• GUIDES TESTING - Mocking	314
• GUIDES TESTING/WEBDRIVER - Ci	317
• GUIDES TESTING/WEBDRIVER/EXAMPLE - Selenium	319
• GUIDES TESTING/WEBDRIVER/EXAMPLE - Setup	324
• GUIDES TESTING/WEBDRIVER/EXAMPLE - Webdriverio	328
• GUIDES TESTING/WEBDRIVER - Introduction	332
• REFERENCES ARCHITECTURE/INTER-PROCESS-COMMUNICATION - Brownfield	333
• REFERENCES ARCHITECTURE/INTER-PROCESS-COMMUNICATION - Isolation	334
• REFERENCES ARCHITECTURE/INTER-PROCESS-COMMUNICATION - Readme	338
• REFERENCES ARCHITECTURE - Process-model	340
• REFERENCES ARCHITECTURE - Readme	342
• REFERENCES ARCHITECTURE/RECIPES - About-recipes	??
• REFERENCES ARCHITECTURE/RECIPES - Bridge	347
• REFERENCES ARCHITECTURE/RECIPES - Cloubridge	350
• REFERENCES ARCHITECTURE/RECIPES - Cloudish	352
• REFERENCES ARCHITECTURE/RECIPES - Glui	354
• REFERENCES ARCHITECTURE/RECIPES - Hermit	356
• REFERENCES ARCHITECTURE/RECIPES - Lockdown	357

• REFERENCES ARCHITECTURE/RECIPES - Multiwin	359
• REFERENCES ARCHITECTURE - Security	361
• REFERENCES - Benchmarks	365
• REFERENCES - Configuration-files	367
• REFERENCES - Security	370
• REFERENCES - Webview-versions	373

```
import Command from '@theme/Command'
```

info

```
{info}
```

It shows a concise list of information about the environment, Rust, Node.js and their versions as well as some relevant configurations.

info This command is pretty helpful when you need to have a quick overview of your application. When requesting some help, it can be useful that you share this report with us.

init

```
{init}
```

plugin init

```
{plugin init}
```

dev

```
{dev}
```

This command will open the WebView in development mode. It makes use of the `build.devPath` property from your `src-tauri/tauri.conf.json` file.

If you have entered a command to the `build.beforeDevCommand` property, this one will be executed before the `dev` command.

[See more about the configuration.](#)

caution Troubleshooting If you're not using `build.beforeDevCommand`, make sure your `build.devPath` is correct and, if using a development server, that it's started before using this command.

build

```
{build}
```

This command will bundle your application, either in production mode or debug mode if you used the `--debug` flag. It makes use of the `build.distDir` property from your `src-tauri/tauri.conf.json` file.

If you have entered a command to the `build.beforeBuildCommand` property, this one will be executed before the `build` command.

[See more about the configuration.](#)

icon

{icon}

[Tauri Icon Guide](#)

version

Description

Returns the current version of tauri

This command will show the current version of Tauri.

CLI usage

See more about the usage through this [complete guide](#).

[Go to TOC](#)

```
import Command from '@theme/Command'
```

info

```
Usage: cargo-tauri info [OPTIONS]
```

Options:

- v, --verbose... Enables verbose logging
- h, --help Print help information
- V, --version Print version information

It shows a concise list of information about the environment, Rust, Node.js and their versions as well as some relevant configurations.

info This command is pretty helpful when you need to have a quick overview of your application. When requesting some help, it can be useful that you share this report with us.

init

```
Usage: cargo-tauri init [OPTIONS]
```

Options:

- ci Skip prompting for values
- v, --verbose... Enables verbose logging
- f, --force Force init to overwrite the src-tauri folder
- l, --log Enables logging
- d, --directory <DIRECTORY> Set target directory for init [default: /home/runner/work/tauri-docs/tauri-docs]
- t, --tauri-path <TAURI_PATH> Path of the Tauri project to use (relative to the cwd)
- A, --app-name <APP_NAME> Name of your Tauri application
- W, --window-title <WINDOW_TITLE> Window title of your Tauri application
- D, --dist-dir <DIST_DIR> Web assets location, relative to <project-dir>/src-tauri
- P, --dev-path <DEV_PATH> Url of your dev server
- before-dev-command <BEFORE_DEV_COMMAND> A shell command to run before `tauri dev` kicks in
- before-build-command <BEFORE_BUILD_COMMAND> A shell command to run before `tauri build` kicks in
- h, --help Print help information
- V, --version Print version information

plugin init

Usage: cargo-tauri plugin init [OPTIONS] --name <PLUGIN_NAME>

Options:

-n, --name <PLUGIN_NAME>	Name of your Tauri plugin
-v, --verbose...	Enables verbose logging
--api	Initializes a Tauri plugin with TypeScript API
-d, --directory <DIRECTORY>	Set target directory for init [default: /home/runner/work/tauri-docs/tauri-docs]
-t, --tauri-path <TAURI_PATH>	Path of the Tauri project to use (relative to the cwd)
-a, --author <AUTHOR>	Author name
-h, --help	Print help information
-V, --version	Print version information

dev

Usage: cargo-tauri dev [OPTIONS] [ARGS]...

Arguments:

[ARGS]... Command line arguments passed to the runner. Arguments after `--` are passed to the application

Options:

-r, --runner <RUNNER>	Binary to use to run the application
-v, --verbose...	Enables verbose logging
-t, --target <TARGET>	Target triple to build against
-f, --features [<FEATURES>...]	List of cargo features to activate
-e, --exit-on-panic	Exit on panic
-c, --config <CONFIG>	JSON string or path to JSON file to merge with tauri.conf.json
--release	Run the code in release mode
--no-watch	Disable the file watcher
-h, --help	Print help information
-V, --version	Print version information

This command will open the WebView in development mode. It makes use of the `build.devPath` property from your `src-tauri/tauri.conf.json` file.

If you have entered a command to the `build.beforeDevCommand` property, this one will be executed before the `dev` command.

[See more about the configuration.](#)

caution Troubleshooting If you're not using `build.beforeDevCommand`, make sure your `build.devPath` is correct and, if using a development server, that it's started before using this command.

build

Usage: cargo-tauri build [OPTIONS] [ARGS]...

Arguments:

[ARGS]...

Command line arguments passed to the runner

```

Options:
  -r, --runner <RUNNER>
    Binary to use to build the application, defaults to `cargo`

  -v, --verbose...
    Enables verbose logging

  -d, --debug
    Builds with the debug flag

  -t, --target <TARGET>
    Target triple to build against.

    It must be one of the values outputted by `rustc --print target-list` or `universal-apple-darwin` for an universal macOS application.

    Note that compiling an universal macOS application requires both `aarch64-apple-darwin` and `x86_64-apple-darwin` targets to be installed.

  -f, --features [<FEATURES>...]
    Space or comma separated list of features to activate

  -b, --bundles [<BUNDLES>...]
    Space or comma separated list of bundles to package.

    Each bundle must be one of `deb`, `appimage`, `msi`, `app` or `dmg` on MacOS and `updater` on all platforms. If `none` is specified, the bundler will be skipped.

    Note that the `updater` bundle is not automatically added so you must specify it if the updater is enabled.

  -c, --config <CONFIG>
    JSON string or path to JSON file to merge with tauri.conf.json

  -h, --help
    Print help information (use `-h` for a summary)

  -V, --version
    Print version information

```

This command will bundle your application, either in production mode or debug mode if you used the `--debug` flag. It makes use of the `build.distDir` property from your `src-tauri/tauri.conf.json` file.

If you have entered a command to the `build.beforeBuildCommand` property, this one will be executed before the `build` command.

[See more about the configuration.](#)

icon

Usage: cargo-tauri icon [OPTIONS] [INPUT]

Arguments:
 [INPUT] Path to the source icon (png, 1240x1240px with transparency) [default:

```
./app-icon.png]  
Options:  
-o, --output <OUTPUT> Output directory. Default: 'icons' directory next to the  
tauri.conf.json file  
-v, --verbose... Enables verbose logging  
-h, --help Print help information  
-V, --version Print version information
```

[Tauri Icon Guide](#)

version

```
Description  
Returns the current version of tauri
```

This command will show the current version of Tauri.

CLI usage

See more about the usage through this [complete guide](#).

[Go to TOC](#)

Configuration

The Tauri configuration object. It is read from a file where you can define your frontend assets, configure the bundler, enable the app updater, define a system tray, enable APIs via the allowlist and more.

The configuration file is generated by the `tauri init` command that lives in your Tauri application source directory (`src-tauri`).

Once generated, you may modify it at will to customize your Tauri application.

File Formats

By default, the configuration is defined as a JSON file named `tauri.conf.json`.

Tauri also supports JSON5 and TOML files via the `config-json5` and `config-toml` Cargo features, respectively. The JSON5 file name must be either `tauri.conf.json` or `tauri.conf.json5`. The TOML file name is `Tauri.toml`.

Platform-Specific Configuration

In addition to the default configuration file, Tauri can read a platform-specific configuration from `tauri.linux.conf.json`, `tauri.windows.conf.json`, and `tauri.macos.conf.json` (or `Tauri.linux.toml`, `Tauri.windows.toml` and `Tauri.macos.toml` if the `Tauri.toml` format is used), which gets merged with the main configuration object.

Configuration Structure

The configuration is composed of the following objects:

- `package` : Package settings
- `tauri` : The Tauri config
- `build` : The build configuration
- `plugins` : The plugins config

Example

```
{
  build: {
    beforeBuildCommand: ,
    beforeDevCommand: ,
    devPath: ../dist,
    distDir: ../dist
  },
  package: {
    productName: tauri-app,
    version: 0.1.0
  },
  tauri: {
```

```

allowlist: {
  all: true
},
bundle: {},
security: {
  csp: null
},
updater: {
  active: false
},
windows: [
  {
    fullscreen: false,
    height: 600,
    resizable: true,
    title: Tauri App,
    width: 800
  }
]
}
}

```

PackageConfig

The package configuration.

Type: `object`

Name	Type	Default	Description
<code>product-</code> Name	<code>string</code> ? <code>null</code>		App name.
<code>version</code>	<code>string</code> ? <code>null</code>		App version. It is a semver version number or a path to a <code>package.json</code> file containing the <code>version</code> field.

TauriConfig

The Tauri configuration object.

Type: `object`

Name	Type	Default	Description
<code>pattern</code>	<code>PatternKind</code>	<code>view</code>	The pattern to use.
<code>windows</code>	<code>WindowConfig[]</code>	<code>[]</code>	The windows configuration.
<code>cli</code>	<code>CliConfig</code> ?	<code>view</code>	The CLI configuration.
<code>bundle</code>	<code>BundleConfig</code>	<code>view</code>	The bundler configuration.
<code>allowlist</code>	<code>AllowlistConfig</code>	<code>view</code>	The allowlist configuration.
<code>security</code>	<code>SecurityConfig</code>	<code>view</code>	Security configuration.
<code>updater</code>	<code>UpdaterConfig</code>	<code>view</code>	The updater configuration.

Name	Type	Default	Description
systemTray	SystemTrayConfig ? view		Configuration for app system tray.
macOSPrivateApi	boolean	false	MacOS private API configuration. Enables the transparent background API and sets the <code>fullScreenEnabled</code> preference to <code>true</code> .

PatternKind

The application pattern.

Can be any **ONE** of the following types:

- `{ use: brownfield }`: Brownfield pattern.
- `{ use: isolation, options: { dir: string } }`: Isolation pattern. Recommended for security purposes.

WindowConfig

The window configuration object.

Type: `object`

Name	Type	Default	Description
label	string	null	The window identifier. It must be alphanumeric.
url	WindowUrl	view	The window webview URL.
userAgent	string ?	null	The user agent for the webview
fileDropEnabled	boolean	true	Whether the file drop is enabled or not on the webview. By default it is enabled.
center	boolean	false	Disabling it is required to use drag and drop on the front-end on Windows.
x	number ? (format: double)	null	The horizontal position of the window's top left corner
y	number ? (format: double)	null	The vertical position of the window's top left corner
width	number (format: double)	800	The window width.
height	number (format: double)	600	The window height.
minWidth	number ? (format: double)	null	The min window width.

Name	Type	Default	Description
minHeight	number ? (format: double)	null	The min window height.
maxWidth	number ? (format: double)	null	The max window width.
maxHeight	number ? (format: double)	null	The max window height.
resizable	boolean	true	Whether the window is resizable or not.
title	string	null	The window title.
fullscreen	boolean	false	Whether the window starts as fullscreen or not.
focus	boolean	true	Whether the window will be initially focused or not.
			Whether the window is transparent or not.
transparent	boolean	false	<p>Note that on <code>macOS</code> this requires the <code>macos-private-api</code> feature flag, enabled under <code>tauri > macOSPrivateApi</code>.</p> <p>WARNING: Using private APIs on <code>macOS</code> prevents your application from being accepted to the <code>App Store</code>.</p>

```

| maximized
| boolean | false | Whether the window is maximized or not. |
| visible
| boolean | true | Whether the window is visible or not. |
| decorations
| boolean | true | Whether the window should have borders and bars. |
| alwaysOnTop
| boolean | false | Whether the window should always be on top of other windows. |
| skipTaskbar
| boolean | false | If true, hides the window icon from the taskbar on Windows and Linux. |
| theme
| Theme ? | view | The initial window theme. Defaults to the system theme. Only implemented on Windows and macOS 10.14+. |
| titleBarStyle
| TitleBarStyle | view | The style of the macOS title bar. |
| hiddenTitle
| boolean | false | If true, sets the window title to be hidden on macOS. |
| acceptFirstMouse
| boolean | false | Whether clicking an inactive window also clicks through to the webview on macOS. |
| tabbingIdentifier

```

| `string` ? | `null` | Defines the window [tabbing identifier] for macOS.

Windows with matching tabbing identifiers will be grouped together. If the tabbing identifier is not set, automatic tabbing will be disabled.

[tabbing identifier]: <https://developer.apple.com/documentation/appkit/nswindow/1644704-tabbingidentifier> |

WindowUrl

An URL to open on a Tauri webview window.

Can be any of the following types:

- `string (format: uri)`: An external URL.
- `string` : The path portion of an app URL. For instance, to load `tauri://localhost/users/john`, you can simply provide `users/john` in this configuration.

Theme

System theme.

Can be any **ONE** of the following types:

- Light: Light theme.
- Dark: Dark theme.

TitleBarStyle

How the window title bar should be displayed on macOS.

Can be any **ONE** of the following types:

- Visible: A normal title bar.
- Transparent: Makes the title bar transparent, so the window background color is shown instead.

Useful if you don't need to have actual HTML under the title bar. This lets you avoid the caveats of using `TitleBarStyle::Overlay`. Will be more useful when Tauri lets you set a custom window background color.

- Overlay: Shows the title bar as a transparent overlay over the window's content.

Keep in mind:

- The height of the title bar is different on different OS versions, which can lead to window controls and title not being where you don't expect.
- You need to define a custom drag region to make your window draggable, however due to a limitation you can't drag the window when it's not in focus <https://github.com/tauri-apps/tauri/issues/4316>.
- The color of the window title depends on the system theme.

CliConfig

describes a CLI configuration

Type: `object`

Name	Type	Default	Description
<code>description</code>	<code>string?</code>	<code>null</code>	Command description which will be shown on the help information.
<code>longDescription</code>	<code>string?</code>	<code>null</code>	Command long description which will be shown on the help information.
<code>beforeHelp</code>	<code>string?</code>	<code>null</code>	Adds additional help information to be displayed in addition to auto-generated help. This information is displayed before the auto-generated help information. This is often used for header information.
<code>afterHelp</code>	<code>string?</code>	<code>null</code>	Adds additional help information to be displayed in addition to auto-generated help. This information is displayed after the auto-generated help information. This is often used to describe how to use the arguments, or caveats to be noted.
<code>args</code>	<code>array?</code>	<code>null</code>	List of arguments for the command
<code>subcommands</code>	<code>object?</code>	<code>null</code>	List of subcommands of this command

CliArg

A CLI argument definition.

Type: `object`

Name	Type	Default	Description
			The short version of the argument, without the preceding <code>-</code> .
<code>short</code>	<code>string?</code>	<code>null</code>	NOTE: Any leading <code>-</code> characters will be stripped, and only the first non-character will be used as the short version.
<code>name</code>	<code>string (required)</code>	<code>null</code>	The unique argument name
<code>description</code>	<code>string?</code>	<code>null</code>	The argument description which will be shown on the help information. Typically, this is a short (one line) description of the arg.

Name	Type	Default	Description
<code>longDescription</code>	<code>string?</code>	<code>null</code>	The argument long description which will be shown on the help information. Typically this a more detailed (multi-line) message that describes the argument.
			Specifies that the argument takes a value at run time.
<code>takesValue</code>	<code>boolean</code>	<code>false</code>	<p>NOTE: values for arguments may be specified in any of the following methods</p> <ul style="list-style-type: none"> - Using a space such as <code>-o value</code> or <code>--option value</code> - Using an equals and no space such as <code>-o=value</code> or <code>--option=value</code> - Use a short and no space such as <code>-ovalue</code>
			Specifies that the argument may have an unknown number of multiple values. Without any other settings, this argument may appear only once.
<code>multiple</code>	<code>boolean</code>	<code>false</code>	<p>For example, <code>--opt val1 val2</code> is allowed, but <code>--opt val1 val2 --opt val3</code> is not.</p> <p>NOTE: Setting this requires <code>takes_value</code> to be set to true.</p>
<code>multipleOccurrences</code>	<code>boolean</code>	<code>false</code>	<p>Specifies that the argument may appear more than once. For flags, this results in the number of occurrences of the flag being recorded. For example <code>-ddd</code> or <code>-d -d -d</code> would count as three occurrences. For options or arguments that take a value, this does not affect how many values they can accept. (i.e. only one at a time is allowed)</p> <p>For example, <code>--opt val1 --opt val2</code> is allowed, but <code>--opt val1 val2</code> is not.</p>

Name	Type	Default	Description
			Specifies how many values are required to satisfy this argument. For example, if you had a <code>-f <file></code> argument where you wanted exactly 3 'files' you would set <code>number_of_values = 3</code> , and this argument wouldn't be satisfied unless the user provided 3 and only 3 values.
<code>numberOfValues</code>	<code>integer? (format: uint)</code>	<code>null</code>	NOTE: Does not require <code>multiple_occurrences = true</code> to be set. Setting <code>multiple_occurrences = true</code> would allow <code>-f <file> <file> <file> -f <file> <file> <file></code> where as not setting it would only allow one occurrence of this argument.
			NOTE: implicitly sets <code>takes_value = true</code> and <code>multiple_values = true</code> .
<code>possibleValues</code>	<code>array?</code>	<code>null</code>	Specifies a list of possible values for this argument. At runtime, the CLI verifies that only one of the specified values was used, or fails with an error message.
<code>minValues</code>	<code>integer? (format: uint)</code>	<code>null</code>	Specifies the minimum number of values for this argument. For example, if you had a <code>-f <file></code> argument where you wanted at least 2 'files', you would set <code>minValues: 2</code> , and this argument would be satisfied if the user provided, 2 or more values.
<code>maxValues</code>	<code>integer? (format: uint)</code>	<code>null</code>	Specifies the maximum number of values are for this argument. For example, if you had a <code>-f <file></code> argument where you wanted up to 3 'files', you would set <code>.max_values(3)</code> , and this argument would be satisfied if the user provided, 1, 2, or 3 values.
<code>required</code>	<code>boolean</code>	<code>false</code>	Sets whether or not the argument is required by default.
			<ul style="list-style-type: none"> - Required by default means it is required, when no other conflicting rules have been evaluated - Conflicting rules take precedence over being required.
<code>requiredUnlessPresent</code>	<code>string?</code>	<code>null</code>	Sets an arg that override this arg's required setting i.e. this arg will be required unless this other argument is present.
<code>requiredUnlessPresentAll</code>	<code>array?</code>	<code>null</code>	Sets args that override this arg's required setting i.e. this arg will be required unless all these other arguments are present.

Name	Type	Default	Description
<code>requiredUnlessPresent</code>	<code>array?</code>	<code>null</code>	Sets args that override this arg's required setting i.e. this arg will be required unless at least one of these other arguments are present.
<code>conflictsWith</code>	<code>string?</code>	<code>null</code>	Sets a conflicting argument by name i.e. when using this argument, the following argument can't be present and vice versa.
<code>conflictsWithAll</code>	<code>array?</code>	<code>null</code>	The same as conflictsWith but allows specifying multiple two-way conflicts per argument.
<code>requires</code>	<code>string?</code>	<code>null</code>	Tets an argument by name that is required when this one is present i.e. when using this argument, the following argument must be present.
<code>requiresAll</code>	<code>array?</code>	<code>null</code>	Sts multiple arguments by names that are required when this one is present i.e. when using this argument, the following arguments must be present.
<code>requiresIf</code>	<code>array?</code>	<code>null</code>	Allows a conditional requirement with the signature [arg, value] the requirement will only become valid if <code>arg</code> 's value equals <code>value</code> .
<code>requiredIfEq</code>	<code>array?</code>	<code>null</code>	Allows specifying that an argument is required conditionally with the signature [arg, value] the requirement will only become valid if the <code>arg</code> 's value equals <code>value</code> .
<code>requireEquals</code>	<code>boolean?</code>	<code>null</code>	Requires that options use the --option=val syntax i.e. an equals between the option and associated value.
<code>index</code>	<code>integer? (format: uint, minimum: 1)</code>	<code>null</code>	The positional argument index, starting at 1.
			The index refers to position according to other positional argument. It does not define position in the argument list as a whole. When utilized with multiple=true, only the last positional argument may be defined as multiple (i.e. the one with the highest index).

BundleConfig

Configuration for tauri-bundler.

Type: `object`

Name	Type	Default	Description
<code>active</code>	<code>boolean</code>	<code>false</code>	Whether Tauri should bundle your application or just output the executable.

Name	Type	Default	Description
targets	BundleTarget	view	The bundle targets, currently supports [deb, appimage, msi, app, dmg, updater] or all.
identifier	string (required)	null	The application identifier in reverse domain name notation (e.g. <code>com.tauri.example</code>). This string must be unique across applications since it is used in system configurations like the bundle ID and path to the webview data directory. This string must contain only alphanumeric characters (A-Z, a-z, and 0-9), hyphens (-), and periods (.).
publisher	string ?	null	The application's publisher. Defaults to the second element in the identifier string. Currently maps to the Manufacturer property of the Windows Installer.
icon	string[]	[]	The app's icons
resources	array ?	null	App resources to bundle. Each resource is a path to a file or directory. Glob patterns are supported.
copyright	string ?	null	A copyright string associated with your application.
category	string ?	null	The application kind. Should be one of the following: Business, DeveloperTool, Education, Entertainment, Finance, Game, ActionGame, AdventureGame, ArcadeGame, BoardGame, CardGame, CasinoGame, DiceGame, EducationalGame, FamilyGame, KidsGame, MusicGame, PuzzleGame, RacingGame, RolePlayingGame, SimulationGame, SportsGame, StrategyGame, TriviaGame, WordGame, GraphicsAndDesign, HealthcareAndFitness, Lifestyle, Medical, Music, News, Photography, Productivity, Reference, SocialNetworking, Sports, Travel, Utility, Video, Weather.
shortDescription	string ?	null	A short description of your application.
longDescription	string ?	null	A longer, multi-line description of the application.
appimage	AppImageConfig	view	Configuration for the AppImage bundle.
deb	DebConfig	view	Configuration for the Debian bundle.
macOS	MacConfig	view	Configuration for the macOS bundles.

Name	Type	Default	Description
<code>external-Bin</code>	<code>array ?</code>	<code>null</code>	<p>A list of—either absolute or relative—paths to binaries to embed with your application.</p> <p>Note that Tauri will look for system-specific binaries following the pattern <code>binary-name{-target-triple}{.system-extension}</code>.</p> <p>E.g. for the external binary <code>my-binary</code>, Tauri looks for:</p> <ul style="list-style-type: none"> - <code>my-binary-x86_64-pc-windows-msvc.exe</code> for Windows - <code>my-binary-x86_64-apple-darwin</code> for macOS - <code>my-binary-x86_64-unknown-linux-gnu</code> for Linux <p>so don't forget to provide binaries for all targeted platforms.</p>

|

`windows`| [WindowsConfig](#) | [view](#) | Configuration for the Windows bundle. |

BundleTarget

Targets to bundle. Each value is case insensitive.

Can be any of the following types:

- `all` : Bundle all targets.
- `BundleType[]` : A list of bundle targets.
- `BundleType` : A single bundle target.

BundleType

A bundle referenced by tauri-bundler.

Can be any **ONE** of the following types:

- `deb`: The debian bundle (`.deb`).
- `appimage`: The AppImage bundle (`.appimage`).
- `msi`: The Microsoft Installer bundle (`.msi`).
- `app`: The macOS application bundle (`.app`).
- `dmg`: The Apple Disk Image bundle (`.dmg`).
- `updater`: The Tauri updater bundle.

AppImageConfig

Configuration for AppImage bundles.

Type: `object`

Name	Type	Default	Description
<code>bundleMediaFramework</code>	<code>boolean</code>	<code>false</code>	Include additional gstreamer dependencies needed for audio and video playback. This increases the bundle size by ~15-35MB depending on your build system.

DebConfig

Configuration for Debian (.deb) bundles.

Type: `object`

Name	Type	Default	Description
<code>depends</code>	<code>array?</code>	<code>null</code>	The list of deb dependencies your application relies on.
<code>files</code>	<code>object</code>	<code>null</code>	The files to include on the package.

MacConfig

Configuration for the macOS bundles.

Type: `object`

Name	Type	Default	Description
<code>frameworks</code>	<code>array?</code>	<code>null</code>	A list of strings indicating any macOS X frameworks that need to be bundled with the application.
<code>minimumSystemVersion</code>	<code>string?</code>	<code>null</code>	If a name is used, .framework must be omitted and it will look for standard install locations. You may also use a path to a specific framework.
<code>exceptionDomain</code>	<code>string?</code>	<code>null</code>	A version string indicating the minimum macOS X version that the bundled application supports. Defaults to <code>10.13</code> .
<code>license</code>	<code>string?</code>	<code>null</code>	Setting it to <code>null</code> completely removes the <code>LSMinimumSystemVersion</code> field on the bundle's <code>Info.plist</code> and the <code>MACOSX_DEPLOYMENT_TARGET</code> environment variable.
<code>license</code>	<code>string?</code>	<code>null</code>	An empty string is considered an invalid value so the default value is used.
<code>exceptionDomain</code>	<code>string?</code>	<code>null</code>	Allows your application to communicate with the outside world. It should be a lowercase, without port and protocol domain name.
<code>license</code>	<code>string?</code>	<code>null</code>	The path to the license file to add to the DMG bundle.

Name	Type	Default	Description
signingIdentity	string ? null		Identity to use for code signing.
providerShortName	string ? null		Provider short name for notarization.
entitlements	string ? null		Path to the entitlements file.

WindowsConfig

Windows bundler configuration.

Type: object

Name	Type	Default	Description
digestAlgorithm	string ?	null	Specifies the file digest algorithm to use for creating file signatures. Required for code signing. SHA-256 is recommended.
certificateThumbprint	string ?	null	Specifies the SHA1 hash of the signing certificate.
timestampUrl	string ?	null	Server to use during timestamping.
tsp	boolean	false	Whether to use Time-Stamp Protocol (TSP, a.k.a. RFC 3161) for the timestamp server. Your code signing provider may use a TSP timestamp server, like e.g. SSL.com does. If so, enable TSP by setting to true.
webviewInstallMode	WebviewInstallMode	view	The installation mode for the Webview2 runtime.
webviewFixedRuntimePath	string ?	null	Path to the webview fixed runtime to use. Overwrites <code>webview_install_mode</code> if set.
			Will be removed in v2, prefer the <code>webview_install_mode</code> option.
			The fixed version can be downloaded on the official website . The <code>.cab</code> file must be extracted to a folder and this folder path must be defined on this field.

Name	Type	Default	Description
			Validates a second app installation, blocking the user from installing an older version if set to <code>false</code> .
<code>allowDowngrades</code>	<code>boolean</code>	<code>true</code>	For instance, if <code>1.2.1</code> is installed, the user won't be able to install app version <code>1.2.0</code> or <code>1.1.5</code> . The default value of this flag is <code>true</code> .
<code>wix</code>	<code>WixConfig?</code>	<code>view</code>	Configuration for the MSI generated with WiX.

WebviewInstallMode

Install modes for the Webview2 runtime. Note that for the updater bundle `DownloadBootstrapper` is used.

For more information see <https://tauri.app/v1/guides/building/windows>.

Can be any **ONE** of the following types:

- { type: skip }: Do not install the Webview2 as part of the Windows Installer.
- { type: downloadBootstrapper, silent: boolean }: Download the bootstrapper and run it. Requires internet connection. Results in a smaller installer size, but is not recommended on Windows 7.
- { type: embedBootstrapper, silent: boolean }: Embed the bootstrapper and run it. Requires internet connection. Increases the installer size by around 1.8MB, but offers better support on Windows 7.
- { type: offlineInstaller, silent: boolean }: Embed the offline installer and run it. Does not require internet connection. Increases the installer size by around 127MB.
- { type: fixedRuntime, path: string }: Embed a fixed webview2 version and use it at runtime. Increases the installer size by around 180MB.

WixConfig

Configuration for the MSI bundle using WiX.

Type: `object`

Name	Type	Default	Description
<code>language</code>	<code>WixLanguage</code>	<code>view</code>	The installer languages to build. See https://docs.microsoft.com/en-us/windows/win32/msi/localizing-the-error-and-actiontext-tables .
<code>template</code>	<code>string?</code>	<code>null</code>	A custom .wxs template to use.
<code>fragmentPaths</code>	<code>string[]</code>	<code>[]</code>	A list of paths to .wxs files with WiX fragments to use.

Name	Type	Default	Description
componentGroupRefs	string[]	[]	The ComponentGroup element ids you want to reference from the fragments.
componentRefs	string[]	[]	The Component element ids you want to reference from the fragments.
featureGroupRefs	string[]	[]	The FeatureGroup element ids you want to reference from the fragments.
featureRefs	string[]	[]	The Feature element ids you want to reference from the fragments.
mergeRefs	string[]	[]	The Merge element ids you want to reference from the fragments.
skipWebviewInstall	boolean	false	Disables the Webview2 runtime installation after app install. Will be removed in v2, prefer the [WindowsConfig::webview_install_mode] option.
license	string?	null	The path to the license file to render on the installer. Must be an RTF file, so if a different extension is provided, we convert it to the RTF format.
enableElevatedUpdateTask	boolean	false	Create an elevated update task within Windows Task Scheduler.
bannerPath	string?	null	Path to a bitmap file to use as the installation user interface banner. This bitmap will appear at the top of all but the first page of the installer. The required dimensions are 493px x 58px.
dialogImagePath	string?	null	Path to a bitmap file to use on the installation user interface dialogs. It is used on the welcome and completion dialogs. The required dimensions are 493px x 312px.

WixLanguage

The languages to build using WiX.

Can be any of the following types:

- `string` : A single language to build, without configuration.
- `string[]` : A list of languages to build, without configuration.
- `object` : A map of languages and its configuration.

WixLanguageConfig

Configuration for a target language for the WiX build.

Type: `object`

Name	Type	Default	Description
<code>localePath</code>	<code>string?</code>	<code>null</code>	The path to a locale (<code>.wxl</code>) file. See https://wixtoolset.org/documentation/manual/v3/howtos/ui_and_localization/build_a_localized_version.html .

AllowlistConfig

Allowlist configuration.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all API features.
<code>fs</code>	<code>FsAllowlistConfig</code>	<code>view</code>	File system API allowlist.
<code>window</code>	<code>WindowAllowlistConfig</code>	<code>view</code>	Window API allowlist.
<code>shell</code>	<code>ShellAllowlistConfig</code>	<code>view</code>	Shell API allowlist.
<code>dialog</code>	<code>DialogAllowlistConfig</code>	<code>view</code>	Dialog API allowlist.
<code>http</code>	<code>HttpAllowlistConfig</code>	<code>view</code>	HTTP API allowlist.
<code>notification</code>	<code>NotificationAllowlistConfig</code>	<code>view</code>	Notification API allowlist.
<code>globalShortcut</code>	<code>GlobalShortcutAllowlistConfig</code>	<code>view</code>	Global shortcut API allowlist.
<code>os</code>	<code>OsAllowlistConfig</code>	<code>view</code>	OS allowlist.
<code>path</code>	<code>PathAllowlistConfig</code>	<code>view</code>	Path API allowlist.
<code>protocol</code>	<code>ProtocolAllowlistConfig</code>	<code>view</code>	Custom protocol allowlist.
<code>process</code>	<code>ProcessAllowlistConfig</code>	<code>view</code>	Process API allowlist.
<code>clipboard</code>	<code>ClipboardAllowlistConfig</code>	<code>view</code>	Clipboard APIs allowlist.
<code>app</code>	<code>AppAllowlistConfig</code>	<code>view</code>	App APIs allowlist.

FsAllowlistConfig

Allowlist for the file system APIs.

Type: `object`

Name	Type	Default	Description
scope	FsAllowlistScope	[]	The access scope for the filesystem APIs.
all	boolean	false	Use this flag to enable all file system API features.
readFile	boolean	false	Read file from local filesystem.
writeFile	boolean	false	Write file to local filesystem.
readDir	boolean	false	Read directory from local filesystem.
copyFile	boolean	false	Copy file from local filesystem.
createDir	boolean	false	Create directory from local filesystem.
removeDir	boolean	false	Remove directory from local filesystem.
removeFile	boolean	false	Remove file from local filesystem.
renameFile	boolean	false	Rename file from local filesystem.
exists	boolean	false	Check if path exists on the local filesystem.

FsAllowlistScope

Filesystem scope definition. It is a list of glob patterns that restrict the API access from the webview.

Each pattern can start with a variable that resolves to a system base directory. The variables are: \$AUDIO, \$CACHE, \$CONFIG, \$DATA, \$LOCALDATA, \$DESKTOP, \$DOCUMENT, \$DOWNLOAD, \$EXE, \$FONT, \$HOME, \$PICTURE, \$PUBLIC, \$RUNTIME, \$TEMPLATE, \$VIDEO, \$RESOURCE, \$APP, \$LOG, \$TEMP, \$APPCONFIG, \$APPDATA, \$APPLOCALDATA, \$APPCACHE, \$APPLOG .

Can be any of the following types:

- string[] : A list of paths that are allowed by this scope.
- object : A complete scope configuration.

WindowAllowlistConfig

Allowlist for the window APIs.

Type: object

Name	Type	Default	Description
all	bool- ean	false	Use this flag to enable all window API features.
create	bool- ean	false	Allows dynamic window creation.
center	bool- ean	false	Allows centering the window.
requestUserAttention	bool- ean	false	Allows requesting user attention on the window.

Name	Type	Default	Description
setResizable	bool- ean	false	Allows setting the resizable flag of the window.
setTitle	bool- ean	false	Allows changing the window title.
maximize	bool- ean	false	Allows maximizing the window.
unmaximize	bool- ean	false	Allows unmaximizing the window.
minimize	bool- ean	false	Allows minimizing the window.
unminimize	bool- ean	false	Allows unminimizing the window.
show	bool- ean	false	Allows showing the window.
hide	bool- ean	false	Allows hiding the window.
close	bool- ean	false	Allows closing the window.
setDecorations	bool- ean	false	Allows setting the decorations flag of the window.
setAlwaysOnTop	bool- ean	false	Allows setting the always_on_top flag of the window.
setSize	bool- ean	false	Allows setting the window size.
setMinSize	bool- ean	false	Allows setting the window minimum size.
setMaxSize	bool- ean	false	Allows setting the window maximum size.
setPosition	bool- ean	false	Allows changing the position of the window.
setFullscreen	bool- ean	false	Allows setting the fullscreen flag of the window.
setFocus	bool- ean	false	Allows focusing the window.
setIcon	bool- ean	false	Allows changing the window icon.

Name	Type	Default	Description
setSkipTaskbar	bool- ean	false	Allows setting the skip_taskbar flag of the window.
setCursorGrab	bool- ean	false	Allows grabbing the cursor.
setCursorVisible	bool- ean	false	Allows setting the cursor visibility.
setCursorIcon	bool- ean	false	Allows changing the cursor icon.
setCursorPosition	bool- ean	false	Allows setting the cursor position.
setIgnoreCursor- Events	bool- ean	false	Allows ignoring cursor events.
startDragging	bool- ean	false	Allows start dragging on the window.
print	bool- ean	false	Allows opening the system dialog to print the window content.

ShellAllowlistConfig

Allowlist for the shell APIs.

Type: `object`

Name	Type	Default	Description
scope	ShellAllowlistScope	[]	Access scope for the binary execution APIs. Sidecars are automatically enabled.
all	boolean	false	Use this flag to enable all shell API features.
exe- cute	boolean	false	Enable binary execution.
side- car	boolean	false	Enable sidecar execution, allowing the JavaScript layer to spawn a sidecar command, an executable that is shipped with the application. For more information see https://tauri.app/v1/guides/building/sidecar .
open	ShellAllowlistOpen	false	Open URL with the user's default application.

ShellAllowlistScope

Shell scope definition. It is a list of command names and associated CLI arguments that restrict the API access from the webview.

Type: `ShellAllowedCommand[]`

ShellAllowedCommand

A command allowed to be executed by the webview API.

Type: `object`

Name	Type	Default	Description
The name for this allowed shell command configuration.			
<code>name</code>	<code>string</code> (required)	<code>null</code>	This name will be used inside of the webview API to call this command along with any specified arguments.
<code>cmd</code>	<code>string</code>	<code>null</code>	The command name. It can start with a variable that resolves to a system base directory. The variables are: <code>\$AUDIO</code> , <code>\$CACHE</code> , <code>\$CONFIG</code> , <code>\$DATA</code> , <code>\$LOCALDATA</code> , <code>\$DESKTOP</code> , <code>\$DOCUMENT</code> , <code>\$DOWNLOAD</code> , <code>\$EXE</code> , <code>\$FONT</code> , <code>\$HOME</code> , <code>\$PICTURE</code> , <code>\$PUBLIC</code> , <code>\$RUNTIME</code> , <code>\$TEMPLATE</code> , <code>\$VIDEO</code> , <code>\$RESOURCE</code> , <code>\$APP</code> , <code>\$LOG</code> , <code>\$TEMP</code> , <code>\$APPCONFIG</code> , <code>\$APPDATA</code> , <code>\$APPLOCALDATA</code> , <code>\$APPCACHE</code> , <code>\$APPLOG</code> .
<code>args</code>	<code>ShellAllowedArgs</code>	<code>false</code>	The allowed arguments for the command execution.
<code>sidecar</code>	<code>boolean</code>	<code>false</code>	If this command is a sidecar command.

ShellAllowedArgs

A set of command arguments allowed to be executed by the webview API.

A value of `true` will allow any arguments to be passed to the command. `false` will disable all arguments. A list of `[ShellAllowedArg]` will set those arguments as the only valid arguments to be passed to the attached command configuration.

Can be any of the following types:

- `boolean` : Use a simple boolean to allow all or disable all arguments to this command configuration.
- `ShellAllowedArg[]` : A specific set of `[ShellAllowedArg]` that are valid to call for the command configuration.

ShellAllowedArg

A command argument allowed to be executed by the webview API.

Can be any of the following types:

- `string` : A non-configurable argument that is passed to the command in the order it was specified.
- `object` : A variable that is set while calling the command from the webview API.

ShellAllowlistOpen

Defines the `shell > open` api scope.

Can be any of the following types:

- `boolean` : If the shell open API should be enabled.

If enabled, the default validation regex (`^https?://`) is used.

- `string` : Enable the shell open API, with a custom regex that the opened path must match against.

If using a custom regex to support a non-`http(s)` schema, care should be used to prevent values that allow flag-like strings to pass validation. e.g. `--enable-debugging`, `-i`, `/R`.

DialogAllowlistConfig

Allowlist for the dialog APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all dialog API features.
<code>open</code>	<code>boolean</code>	<code>false</code>	Allows the API to open a dialog window to pick files.
<code>save</code>	<code>boolean</code>	<code>false</code>	Allows the API to open a dialog window to pick where to save files.
<code>message</code>	<code>boolean</code>	<code>false</code>	Allows the API to show a message dialog window.
<code>ask</code>	<code>boolean</code>	<code>false</code>	Allows the API to show a dialog window with Yes/No buttons.
<code>confirm</code>	<code>boolean</code>	<code>false</code>	Allows the API to show a dialog window with Ok/Cancel buttons.

HttpAllowlistConfig

Allowlist for the HTTP APIs.

Type: `object`

Name	Type	Default	Description
<code>scope</code>	<code>HttpAllowlistScope</code>	<code>[]</code>	The access scope for the HTTP APIs.
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all HTTP API features.
<code>request</code>	<code>boolean</code>	<code>false</code>	Allows making HTTP requests.

HttpAllowlistScope

HTTP API scope definition. It is a list of URLs that can be accessed by the webview when using the HTTP APIs. The scoped URL is matched against the request URL using a glob pattern.

Examples:

- `https://**:` allows all HTTPS urls
- `https://*.github.com/tauri-apps/tauri:` allows any subdomain of github.com with the tauri-apps/api path
- `https://myapi.service.com/users/*:` allows access to any URLs that begins with `https://myapi.service.-com/users/`

Type: `string _(format: uri)_[]`

NotificationAllowlistConfig

Allowlist for the notification APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all notification API features.

GlobalShortcutAllowlistConfig

Allowlist for the global shortcut APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all global shortcut API features.

OsAllowlistConfig

Allowlist for the OS APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all OS API features.

PathAllowlistConfig

Allowlist for the path APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all path API features.

ProtocolAllowlistConfig

Allowlist for the custom protocols.

Type: `object`

Name	Type	Default	Description
<code>assetScope</code>	<code>FsAllowlistScope</code>	<code>[]</code>	The access scope for the asset protocol.
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all custom protocols.
<code>asset</code>	<code>boolean</code>	<code>false</code>	Enables the asset protocol.

ProcessAllowlistConfig

Allowlist for the process APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>bool-</code> <code>ean</code>	<code>false</code>	Use this flag to enable all process APIs.
<code>relaunch</code>	<code>bool-</code> <code>ean</code>	<code>false</code>	Enables the relaunch API.
			Dangerous option that allows macOS to relaunch even if the binary contains a symlink.
<code>relaunchDangerousAllowSym-</code> <code>linkMacos</code>	<code>bool-</code> <code>ean</code>	<code>false</code>	This is due to macOS having less symlink protection. Highly recommended to not set this flag unless you have a very specific reason too, and understand the implications of it.
<code>exit</code>	<code>bool-</code> <code>ean</code>	<code>false</code>	Enables the exit API.

ClipboardAllowlistConfig

Allowlist for the clipboard APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all clipboard APIs.
<code>writeText</code>	<code>boolean</code>	<code>false</code>	Enables the clipboard's <code>writeText</code> API.
<code>readText</code>	<code>boolean</code>	<code>false</code>	Enables the clipboard's <code>readText</code> API.

AppAllowlistConfig

Allowlist for the app APIs.

Type: `object`

Name	Type	Default	Description
<code>all</code>	<code>boolean</code>	<code>false</code>	Use this flag to enable all app APIs.
<code>show</code>	<code>boolean</code>	<code>false</code>	Enables the app's <code>show</code> API.
<code>hide</code>	<code>boolean</code>	<code>false</code>	Enables the app's <code>hide</code> API.

SecurityConfig

Security configuration.

Type: `object`

Name	Type	Default	Description
<code>csp</code>	<code>Csp?</code>	<code>view</code>	The Content Security Policy that will be injected on all HTML files on the built application. If <code>dev_csp</code> is not specified, this value is also injected on dev.
<code>devCsp</code>	<code>Csp?</code>	<code>view</code>	The Content Security Policy that will be injected on all HTML files on development.
<code>freezePrototype</code>	<code>boolean</code>	<code>false</code>	Freeze the <code>Object.prototype</code> when using the custom protocol.

Name	Type	Default	Description
			Disables the Tauri-injected CSP sources.
			At compile time, Tauri parses all the frontend assets and changes the Content-Security-Policy to only allow loading of your own scripts and styles by injecting nonce and hash sources. This stricts your CSP, which may introduce issues when using along with other flexing sources.
<code>dangerousDisableAssetCspModification</code>	<code>DisabledCspModificationKind</code>	<code>false</code>	This configuration option allows both a boolean and a list of strings as value. A boolean instructs Tauri to disable the injection for all CSP injections, and a list of strings indicates the CSP directives that Tauri cannot inject.
			WARNING: Only disable this if you know what you are doing and have properly configured the CSP. Your application might be vulnerable to XSS attacks without this Tauri protection.

Csp

A Content-Security-Policy definition. See <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.

Can be any of the following types:

- `string` : The entire CSP policy in a single text string.
- `object` : An object mapping a directive with its sources values as a list of strings.

CspDirectiveSources

A Content-Security-Policy directive source list. See <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/Sources#sources>.

Can be any of the following types:

- `string` : An inline list of CSP sources. Same as `List`, but concatenated with a space separator.
- `string[]` : A list of CSP sources. The collection will be concatenated with a space separator for the CSP string.

DisabledCspModificationKind

The possible values for the `dangerous_disable_asset_csp_modification` config option.

Can be any of the following types:

- `boolean` : If `true`, disables all CSP modification. `false` is the default value and it configures Tauri to control the CSP.
- `string[]` : Disables the given list of CSP directives modifications.

UpdaterConfig

The Updater configuration object.

Type: `object`

Name	Type	Default	Description
<code>active</code>	<code>boolean</code>	<code>false</code>	Whether the updater is active or not.
<code>dialog</code>	<code>boolean</code>	<code>true</code>	Display built-in dialog or use event system if disabled.
<code>end-points</code>	<code>array ?</code>	<code>null</code>	<p>The updater endpoints. TLS is enforced on production.</p> <p>The updater URL can contain the following variables:</p> <ul style="list-style-type: none"> - <code>{{current_version}}</code>: The version of the app that is requesting the update - <code>{{target}}</code>: The operating system name (one of <code>linux</code>, <code>windows</code> or <code>darwin</code>). - <code>{{arch}}</code>: The architecture of the machine (one of <code>x86_64</code>, <code>i686</code>, <code>aarch64</code> or <code>armv7</code>). <p># Examples</p> <ul style="list-style-type: none"> - https://my.cdn.com/latest.json: a raw JSON endpoint that returns the latest version and download links for each platform.

- https://updates.app.dev/{{target}}?version={{current_version}}&arch={{arch}}: a dedicated API with positional and query string arguments.

pubkey	string	null	Signature public key.
win-dows	UpdaterWindowsConfig	view	The Windows configuration for the updater.

UpdaterEndpoint

A URL to an updater server.

The URL must use the `https` scheme on production.

Type: `string (format: uri)`

UpdaterWindowsConfig

The updater configuration for Windows.

Type: `object`

Name	Type	Default	Description
install-Mode	WindowsUpdateInstallMode	view	The installation mode for the update on Windows. Defaults to <code>passive</code> .

WindowsUpdateInstallMode

Install modes for the Windows update.

Can be any **ONE** of the following types:

- basicUi: Specifies there's a basic UI during the installation process, including a final dialog box at the end.
- quiet: The quiet mode means there's no user interaction required. Requires admin privileges if the installer does.
- passive: Specifies unattended mode, which means the installation only shows a progress bar.

SystemTrayConfig

Configuration for application system tray icon.

Type: `object`

Name	Type	Default	Description
iconPath	string (required)	null	Path to the default icon to use on the system tray.

Name	Type	Default	Description
iconAsTemplate	boolean	false	A Boolean value that determines whether the image represents a template image on macOS.
menuOnLeftClick	boolean	true	A Boolean value that determines whether the menu should appear when the tray icon receives a left click on macOS.
title	string ?	null	Title for MacOS tray

BuildConfig

The Build configuration object.

Type: `object`

Name	Type	Default	Description
runner	string ?	null	The binary used to build and run the application.
devPath	AppUrl	view	The path to the application assets or URL to load in development. This is usually an URL to a dev server, which serves your application assets with live reloading. Most modern JavaScript bundlers provides a way to start a dev server by default. See vite , Webpack DevServer and sirv for examples on how to set up a dev server.
distDir	AppUrl	view	The path to the application assets or URL to load in production. When a path relative to the configuration file is provided, it is read recursively and all files are embedded in the application binary. Tauri then looks for an <code>index.html</code> file unless you provide a custom window URL. You can also provide a list of paths to be embedded, which allows granular control over what files are added to the binary. In this case, all files are added to the root and you must reference it that way in your HTML files.

Name	Type	Default	Description
<code>beforeDevCommand</code>	<code>BeforeDevCommand?</code>	<code>view</code>	When an URL is provided, the application won't have bundled assets and the application will load that URL by default.
<code>beforeBuildCommand</code>	<code>HookCommand?</code>	<code>view</code>	A shell command to run before <code>tauri dev</code> kicks in.
<code>beforeBundleCommand</code>	<code>HookCommand?</code>	<code>view</code>	The TAURI_PLATFORM, TAURI_ARCH, TAURI_FAMILY, TAURI_PLATFORM_VERSION, TAURI_PLATFORM_TYPE and TAURI_DEBUG environment variables are set if you perform conditional compilation.
<code>features</code>	<code>array?</code>	<code>null</code>	Features passed to <code>cargo</code> commands.
<code>withGlobalTauri</code>	<code>boolean</code>	<code>false</code>	Whether we should inject the Tauri API on <code>window.__TAURI__</code> or not.

AppUrl

Defines the URL or assets to embed in the application.

Can be any of the following types:

- `WindowUrl` : The app's external URL, or the path to the directory containing the app assets.
- `string[]` : An array of files to embed on the app.

BeforeDevCommand

Describes the shell command to run before `tauri dev`.

Can be any of the following types:

- `string` : Run the given script with the default options.
- `object` : Run the given script with custom options.

HookCommand

Describes a shell command to be executed when a CLI hook is triggered.

Can be any of the following types:

- `string` : Run the given script with the default options.
- `object` : Run the given script with custom options.

PluginConfig

The plugin configs holds a HashMap mapping a plugin name to its configuration object.

Type: `object`

[Go to TOC](#)

@tauri-apps/api

Modules

- [app](#)
- [cli](#)
- [clipboard](#)
- [dialog](#)
- [event](#)
- [fs](#)
- [globalShortcut](#)
- [http](#)
- [mocks](#)
- [notification](#)
- [os](#)
- [path](#)
- [process](#)
- [shell](#)
- [tauri](#)
- [updater](#)
- [window](#)

[Go to TOC](#)

app

Get application metadata.

This package is also accessible with `window.__TAURI__.app` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.app` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      app: {
        all: true, // enable all app APIs
        show: true,
        hide: true
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Functions

getName

```
getName(): Promise < string >
```

Gets the application name.

Example

```
import { getName } from '@tauri-apps/api/app';
const appName = await getName();
```

Since: 1.0.0

Returns: `Promise < string >`

getTauriVersion

```
getTauriVersion(): Promise < string >
```

Gets the Tauri version.

Example

```
import { getTauriVersion } from '@tauri-apps/api/app';
const tauriVersion = await getTauriVersion();
```

Since: 1.0.0**Returns:** `Promise < string >`**getVersion**

```
getVersion(): Promise < string >
```

Gets the application version.

Example

```
import { getVersion } from '@tauri-apps/api/app';
const appVersion = await getVersion();
```

Since: 1.0.0**Returns:** `Promise < string >`**hide**

```
hide(): Promise < void >
```

Hides the application on macOS.

Example

```
import { hide } from '@tauri-apps/api/app';
await hide();
```

Since: 1.2.0**Returns:** `Promise < void >`**show**

```
show(): Promise < void >
```

Shows the application on macOS. This function does not automatically focus any specific app window.

Example

```
import { show } from '@tauri-apps/api/app';
await show();
```

Since: 1.2.0

Returns: `Promise<void>`

[Go to TOC](#)

cli

Parse arguments from your Command Line Interface.

This package is also accessible with `window.__TAURI__.cli` when `build.withGlobalTauri` in `tauri-conf.json` is set to `true`.

Interfaces

ArgMatch

Since: 1.0.0

Properties

occurrences

occurrences: number

Number of occurrences

Defined in: [cli.ts:27](#)

value

value: null | string | boolean | string []

string if takes value boolean if flag string[] or null if takes multiple values

Defined in: [cli.ts:23](#)

CliMatches

Since: 1.0.0

Properties

args

args: { [name: string]: ArgMatch ; }

Defined in: [cli.ts:42](#)

subcommand**subcommand**: null | SubcommandMatch**Defined in:** [cli.ts:43](#)

SubcommandMatch

Since: 1.0.0

Properties

matches**matches**: CliMatches**Defined in:** [cli.ts:35](#)**name****name**: string**Defined in:** [cli.ts:34](#)

Functions

getMatches

getMatches(): Promise < CliMatches >

Parse the arguments provided to the current process and get the matches using the configuration defined `tauri.cli` in `tauri.conf.json`

Example

```
import { getMatches } from '@tauri-apps/api/cli';
const matches = await getMatches();
if (matches.subcommand?.name === 'run') {
    // `./your-app run $ARGS` was executed
    const args = matches.subcommand?.matches.args
    if ('debug' in args) {
        // `./your-app run --debug` was executed
    }
}
```

```
    } else {
      const args = matches.args
      // `./your-app $ARGS` was executed
    }
```

Since: 1.0.0

Returns: `Promise < CliMatches >`

[Go to TOC](#)

clipboard

Read and write to the system clipboard.

This package is also accessible with `window.__TAURI__.clipboard` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.clipboard` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      clipboard: {
        all: true, // enable all Clipboard APIs
        writeText: true,
        readText: true
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Functions

readText

```
readText(): Promise<string | null>
```

Gets the clipboard content as plain text.

Example

```
import { readText } from '@tauri-apps/api/clipboard';
const clipboardText = await readText();
```

Since: 1.0.0.

Returns: `Promise<string | null>`

writeText

```
writeText(text : string): Promise<void>
```

Writes plain text to the clipboard.

Example

```
import { writeText, readText } from '@tauri-apps/api/clipboard';
await writeText('Tauri is awesome!');
assert(await readText(), 'Tauri is awesome!');
```

Since: 1.0.0.

Parameters

Name	Type
text	string

Returns: `Promise<void>`

A promise indicating the success or failure of the operation.

[Go to TOC](#)

dialog

Native system dialogs for opening and saving files.

This package is also accessible with `window.__TAURI__.dialog` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.dialog` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      dialog: {
        all: true, // enable all dialog APIs
        open: true, // enable file open API
        save: true // enable file save API
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Interfaces

DialogFilter

Extension filters for the file dialog.

Since: 1.0.0

Properties

extensions

```
extensions: string []
```

Extensions to filter, without a `.` prefix.

Example

```
extensions: ['svg', 'png']
```

Defined in: [dialog.ts:45](#)

name**name:** string

Filter name.

Defined in: [dialog.ts:37](#)

MessageDialogOptions

Since: 1.0.0

Properties**title**

Optional **title:** string

The title of the dialog. Defaults to the app name.

Defined in: [dialog.ts:94](#)

type

Optional **type:** info | warning | error

The type of the dialog. Defaults to `info`.

Defined in: [dialog.ts:96](#)

OpenDialogOptions

Options for the open dialog.

Since: 1.0.0

Properties**defaultPath**

Optional **defaultPath:** string

Initial directory or file path.

Defined in: [dialog.ts:59](#)

`directory`

Optional **directory**: boolean

Whether the dialog is a directory selection or not.

Defined in: [dialog.ts:63](#)

`filters`

Optional **filters**: DialogFilter []

The filters of the dialog.

Defined in: [dialog.ts:57](#)

`multiple`

Optional **multiple**: boolean

Whether the dialog allows multiple selection or not.

Defined in: [dialog.ts:61](#)

`recursive`

Optional **recursive**: boolean

If `directory` is true, indicates that it will be read recursively later. Defines whether subdirectories will be allowed on the scope or not.

Defined in: [dialog.ts:68](#)

`title`

Optional **title**: string

The title of the dialog window.

Defined in: [dialog.ts:55](#)

SaveDialogOptions

Options for the save dialog.

Since: 1.0.0

Properties

defaultPath

Optional **defaultPath**: string

Initial directory or file path. If it's a directory path, the dialog interface will change to that folder. If it's not an existing directory, the file name will be set to the dialog's file name input and the dialog will be set to the parent folder.

Defined in: [dialog.ts:86](#)

filters

Optional **filters**: DialogFilter []

The filters of the dialog.

Defined in: [dialog.ts:80](#)

title

Optional **title**: string

The title of the dialog window.

Defined in: [dialog.ts:78](#)

Functions

ask

```
ask( message : string , options? : string | MessageDialogOptions ): Promise < boolean >
```

Shows a question dialog with `Yes` and `No` buttons.

Example

```
import { ask } from '@tauri-apps/api/dialog';
const yes = await ask('Are you sure?', 'Tauri');
const yes2 = await ask('This action cannot be reverted. Are you sure?', { title: 'Tauri', type: 'warning' });
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>message</code>	<code>string</code>	The message to show.
<code>options?</code>	<code>string MessageDialogOptions</code>	The dialog's options. If a string, it represents the dialog title.

Returns: `Promise < boolean >`

A promise resolving to a boolean indicating whether `Yes` was clicked or not.

confirm

```
confirm( message : string , options? : string | MessageDialogOptions ): Promise < boolean
>
```

Shows a question dialog with `Ok` and `Cancel` buttons.

Example

```
import { confirm } from '@tauri-apps/api/dialog';
const confirmed = await confirm('Are you sure?', 'Tauri');
const confirmed2 = await confirm('This action cannot be reverted. Are you sure?',
{ title: 'Tauri', type: 'warning' });
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>message</code>	<code>string</code>	The message to show.
<code>options?</code>	<code>string MessageDialogOptions</code>	The dialog's options. If a string, it represents the dialog title.

Returns: `Promise < boolean >`

A promise resolving to a boolean indicating whether `Ok` was clicked or not.

message

```
message( message : string , options? : string | MessageDialogOptions ): Promise < void >
```

Shows a message dialog with an `Ok` button.

Example

```
import { message } from '@tauri-apps/api/dialog';
await message('Tauri is awesome', 'Tauri');
await message('File not found', { title: 'Tauri', type: 'error' });
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>message</code>	<code>string</code>	The message to show.
<code>options?</code>	<code>string</code> <code>MessageDialogOptions</code>	The dialog's options. If a string, it represents the dialog title.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

open

```
open( options? : OpenDialogOptions ): Promise < null | string | string []>
```

Open a file/directory selection dialog.

The selected paths are added to the filesystem and asset protocol allowlist scopes. When security is more important than the ease of use of this API, prefer writing a dedicated command instead.

Note that the allowlist scope change is not persisted, so the values are cleared when the application is restarted.

You can save it to the filesystem using [tauri-plugin-persisted-scope](#).

Example

```
import { open } from '@tauri-apps/api/dialog';
// Open a selection dialog for image files
const selected = await open({
```

```

multiple: true,
filters: [
  name: 'Image',
  extensions: ['png', 'jpeg']
]
});
if (Array.isArray(selected)) {
  // user selected multiple files
} else if (selected === null) {
  // user cancelled the selection
} else {
  // user selected a single file
}

```

Example

```

import { open } from '@tauri-apps/api/dialog';
import { appDir } from '@tauri-apps/api/path';
// Open a selection dialog for directories
const selected = await open({
  directory: true,
  multiple: true,
  defaultPath: await appDir(),
});
if (Array.isArray(selected)) {
  // user selected multiple directories
} else if (selected === null) {
  // user cancelled the selection
} else {
  // user selected a single directory
}

```

Since: 1.0.0**Parameters**

Name	Type
options	OpenDialogOptions

Returns: Promise<null | string | string[]>

A promise resolving to the selected path(s)

save

```
save( options? : SaveDialogOptions ): Promise<string | null>
```

Open a file/directory save dialog.

The selected path is added to the filesystem and asset protocol allowlist scopes. When security is more important than the ease of use of this API, prefer writing a dedicated command instead.

Note that the allowlist scope change is not persisted, so the values are cleared when the application is restarted.

You can save it to the filesystem using [tauri-plugin-persisted-scope](#).

Example

```
import { save } from '@tauri-apps/api/dialog';
const filePath = await save({
  filters: [
    {
      name: 'Image',
      extensions: ['png', 'jpeg']
    }
  ]
});
```

Since: 1.0.0

Parameters

Name	Type
options	SaveDialogOptions

Returns: `Promise<string | null>`

A promise resolving to the selected path.

[Go to TOC](#)

event

The event system allows you to emit events to the backend and listen to events from it.

This package is also accessible with `window.__TAURI__.event` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

Enumerations

TauriEvent

Since: 1.1.0

Enumeration Members

Name	Type	Defined in
CHECK_UPDATE	<code>tauri://update</code>	event.ts:34
DOWNLOAD_PROGRESS	<code>tauri://update-download-progress</code>	event.ts:38
INSTALL_UPDATE	<code>tauri://update-install</code>	event.ts:36
MENU	<code>tauri://menu</code>	event.ts:33
STATUS_UPDATE	<code>tauri://update-status</code>	event.ts:37
UPDATE_AVAILABLE	<code>tauri://update-available</code>	event.ts:35
WINDOW_BLUR	<code>tauri://blur</code>	event.ts:27
WINDOW_CLOSE_REQUESTED	<code>tauri://close-requested</code>	event.ts:23
WINDOW_CREATED	<code>tauri://window-created</code>	event.ts:24
WINDOW_DESTROYED	<code>tauri://destroyed</code>	event.ts:25
WINDOW_FILE_DROP	<code>tauri://file-drop</code>	event.ts:30
WINDOW_FILE_DROP_CANCELLED	<code>tauri://file-drop-cancelled</code>	event.ts:32
WINDOW_FILE_DROP_HOVER	<code>tauri://file-drop-hover</code>	event.ts:31
WINDOW_FOCUS	<code>tauri://focus</code>	event.ts:26
WINDOW_MOVED	<code>tauri://move</code>	event.ts:22
WINDOW_RESIZED	<code>tauri://resize</code>	event.ts:21
WINDOW_SCALE_FACTOR_CHANGED	<code>tauri://scale-change</code>	event.ts:28
WINDOW_THEME_CHANGED	<code>tauri://theme-changed</code>	event.ts:29

Interfaces

Event<T>

Type parameters

- `T`

Properties

`event`

event: string

Event name

Defined in: [helpers/event.ts:12](#)

`id`

id: number

Event identifier used to unlisten

Defined in: [helpers/event.ts:16](#)

`payload`

payload: T

Event payload

Defined in: [helpers/event.ts:18](#)

`windowLabel`

windowLabel: string

The label of the window that emitted this event.

Defined in: [helpers/event.ts:14](#)

Type Aliases

EventCallback<T>

```
EventCallback< T >: ( event : Event < T >) => void
```

Type parameters

- `T`

Type declaration

```
( event : Event < T >): void
```

Parameters

Name	Type
event	Event < T >

Returns: `void`

Defined in: [helpers/event.ts:21](#)

EventName

```
EventName: TauriEvent | string
```

Defined in: [event.ts:15](#)

UnlistenFn

```
UnlistenFn: () => void
```

Type declaration

```
(): void
```

Returns: void**Defined in:** helpers/event.ts:23

Functions

emit

```
emit( event : string , payload? : unknown ): Promise < void >
```

Emits an event to the backend.

Example

```
import { emit } from '@tauri-apps/api/event';
await emit('frontend-loaded', { loggedIn: true, token: 'authToken' });
```

Since: 1.0.0

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
payload?	unknown	-

Returns: Promise < void >

listen

```
listen< T >( event : string , handler: EventCallback < T > ): Promise < UnlistenFn >
```

Listen to an event from the backend.

Example

```
import { listen } from '@tauri-apps/api/event';
const unlisten = await listen<string>('error', (event) => {
  console.log(`Got error in window ${event.windowLabel}, payload: ${event.payload}`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.0

Type parameters

- `T`

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>handler</code>	<code>EventCallback < T ></code>	Event handler callback.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

once

```
once< T >( event: string, handler: EventCallback < T >): Promise < UnlistenFn >
```

Listen to an one-off event from the backend.

Example

```
import { once } from '@tauri-apps/api/event';
interface LoadedPayload {
    loggedIn: boolean,
    token: string
}
const unlisten = await once<LoadedPayload>('loaded', (event) => {
    console.log(`App is loaded, loggedIn: ${event.payload.loggedIn}, token: ${event.payload.token}`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.0

Type parameters

- `T`

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
handler	EventCallback < T >	-

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

fs

Access the file system.

This package is also accessible with `window.__TAURI__.fs` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.fs` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      fs: {
        all: true, // enable all FS APIs
        readFile: true,
        writeFile: true,
        readDir: true,
        copyFile: true,
        createDir: true,
        removeDir: true,
        removeFile: true,
        renameFile: true,
        exists: true
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Security

This module prevents path traversal, not allowing absolute paths or parent dir components (i.e. `/usr/path/to/file` or `../path/to/file` paths are not allowed). Paths accessed with this API must be relative to one of the [base directories](#) so if you need access to arbitrary filesystem paths, you must write such logic on the core layer instead.

The API has a scope configuration that forces you to restrict the paths that can be accessed using glob patterns.

The scope configuration is an array of glob patterns describing folder paths that are allowed. For instance, this scope configuration only allows accessing files on the *databases* folder of the [\\$APPDATA](#) directory:

```
{
  tauri: {
    allowlist: {
      fs: {
        scope: ['$APPDATA/databases/*']
      }
    }
  }
}
```

Notice the use of the `$APPDATA` variable. The value is injected at runtime, resolving to the app data directory. The available variables are: `$APPCONFIG`, `$APPDATA`, `$APPLOCALDATA`, `$APPCACHE`, `$APPLOG`, `$AUDIO`, `$CACHE`, `$CONFIG`, `$DATA`, `$LOCALDATA`, `$DESKTOP`, `$DOCUMENT`, `$DOWNLOAD`, `$EXE`, `$FONT`, `$HOME`, `$PICTURE`, `$PUBLIC`, `$RUNTIME`, `$TEMPLATE`, `$VIDEO`, `$RESOURCE`, `$APP`, `$LOG`, `$TEMP`.

Trying to execute any API with a URL not configured on the scope results in a promise rejection due to denied access.

Note that this scope applies to **all** APIs on this module.

References

Dir

Renames and re-exports [BaseDirectory](#)

writeFile

Renames and re-exports [writeTextFile](#)

Enumerations

BaseDirectory

Since: 1.0.0

Enumeration Members

Name	Type	Defined in
App	18	fs.ts:98
AppCache	24	fs.ts:104
AppConfig	21	fs.ts:101
AppData	22	fs.ts:102
AppLocalData	23	fs.ts:103
AppLog	25	fs.ts:105
Audio	1	fs.ts:81
Cache	2	fs.ts:82
Config	3	fs.ts:83
Data	4	fs.ts:84
Desktop	6	fs.ts:86
Document	7	fs.ts:87
Download	8	fs.ts:88

Name	Type	Defined in
Executable	9	fs.ts:89
Font	10	fs.ts:90
Home	11	fs.ts:91
LocalData	5	fs.ts:85
Log	19	fs.ts:99
Picture	12	fs.ts:92
Public	13	fs.ts:93
Resource	17	fs.ts:97
Runtime	14	fs.ts:94
Temp	20	fs.ts:100
Template	15	fs.ts:95
Video	16	fs.ts:96

Interfaces

FileEntry

Since: 1.0.0

Properties

children

Optional **children**: FileEntry []

Children of this entry if it's a directory; null otherwise

Defined in: [fs.ts:161](#)

name

Optional **name**: string

Name of the directory/file can be null if the path terminates with ..

Defined in: [fs.ts:159](#)

path**path:** string**Defined in:** [fs.ts:154](#)**FsBinaryFileOption**

Options object used to write a binary data to a file.

Since: 1.0.0**Properties****contents****contents:** BinaryFileContents

The byte array contents.

Defined in: [fs.ts:147](#)**path****path:** string

Path to the file to write.

Defined in: [fs.ts:145](#)**FsDirOptions****Since:** 1.0.0**Properties****dir**

Optional **dir:** BaseDirectory

Defined in: [fs.ts:120](#)

recursive

Optional **recursive**: boolean

Defined in: [fs.ts:121](#)

FsOptions

Since: 1.0.0

Properties

dir

Optional **dir**: BaseDirectory

Defined in: [fs.ts:112](#)

FsTextFileOption

Options object used to write a UTF-8 string to a file.

Since: 1.0.0

Properties

contents

contents: string

The UTF-8 string to write to the file.

Defined in: [fs.ts:133](#)

path

path: string

Path to the file to write.

Defined in: [fs.ts:131](#)

Type Aliases

BinaryFileContents

BinaryFileContents: Iterable < number > | ArrayLike < number > | ArrayBuffer

Defined in: [fs.ts:136](#)

Functions

copyFile

copyFile(source: string, destination: string, options?: FsOptions): Promise < void >

Copies a file to a destination.

Example

```
import { copyFile, BaseDirectory } from '@tauri-apps/api/fs';
// Copy the `$APP CONFIG/app.conf` file to `$APP CONFIG/app.conf.bk`
await copyFile('app.conf', 'app.conf.bk', { dir: BaseDirectory.AppConfig });
```

Since: 1.0.0

Parameters

Name	Type
source	string
destination	string
options	FsOptions

Returns: Promise < void >

A promise indicating the success or failure of the operation.

createDir

createDir(dir: string, options?: FsDirOptions): Promise < void >

Creates a directory. If one of the path's parent components doesn't exist and the `recursive` option isn't set to true, the promise will be rejected.

Example

```
import { createDir, BaseDirectory } from '@tauri-apps/api/fs';
// Create the `$APPDATA/users` directory
await createDir('users', { dir: BaseDirectory.AppData, recursive: true });
```

Since: 1.0.0**Parameters**

Name	Type
dir	string
options	FsDirOptions

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

exists

```
exists( path: string, options?: FsOptions ): Promise < boolean >
```

Check if a path exists.

Example

```
import { exists, BaseDirectory } from '@tauri-apps/api/fs';
// Check if the `$APPDATA/avatar.png` file exists
await exists('avatar.png', { dir: BaseDirectory.AppData });
```

Since: 1.1.0**Parameters**

Name	Type
path	string
options	FsOptions

Returns: `Promise < boolean >`**readBinaryFile**

```
readBinaryFile( filePath: string, options?: FsOptions ): Promise < Uint8Array >
```

Reads a file as byte array.

Example

```
import { readBinaryFile, BaseDirectory } from '@tauri-apps/api/fs';
// Read the image file in the `$RESOURCEDIR/avatar.png` path
const contents = await readBinaryFile('avatar.png', { dir: BaseDirectory.Resource });
};
```

Since: 1.0.0

Parameters

Name	Type
filePath	string
options	FsOptions

Returns: Promise < Uint8Array >

readDir

```
readDir( dir: string, options?: FsDirOptions ): Promise < FileEntry []>
```

List directory files.

Example

```
import { readDir, BaseDirectory } from '@tauri-apps/api/fs';
// Reads the `$APPDATA/users` directory recursively
const entries = await readDir('users', { dir: BaseDirectory.AppData, recursive: true });

function processEntries(entries) {
    for (const entry of entries) {
        console.log(`Entry: ${entry.path}`);
        if (entry.children) {
            processEntries(entry.children)
        }
    }
}
```

Since: 1.0.0

Parameters

Name	Type
dir	string
options	FsDirOptions

Returns: `Promise < FileEntry []>`**readTextFile**`readTextFile(filePath: string, options?: FsOptions): Promise < string >`

Reads a file as an UTF-8 encoded string.

Example

```
import { readTextFile, BaseDirectory } from '@tauri-apps/api/fs';
// Read the text file in the `$APP CONFIG/app.conf` path
const contents = await readTextFile('app.conf', { dir: BaseDirectory.AppConfig });
```

Since: 1.0.0**Parameters**

Name	Type
<code>filePath</code>	<code>string</code>
<code>options</code>	<code>FsOptions</code>

Returns: `Promise < string >`**removeDir**`removeDir(dir: string, options?: FsDirOptions): Promise < void >`

Removes a directory. If the directory is not empty and the `recursive` option isn't set to true, the promise will be rejected.

Example

```
import { removeDir, BaseDirectory } from '@tauri-apps/api/fs';
// Remove the directory `$APPDATA/users`
await removeDir('users', { dir: BaseDirectory.AppData });
```

Since: 1.0.0**Parameters**

Name	Type
<code>dir</code>	<code>string</code>
<code>options</code>	<code>FsDirOptions</code>

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

removeFile

```
removeFile( file : string , options? : FsOptions ): Promise < void >
```

Removes a file.

Example

```
import { removeFile, BaseDirectory } from '@tauri-apps/api/fs';
// Remove the `$APPConfig/app.conf` file
await removeFile('app.conf', { dir: BaseDirectory.AppConfig });
```

Since: 1.0.0

Parameters

Name	Type
<code>file</code>	<code>string</code>
<code>options</code>	<code>FsOptions</code>

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

renameFile

```
renameFile( oldPath : string , newPath : string , options? : FsOptions ): Promise < void >
```

Renames a file.

Example

```
import { renameFile, BaseDirectory } from '@tauri-apps/api/fs';
// Rename the `$APPDATA/avatar.png` file
await renameFile('avatar.png', 'deleted.png', { dir: BaseDirectory.AppData });
```

Since: 1.0.0

Parameters

Name	Type
oldPath	string
newPath	string
options	FsOptions

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

writeBinaryFile

```
writeBinaryFile( path: string , contents: BinaryFileContents , options?: FsOptions ): Promise < void >
```

Writes a byte array content to a file.

Example

```
import { writeBinaryFile, BaseDirectory } from '@tauri-apps/api/fs';
// Write a binary file to the `'$APPDATA/avatar.png` path
await writeBinaryFile('avatar.png', new Uint8Array([]), { dir:
BaseDirectory.AppData });
```

Since: 1.0.0

Parameters

Name	Type	Description
path	string	-
contents	BinaryFileContents	-
options?	FsOptions	Configuration object.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
writeBinaryFile( file: FsBinaryFileOption , options?: FsOptions ): Promise < void >
```

Writes a byte array content to a file.

Example

```
import { writeBinaryFile, BaseDirectory } from '@tauri-apps/api/fs';
// Write a binary file to the `$APPDATA/avatar.png` path
await writeBinaryFile({ path: 'avatar.png', contents: new Uint8Array([]) }, { dir: BaseDirectory.AppData });
```

Since: 1.0.0

Parameters

Name	Type	Description
file	FsBinaryFileOption	The object containing the file path and contents.
options?	FsOptions	Configuration object.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

writeTextFile

```
writeTextFile( path: string, contents: string, options?: FsOptions ): Promise < void >
```

Writes a UTF-8 text file.

Example

```
import { writeTextFile, BaseDirectory } from '@tauri-apps/api/fs';
// Write a text file to the `$APPCONFIG/app.conf` path
await writeTextFile('app.conf', 'file contents', { dir: BaseDirectory.AppConfig });
```

Since: 1.0.0

Parameters

Name	Type
path	string
contents	string
options?	FsOptions

Returns: Promise < void >

```
writeTextFile( file: FsTextFileOption, options?: FsOptions ): Promise < void >
```

Writes a UTF-8 text file.

Example

```
import { writeTextFile, BaseDirectory } from '@tauri-apps/api/fs';
// Write a text file to the `$APP CONFIG/app.conf` path
await writeTextFile({ path: 'app.conf', contents: 'file contents' }, { dir:
BaseDirectory.AppConfig });
```

Since: 1.0.0

Parameters

Name	Type
file	FsTextFileOption
options?	FsOptions

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

[Go to TOC](#)

globalShortcut

Register global shortcuts.

This package is also accessible with `window.__TAURI__.globalShortcut` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.globalShortcut` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      globalShortcut: {
        all: true // enable all global shortcut APIs
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Type Aliases

`ShortcutHandler`

ShortcutHandler: (`shortcut`: string) => void

Type declaration

```
(shortcut: string): void
```

Parameters

Name	Type
<code>shortcut</code>	string

Returns: void

Defined in: [globalShortcut.ts:29](#)

Functions

isRegistered

```
isRegistered( shortcut: string ): Promise < boolean >
```

Determines whether the given shortcut is registered by this application or not.

Example

```
import { isRegistered } from '@tauri-apps/api/globalShortcut';
const isRegistered = await isRegistered('CommandOrControl+P');
```

Since: 1.0.0

Parameters

Name	Type	Description
short- cut	string	Array of shortcut definitions, modifiers and key separated by + e.g. CmdOrControl+Q

Returns: `Promise < boolean >`

register

```
register( shortcut: string , handler: ShortcutHandler ): Promise < void >
```

Register a global shortcut.

Example

```
import { register } from '@tauri-apps/api/globalShortcut';
await register('CommandOrControl+Shift+C', () => {
  console.log('Shortcut triggered');
});
```

Since: 1.0.0

Parameters

Name	Type	Description
short- cut	string	Shortcut definition, modifiers and key separated by + e.g. CmdOrControl+Q
handler	ShortcutHandler	Shortcut handler callback - takes the triggered shortcut as argument

Returns: `Promise < void >`

registerAll

`registerAll(shortcuts : string [], handler : ShortcutHandler): Promise < void >`

Register a collection of global shortcuts.

Example

```
import { registerAll } from '@tauri-apps/api/globalShortcut';
await registerAll(['CommandOrControl+Shift+C', 'Ctrl+Alt+F12'], (shortcut) => {
  console.log(`Shortcut ${shortcut} triggered`);
});
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>shortcuts</code>	<code>string []</code>	Array of shortcut definitions, modifiers and key separated by + e.g. CmdOrControl+Q
<code>handler</code>	<code>ShortcutHandler</code>	Shortcut handler callback - takes the triggered shortcut as argument

Returns: `Promise < void >`

unregister

`unregister(shortcut : string): Promise < void >`

Unregister a global shortcut.

Example

```
import { unregister } from '@tauri-apps/api/globalShortcut';
await unregister('CmdOrControl+Space');
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>shortcut</code>	<code>string</code>	shortcut definition, modifiers and key separated by + e.g. CmdOrControl+Q

Returns: `Promise < void >`

unregisterAll

unregisterAll(): Promise < void >

Unregisters all shortcuts registered by the application.

Example

```
import { unregisterAll } from '@tauri-apps/api/globalShortcut';
await unregisterAll();
```

Since: 1.0.0

Returns: `Promise < void >`

[Go to TOC](#)

http

Access the HTTP client written in Rust.

This package is also accessible with `window.__TAURI__.http` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be allowlisted on `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      http: {
        all: true, // enable all http APIs
        request: true // enable HTTP request API
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Security

This API has a scope configuration that forces you to restrict the URLs and paths that can be accessed using glob patterns.

For instance, this scope configuration only allows making HTTP requests to the GitHub API for the `tauri-apps` organization:

```
{
  tauri: {
    allowlist: {
      http: {
        scope: [https://api.github.com/repos/tauri-apps/*]
      }
    }
  }
}
```

Trying to execute any API with a URL not configured on the scope results in a promise rejection due to denied access.

Enumerations

ResponseType

Since: 1.0.0

Enumeration Members

Name	Type	Defined in
Binary	3	http.ts:74
JSON	1	http.ts:72
Text	2	http.ts:73

Classes

Body

The body object to be used on POST and PUT requests.

Since: 1.0.0

Properties

payload

payload: unknown

Defined in: [http.ts:95](#)

type

type: string

Defined in: [http.ts:94](#)

Methods

bytes

```
Static bytes( bytes : Iterable < number > | ArrayBuffer | ArrayLike < number >): Body
```

Creates a new byte array body.

Example

```
import { Body } from '@tauri-apps/api/http
Body.bytes(new Uint8Array([1, 2, 3]));
```

Parameters

Name	Type	Description
bytes	Iterable<number> ArrayBuffer ArrayLike<number>	The body byte array.

Returns: Body

The body object ready to be used on the POST and PUT requests.

form

```
Static form( data : Record<string, Part> | FormData ): Body
```

Creates a new form data body. The form data is an object where each key is the entry name, and the value is either a string or a file object.

By default it sets the `application/x-www-form-urlencoded` Content-Type header, but you can set it to `multipart/form-data` if the Cargo feature `http-multipart` is enabled.

Note that a file path must be allowed in the `fs` allowlist scope.

Example

```
import { Body } from '@tauri-apps/api/http'
const body = Body.form({
  key: 'value',
  image: {
    file: '/path/to/file', // either a path or an array buffer of the file
    contents,
    mime: 'image/jpeg', // optional
    fileName: 'image.jpg' // optional
  }
});

// alternatively, use a FormData:
const form = new FormData();
form.append('key', 'value');
form.append('image', {
  file: '/path/to/file',
  mime: 'image/jpeg',
  fileName: 'image.jpg'
});
const formBody = Body.form(form);
```

Parameters

Name	Type	Description
data	Record<string, Part> FormData	The body data.

Returns: Body

The body object ready to be used on the POST and PUT requests.

json

```
Static json( data : Record< any , any >): Body
```

Creates a new JSON body.

Example

```
import { Body } from '@tauri-apps/api/http
Body.json({
  registered: true,
  name: 'tauri'
});
```

Parameters

Name	Type	Description
data	Record< any , any >	The body JSON object.

Returns: `Body`

The body object ready to be used on the POST and PUT requests.

text

```
Static text( value : string ): Body
```

Creates a new UTF-8 string body.

Example

```
import { Body } from '@tauri-apps/api/http
Body.text('The body content as a string');
```

Parameters

Name	Type	Description
value	string	The body string.

Returns: `Body`

The body object ready to be used on the POST and PUT requests.

Client

Since: 1.0.0

Properties

`id`

`id: number`

Defined in: [http.ts:307](#)

Methods

`delete`

`delete< T >(url: string, options?: RequestOptions): Promise < Response < T >>`

Makes a DELETE request.

Example

```
import { getClient } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.delete('http://localhost:3003/users/1');
```

Type parameters

- `T`

Parameters

Name	Type
<code>url</code>	<code>string</code>
<code>options?</code>	<code>RequestOptions</code>

Returns: `Promise < Response < T >>`

`drop`

`drop(): Promise < void >`

Drops the client instance.

Example

```
import { getClient } from '@tauri-apps/api/http';
const client = await getClient();
await client.drop();
```

Returns: `Promise < void >``get`

```
get< T >( url: string, options?: RequestOptions ): Promise < Response < T >>
```

Makes a GET request.

Example

```
import { getClient, ResponseType } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.get('http://localhost:3003/users', {
  timeout: 30,
  // the expected response type
  responseType: ResponseType.JSON
});
```

Type parameters

- `T`

Parameters

Name	Type
<code>url</code>	<code>string</code>
<code>options?</code>	<code>RequestOptions</code>

Returns: `Promise < Response < T >>``patch`

```
patch< T >( url: string, options?: RequestOptions ): Promise < Response < T >>
```

Makes a PATCH request.

Example

```
import { getClient, Body } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.patch('http://localhost:3003/users/1', {
  body: Body.json({ email: 'contact@tauri.app' })
});
```

Type parameters

- T

Parameters

Name	Type
url	string
options?	RequestOptions

Returns: Promise < Response < T >>

post

```
post< T >( url: string, body?: Body, options?: RequestOptions ): Promise < Response < T >>
```

Makes a POST request.

Example

```
import { getClient, Body, ResponseType } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.post('http://localhost:3003/users', {
  body: Body.json({
    name: 'tauri',
    password: 'awesome'
  }),
  // in this case the server returns a simple string
  responseType: ResponseType.Text,
});
```

Type parameters

- T

Parameters

Name	Type
url	string
body?	Body
options?	RequestOptions

Returns: `Promise < Response < T >>``put`

```
put< T >( url: string, body?: Body, options?: RequestOptions ): Promise < Response < T >>
```

Makes a PUT request.

Example

```
import { getClient, Body } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.put('http://localhost:3003/users/1', {
  body: Body.form({
    file: {
      file: '/home/tauri/avatar.png',
      mime: 'image/png',
      fileName: 'avatar.png'
    }
  })
});
```

Type parameters

- `T`

Parameters

Name	Type
<code>url</code>	<code>string</code>
<code>body?</code>	<code>Body</code>
<code>options?</code>	<code>RequestOptions</code>

Returns: `Promise < Response < T >>``request`

```
request< T >( options: HttpOptions ): Promise < Response < T >>
```

Makes an HTTP request.

Example

```
import { getClient } from '@tauri-apps/api/http';
const client = await getClient();
const response = await client.request({
```

```
    method: 'GET',
    url: 'http://localhost:3003/users',
});
```

Type parameters

- `T`

Parameters

Name	Type
<code>options</code>	<code>HttpOptions</code>

Returns: `Promise < Response < T >>`

`Response<T>`

Response object.

Since: 1.0.0

Type parameters

- `T`

Properties

`data`

```
data: T
```

The response data.

Defined in: [http.ts:290](#)

`headers`

```
headers: Record < string , string >
```

The response headers.

Defined in: [http.ts:286](#)

ok**ok:** boolean

A boolean indicating whether the response was successful (status in the range 200–299) or not.

Defined in: [http.ts:284](#)

rawHeaders**rawHeaders:** Record < string , string []>

The response raw headers.

Defined in: [http.ts:288](#)

status**status:** number

The response status code.

Defined in: [http.ts:282](#)

url**url:** string

The request URL.

Defined in: [http.ts:280](#)

Interfaces

ClientOptions

Since: 1.0.0

Properties

`connectTimeout`

Optional **connectTimeout**: number | Duration

Defines the maximum number of redirects the client should follow. If set to 0, no redirects will be followed.

Defined in: [http.ts:65](#)

`maxRedirects`

Optional **maxRedirects**: number

Defined in: [http.ts:60](#)

Duration

Since: 1.0.0

Properties

`nanos`

nanos: number

Defined in: [http.ts:53](#)

`secs`

secs: number

Defined in: [http.ts:52](#)

FilePart<T>

Since: 1.0.0

Type parameters

- `T`

Properties

`file`

file: string | T

Defined in: [http.ts:81](#)

`fileName`

Optional **fileName**: string

Defined in: [http.ts:83](#)

`mime`

Optional **mime**: string

Defined in: [http.ts:82](#)

HttpOptions

Options object sent to the backend.

Since: 1.0.0

Properties

`body`

Optional **body**: Body

Defined in: [http.ts:254](#)

`headers`

Optional **headers**: Record < string , any >

Defined in: [http.ts:252](#)

`method`

method: `HttpVerb`

Defined in: [http.ts:250](#)

`query`

Optional **query:** `Record < string , any >`

Defined in: [http.ts:253](#)

`responseType`

Optional **responseType:** `ResponseType`

Defined in: [http.ts:256](#)

`timeout`

Optional **timeout:** `number | Duration`

Defined in: [http.ts:255](#)

`url`

url: `string`

Defined in: [http.ts:251](#)

Type Aliases

`FetchOptions`

FetchOptions: `Omit < HttpOptions , url >`

Options for the `fetch` API.

Defined in: [http.ts:262](#)

HttpVerb

HttpVerb: GET | POST | PUT | DELETE | PATCH | HEAD | OPTIONS | CONNECT | TRACE

The request HTTP verb.

Defined in: [http.ts:233](#)

Part

Part: string | Uint8Array | FilePart < Uint8Array >**Defined in:** [http.ts:86](#)

RequestOptions

RequestOptions: Omit < HttpOptions , method | url >

Request options.

Defined in: [http.ts:260](#)

Functions

fetch

fetch< T >(url : string , options? : FetchOptions): Promise < Response < T >>

Perform an HTTP request using the default client.

Example

```
import { fetch } from '@tauri-apps/api/http';
const response = await fetch('http://localhost:3003/users/2', {
    method: 'GET',
    timeout: 30,
});
```

Type parameters

- T

Parameters

Name	Type
url	string
options?	FetchOptions

Returns: Promise < Response < T >>

getClient

```
getClient( options? : ClientOptions ): Promise < Client >
```

Creates a new client using the specified options.

Example

```
import { getClient } from '@tauri-apps/api/http';
const client = await getClient();
```

Since: 1.0.0

Parameters

Name	Type	Description
options?	ClientOptions	Client configuration.

Returns: Promise < Client >

A promise resolving to the client instance.

[Go to TOC](#)

mocks

Functions

clearMocks

clearMocks(): void

Clears mocked functions/data injected by the other functions in this module. When using a test runner that doesn't provide a fresh window object for each test, calling this function will reset tauri specific properties.

Example

```
import { mockWindows, clearMocks } from '@tauri-apps/api/mocks

afterEach(() => {
    clearMocks()
})

test('mocked windows', () => {
    mockWindows(main, second, third);

    expect(window).toHaveProperty('__TAURI_METADATA__')
})

test('no mocked windows', () => {
    expect(window).not.toHaveProperty('__TAURI_METADATA__')
})
```

Since: 1.0.0

Returns: `void`

mockIPC

mockIPC(cb : fn): void

Intercepts all IPC requests with the given mock handler.

This function can be used when testing tauri frontend applications or when running the frontend in a Node.js context during static site generation.

Examples

Testing setup using vitest:

```
import { mockIPC, clearMocks } from '@tauri-apps/api/mocks'
import { invoke } from '@tauri-apps/api/tauri'

afterEach(() => {
  clearMocks()
})

test(mocked command, () => {
  mockIPC((cmd, args) => {
    switch (.html) {
      case add:
        return (args.a as number) + (args.b as number);
      default:
        break;
    }
  });
  expect(invoke('add', { a: 12, b: 15 })).resolves.toBe(27);
})
```

The callback function can also return a Promise:

```
import { mockIPC, clearMocks } from '@tauri-apps/api/mocks'
import { invoke } from '@tauri-apps/api/tauri'

afterEach(() => {
  clearMocks()
})

test(mocked command, () => {
  mockIPC((cmd, args) => {
    if(cmd === get_data) {
      return fetch('https://example.com/data.json')
        .then((response) => response.json())
    }
  });
  expect(invoke('get_data')).resolves.toBe({ foo: 'bar' });
})
```

Since: 1.0.0

Parameters

Name	Type
cb	(cmd: string, args: Record<string, unknown>) => any

Returns: void

mockWindows

```
mockWindows( current : string , ... additionalWindows : string []): void
```

Mocks one or many window labels. In non-tauri context it is required to call this function *before* using the `@tauri-apps/api/window` module.

This function only mocks the *presence* of windows, window properties (e.g. width and height) can be mocked like regular IPC calls using the `mockIPC` function.

Examples

```
import { mockWindows } from '@tauri-apps/api/mocks';
import { getCurrent } from '@tauri-apps/api/window';

mockWindows(main, second, third);

const win = getCurrent();

win.label // main
```

```
import { mockWindows } from '@tauri-apps/api/mocks';

mockWindows(main, second, third);

mockIPC((cmd, args) => {
  if (cmd === tauri) {
    if (
      args.__tauriModule === Window &&
      args.message?.cmd === manage &&
      args.message?.data?.cmd?.type === close
    ) {
      console.log('closing window!');
    }
  }
});

const { getCurrent } = await import('@tauri-apps/api/window');

const win = getCurrent();
await win.close(); // this will cause the mocked IPC handler to log to the
console.
```

Since: 1.0.0

Parameters

Name	Type	Description
current	string	Label of window this JavaScript context is running in.
...additionalWindows	string []	Label of additional windows the app has.

Returns: void

[Go to TOC](#)

notification

Send toast notifications (brief auto-expiring OS window element) to your user. Can also be used with the Notification Web API.

This package is also accessible with `window.__TAURI__.notification` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.notification` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      notification: {
        all: true // enable all notification APIs
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Interfaces

Options

Options to send a notification.

Since: 1.0.0

Properties

body

Optional **body**: string

Optional notification body.

Defined in: [notification.ts:38](#)

icon

Optional **icon**: string

Optional notification icon.

Defined in: notification.ts:40

`title`

title: string

Notification title.

Defined in: notification.ts:36

Type Aliases

`Permission`

Permission: granted | denied | default

Possible permission values.

Defined in: notification.ts:44

Functions

`isPermissionGranted`

isPermissionGranted(): Promise < boolean >

Checks if the permission to send notifications is granted.

Example

```
import { isPermissionGranted } from '@tauri-apps/api/notification';
const permissionGranted = await isPermissionGranted();
```

Since: 1.0.0

Returns: Promise < boolean >

`requestPermission`

requestPermission(): Promise < Permission >

Requests the permission to send notifications.

Example

```
import { isPermissionGranted, requestPermission } from '@tauri-
apps/api/notification';
let permissionGranted = await isPermissionGranted();
if (!permissionGranted) {
  const permission = await requestPermission();
  permissionGranted = permission === 'granted';
}
```

Since: 1.0.0

Returns: `Promise < Permission >`

A promise resolving to whether the user granted the permission or not.

sendNotification

```
sendNotification( options: string | Options ): void
```

Sends a notification to the user.

Example

```
import { isPermissionGranted, requestPermission, sendNotification } from '@tauri-
apps/api/notification';
let permissionGranted = await isPermissionGranted();
if (!permissionGranted) {
  const permission = await requestPermission();
  permissionGranted = permission === 'granted';
}
if (permissionGranted) {
  sendNotification('Tauri is awesome!');
  sendNotification({ title: 'TAURI', body: 'Tauri is awesome!' });
}
```

Since: 1.0.0

Parameters

Name	Type
<code>options</code>	<code>string Options</code>

Returns: `void`

[Go to TOC](#)

OS

Provides operating system-related utility methods and properties.

This package is also accessible with `window.__TAURI__.os` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.os` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      os: {
        all: true, // enable all Os APIs
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Type Aliases

Arch

Arch: x86 | x86_64 | arm | aarch64 | mips | mips64 | powerpc | powerpc64 | riscv64 | s390x | sparc64

Defined in: [os.ts:43](#)

OsType

OsType: Linux | Darwin | Windows_NT

Defined in: [os.ts:41](#)

Platform

Platform: linux | darwin | ios | freebsd | dragonfly | netbsd | openbsd | solaris | android | win32

Defined in: [os.ts:29](#)

Variables

EOL

```
Const EOL: \n | \r\n
```

The operating system-specific end-of-line marker.

- `\n` on POSIX
- `\r\n` on Windows

Since: 1.0.0

Defined in: [os.ts:63](#)

Functions

arch

```
arch(): Promise < Arch >
```

Returns the operating system CPU architecture for which the tauri app was compiled. Possible values are `'x86'`, `'x86_64'`, `'arm'`, `'aarch64'`, `'mips'`, `'mips64'`, `'powerpc'`, `'powerpc64'`, `'riscv64'`, `'s390x'`, `'sparc64'`.

Example

```
import { arch } from '@tauri-apps/api/os';
const archName = await arch();
```

Since: 1.0.0

Returns: `Promise < Arch >`

platform

```
platform(): Promise < Platform >
```

Returns a string identifying the operating system platform. The value is set at compile time. Possible values are `'linux'`, `'darwin'`, `'ios'`, `'freebsd'`, `'dragonfly'`, `'netbsd'`, `'openbsd'`, `'solaris'`, `'android'`, `'win32'`

Example

```
import { platform } from '@tauri-apps/api/os';
const platformName = await platform();
```

Since: 1.0.0

Returns: `Promise < Platform >`

tempdir

```
tempdir(): Promise < string >
```

Returns the operating system's default directory for temporary files as a string.

Example

```
import { tempdir } from '@tauri-apps/api/os';
const tempdirPath = await tempdir();
```

Since: 1.0.0

Returns: `Promise < string >`

type

```
type(): Promise < OsType >
```

Returns `'Linux'` on Linux, `'Darwin'` on macOS, and `'Windows_NT'` on Windows.

Example

```
import { type } from '@tauri-apps/api/os';
const osType = await type();
```

Since: 1.0.0

Returns: `Promise < OsType >`

version

```
version(): Promise < string >
```

Returns a string identifying the kernel version.

Example

```
import { version } from '@tauri-apps/api/os';
const osVersion = await version();
```

Since: 1.0.0

Returns: `Promise<string>`

[Go to TOC](#)

path

The path module provides utilities for working with file and directory paths.

This package is also accessible with `window.__TAURI__.path` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.path` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      path: {
        all: true, // enable all Path APIs
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

References

BaseDirectory

Re-exports [BaseDirectory](#)

Variables

delimiter

```
Const delimiter: ; | :
```

Provides the platform-specific path segment delimiter:

- `;` on Windows
- `:` on POSIX

Since: 1.0.0

Defined in: [path.ts:660](#)

sep

```
Const sep: \\ | /
```

Provides the platform-specific path segment separator:

- `\` on Windows
- `/` on POSIX

Since: 1.0.0

Defined in: [path.ts:651](#)

Functions

appCacheDir

```
appCacheDir(): Promise < string >
```

Returns the path to the suggested directory for your app's cache files. Resolves to `${cacheDir}/${bundleIdentifier}`, where `bundleIdentifier` is the value `tauri.bundle.identifier` is configured in `tauri.conf.json`.

Example

```
import { appCacheDir } from '@tauri-apps/api/path';
const appCacheDirPath = await appCacheDir();
```

Since: 1.2.0

Returns: `Promise < string >`

appConfigDir

```
appConfigDir(): Promise < string >
```

Returns the path to the suggested directory for your app's config files. Resolves to `${configDir}/${bundleIdentifier}`, where `bundleIdentifier` is the value `tauri.bundle.identifier` is configured in `tauri.conf.json`.

Example

```
import { appConfigDir } from '@tauri-apps/api/path';
const appConfigDirPath = await appConfigDir();
```

Since: 1.2.0

Returns: `Promise < string >`

appDataDir

appDataDir(): Promise < string >

Returns the path to the suggested directory for your app's data files. Resolves to `${dataDir}/${bundleIdentifier}`, where `bundleIdentifier` is the value `tauri.bundle.identifier` is configured in `tauri.conf.json`.

Example

```
import { appDataDir } from '@tauri-apps/api/path';
const appDataDirPath = await appDataDir();
```

Since: 1.2.0

Returns: `Promise < string >`

appDir

appDir(): Promise < string >

Returns the path to the suggested directory for your app config files.

Deprecated

since 1.2.0: Will be removed in 2.0.0. Use [appConfigDir](#) or [appDataDir](#) instead.

Since: 1.0.0

Returns: `Promise < string >`

appLocalDataDir

appLocalDataDir(): Promise < string >

Returns the path to the suggested directory for your app's local data files. Resolves to `${localDataDir}/${bundleIdentifier}`, where `bundleIdentifier` is the value `tauri.bundle.identifier` is configured in `tauri.conf.json`.

Example

```
import { appLocalDataDir } from '@tauri-apps/api/path';
const appLocalDataDirPath = await appLocalDataDir();
```

Since: 1.2.0**Returns:** `Promise < string >`

appLogDir

`appLogDir(): Promise < string >`

Returns the path to the suggested directory for your app's log files.

Platform-specific

- **Linux:** Resolves to `${configDir}/${bundleIdentifier}/logs`.
- **macOS:** Resolves to `${homeDir}/Library/Logs/${bundleIdentifier}`
- **Windows:** Resolves to `${configDir}/${bundleIdentifier}/logs`.

Example

```
import { appLogDir } from '@tauri-apps/api/path';
const appLogDirPath = await appLogDir();
```

Since: 1.2.0**Returns:** `Promise < string >`

audioDir

`audioDir(): Promise < string >`

Returns the path to the user's audio directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_MUSIC_DIR`.
- **macOS:** Resolves to `${HOME/Music}`.
- **Windows:** Resolves to `{FOLDERID_Music}`.

Example

```
import { audioDir } from '@tauri-apps/api/path';
const audioDirPath = await audioDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

basename

```
basename( path : string , ext? : string ): Promise < string >
```

Returns the last portion of a `path`. Trailing directory separators are ignored.

Example

```
import { basename, resolveResource } from '@tauri-apps/api/path';
const resourcePath = await resolveResource('app.conf');
const base = await basename(resourcePath);
assert(base === 'app');
```

Since: 1.0.0

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	-
<code>ext?</code>	<code>string</code>	An optional file extension to be removed from the returned path.

Returns: `Promise < string >`

cacheDir

```
cacheDir(): Promise < string >
```

Returns the path to the user's cache directory.

Platform-specific

- **Linux:** Resolves to `$XDG_CACHE_HOME` or `$HOME/.cache`.
- **macOS:** Resolves to `$HOME/Library/Caches`.
- **Windows:** Resolves to `{FOLDERID_LocalAppData}`.

Example

```
import { cacheDir } from '@tauri-apps/api/path';
const cacheDirPath = await cacheDir();
```

Since: 1.0.0

Returns: `Promise < string >`

configDir

```
configDir(): Promise < string >
```

Returns the path to the user's config directory.

Platform-specific

- **Linux:** Resolves to `$XDG_CONFIG_HOME` or `$HOME/.config`.
- **macOS:** Resolves to `$HOME/Library/Application Support`.
- **Windows:** Resolves to `{FOLDERID_RoamingAppData}`.

Example

```
import { configDir } from '@tauri-apps/api/path';
const configDirPath = await configDir();
```

Since: 1.0.0

Returns: `Promise < string >`

dataDir

```
dataDir(): Promise < string >
```

Returns the path to the user's data directory.

Platform-specific

- **Linux:** Resolves to `$XDG_DATA_HOME` or `$HOME/.local/share`.
- **macOS:** Resolves to `$HOME/Library/Application Support`.
- **Windows:** Resolves to `{FOLDERID_RoamingAppData}`.

Example

```
import { dataDir } from '@tauri-apps/api/path';
const dataDirPath = await dataDir();
```

Since: 1.0.0

Returns: `Promise < string >`

desktopDir

```
desktopDir(): Promise < string >
```

Returns the path to the user's desktop directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_DESKTOP_DIR`.
- **macOS:** Resolves to `$HOME/Desktop`.
- **Windows:** Resolves to `{FOLDERID/Desktop}`.

Example

```
import { desktopDir } from '@tauri-apps/api/path';
const desktopPath = await desktopDir();
```

Since: 1.0.0

Returns: `Promise<string>`

dirname

```
dirname( path: string ): Promise<string>
```

Returns the directory name of a `path`. Trailing directory separators are ignored.

Example

```
import { dirname, appDataDir } from '@tauri-apps/api/path';
const appDataDirPath = await appDataDir();
const dir = await dirname(appDataDirPath);
```

Since: 1.0.0

Parameters

Name	Type
<code>path</code>	<code>string</code>

Returns: `Promise<string>`

documentDir

```
documentDir(): Promise<string>
```

Returns the path to the user's document directory.

Example

```
import { documentDir } from '@tauri-apps/api/path';
const documentDirPath = await documentDir();
```

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_DOCUMENTS_DIR`.
- **macOS:** Resolves to `$HOME/Documents`.
- **Windows:** Resolves to `{FOLDERID_Documents}`.

Since: 1.0.0

Returns: `Promise < string >`

downloadDir

```
downloadDir(): Promise < string >
```

Returns the path to the user's download directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_DOWNLOAD_DIR`.
- **macOS:** Resolves to `$HOME/Downloads`.
- **Windows:** Resolves to `{FOLDERID_Downloads}`.

Example

```
import { downloadDir } from '@tauri-apps/api/path';
const downloadDirPath = await downloadDir();
```

Since: 1.0.0

Returns: `Promise < string >`

executableDir

```
executableDir(): Promise < string >
```

Returns the path to the user's executable directory.

Platform-specific

- **Linux:** Resolves to `$XDG_BIN_HOME/.../bin` or `$XDG_DATA_HOME/.../bin` or `$HOME/.local/bin`.
- **macOS:** Not supported.
- **Windows:** Not supported.

Example

```
import { executableDir } from '@tauri-apps/api/path';
const executableDirPath = await executableDir();
```

Since: 1.0.0

Returns: `Promise < string >`

extname

```
extname( path : string ): Promise < string >
```

Returns the extension of the `path`.

Example

```
import { extname, resolveResource } from '@tauri-apps/api/path';
const resourcePath = await resolveResource('app.conf');
const ext = await extname(resourcePath);
assert(ext === 'conf');
```

Since: 1.0.0

Parameters

Name	Type
<code>path</code>	<code>string</code>

Returns: `Promise < string >`

fontDir

```
fontDir(): Promise < string >
```

Returns the path to the user's font directory.

Platform-specific

- **Linux:** Resolves to `$XDG_DATA_HOME/fonts` or `$HOME/.local/share/fonts`.
- **macOS:** Resolves to `$HOME/Library/Fonts`.
- **Windows:** Not supported.

Example

```
import { fontDir } from '@tauri-apps/api/path';
const fontDirPath = await fontDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

homeDir

`homeDir(): Promise < string >`

Returns the path to the user's home directory.

Platform-specific

- **Linux:** Resolves to `$HOME`.
- **macOS:** Resolves to `$HOME`.
- **Windows:** Resolves to `{FOLDERID_Profile}`.

Example

```
import { homeDir } from '@tauri-apps/api/path';
const homeDirPath = await homeDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

isAbsolute

`isAbsolute(path: string): Promise < boolean >`

Returns whether the path is absolute or not.

Example

```
import { isAbsolute } from '@tauri-apps/api/path';
assert(await isAbsolute('/home/tauri'));
```

Since: 1.0.0

Parameters

Name	Type
<code>path</code>	<code>string</code>

Returns: `Promise < boolean >`

join

```
join(... paths : string []): Promise < string >
```

Joins all given `path` segments together using the platform-specific separator as a delimiter, then normalizes the resulting path.

Example

```
import { join, appDataDir } from '@tauri-apps/api/path';
const appDataDirPath = await appDataDir();
const path = await join(appDataDirPath, 'users', 'tauri', 'avatar.png');
```

Since: 1.0.0

Parameters

Name	Type
<code>...paths</code>	<code>string []</code>

Returns: `Promise < string >`

localDataDir

```
localDataDir(): Promise < string >
```

Returns the path to the user's local data directory.

Platform-specific

- **Linux:** Resolves to `$XDG_DATA_HOME` or `$HOME/.local/share`.
- **macOS:** Resolves to `$HOME/Library/Application Support`.
- **Windows:** Resolves to `{FOLDERID_LocalAppData}`.

Example

```
import { localDataDir } from '@tauri-apps/api/path';
const localDataDirPath = await localDataDir();
```

Since: 1.0.0

Returns: `Promise < string >`

logDir

logDir(): Promise < string >

Returns the path to the suggested log directory.

Deprecated

since 1.2.0: Will be removed in 2.0.0. Use [appLogDir](#) instead.

Since: 1.0.0

Returns: `Promise < string >`

normalize

normalize(path : string): Promise < string >

Normalizes the given `path`, resolving `'..'` and `'.'` segments and resolve symbolic links.

Example

```
import { normalize, appDataDir } from '@tauri-apps/api/path';
const appDataDirPath = await appDataDir();
const path = await normalize(appDataDirPath, '..', 'users', 'tauri',
  'avatar.png');
```

Since: 1.0.0

Parameters

Name	Type
<code>path</code>	<code>string</code>

Returns: `Promise < string >`

pictureDir

pictureDir(): Promise < string >

Returns the path to the user's picture directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_PICTURES_DIR`.
- **macOS:** Resolves to `$HOME/Pictures`.
- **Windows:** Resolves to `{FOLDERID_Pictures}`.

Example

```
import { pictureDir } from '@tauri-apps/api/path';
const pictureDirPath = await pictureDir();
```

Since: 1.0.0**Returns:** `Promise < string >`**publicDir**

```
publicDir(): Promise < string >
```

Returns the path to the user's public directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_PUBLICSHARE_DIR`.
- **macOS:** Resolves to `$HOME/Public`.
- **Windows:** Resolves to `{FOLDERID_Public}`.

Example

```
import { publicDir } from '@tauri-apps/api/path';
const publicDirPath = await publicDir();
```

Since: 1.0.0**Returns:** `Promise < string >`**resolve**

```
resolve(... paths : string[]): Promise < string >
```

Resolves a sequence of `paths` or `path` segments into an absolute path.

Example

```
import { resolve, appDataDir } from '@tauri-apps/api/path';
const appDataDirPath = await appDataDir();
const path = await resolve(appDataDirPath, '..', 'users', 'tauri', 'avatar.png');
```

Since: 1.0.0**Parameters**

Name	Type
...paths	string []

Returns: Promise < string >**resolveResource****resolveResource(** resourcePath : string **):** Promise < string >

Resolve the path to a resource file.

Example

```
import { resolveResource } from '@tauri-apps/api/path';
const resourcePath = await resolveResource('script.sh');
```

Since: 1.0.0**Parameters**

Name	Type	Description
resourcePath	string	The path to the resource. Must follow the same syntax as defined in <code>tauri.conf.json > tauri > bundle > resources</code> , i.e. keeping subfolders and parent dir components (<code>..</code>).

Returns: Promise < string >

The full path to the resource.

resourceDir**resourceDir():** Promise < string >Returns the path to the application's resource directory. To resolve a resource path, see the [[resolveResource | `resolveResource API`]].**Example**

```
import { resourceDir } from '@tauri-apps/api/path';
const resourceDirPath = await resourceDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

runtimeDir

`runtimeDir(): Promise < string >`

Returns the path to the user's runtime directory.

Platform-specific

- **Linux:** Resolves to `$XDG_RUNTIME_DIR`.
- **macOS:** Not supported.
- **Windows:** Not supported.

Example

```
import { runtimeDir } from '@tauri-apps/api/path';
const runtimeDirPath = await runtimeDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

templateDir

`templateDir(): Promise < string >`

Returns the path to the user's template directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` | XDG_TEMPLATES_DIR`.
- **macOS:** Not supported.
- **Windows:** Resolves to `{FOLDERID_Templates}`.

Example

```
import { templateDir } from '@tauri-apps/api/path';
const templateDirPath = await templateDir();
```

Since: 1.0.0**Returns:** `Promise < string >`

videoDir

videoDir(): `Promise < string >`

Returns the path to the user's video directory.

Platform-specific

- **Linux:** Resolves to `xdg-user-dirs` XDG_VIDEOS_DIR`.
- **macOS:** Resolves to `$HOME/Movies`.
- **Windows:** Resolves to `{FOLDERID_Videos}`.

Example

```
import { videoDir } from '@tauri-apps/api/path';
const videoDirPath = await videoDir();
```

Since: 1.0.0

Returns: `Promise < string >`

[Go to TOC](#)

process

Perform operations on the current process.

This package is also accessible with `window.__TAURI__.process` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

Functions

exit

```
exit( exitCode? : number ): Promise < void >
```

Exits immediately with the given `exitCode`.

Example

```
import { exit } from '@tauri-apps/api/process';
await exit(1);
```

Since: 1.0.0

Parameters

Name	Type	Default value	Description
<code>exitCode</code>	<code>number</code>	<code>0</code>	The exit code to use.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

relaunch

```
relaunch(): Promise < void >
```

Exits the current instance of the app then relaunches it.

Example

```
import { relaunch } from '@tauri-apps/api/process';
await relaunch();
```

Since: 1.0.0

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

[Go to TOC](#)

shell

Access the system shell. Allows you to spawn child processes and manage files and URLs using their default application.

This package is also accessible with `window.__TAURI__.shell` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.shell` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      shell: {
        all: true, // enable all shell APIs
        execute: true, // enable process spawn APIs
        sidecar: true, // enable spawning sidecars
        open: true // enable opening files/URLs using the default program
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Security

This API has a scope configuration that forces you to restrict the programs and arguments that can be used.

Restricting access to the `open` API

On the allowlist, `open: true` means that the `open` API can be used with any URL, as the argument is validated with the `^https?://` regex. You can change that regex by changing the boolean value to a string, e.g. `open: ^https://github.com/`.

Restricting access to the Command APIs

The `shell` allowlist object has a `scope` field that defines an array of CLIs that can be used. Each CLI is a configuration object `{ name: string, cmd: string, sidecar?: boolean, args?: boolean | Arg[] }`.

- `name`: the unique identifier of the command, passed to the [Command constructor](#). If it's a sidecar, this must be the value defined on `tauri.conf.json > tauri > bundle > externalBin`.
- `cmd`: the program that is executed on this configuration. If it's a sidecar, this value is ignored.
- `sidecar`: whether the object configures a sidecar or a system program.
- `args`: the arguments that can be passed to the program. By default no arguments are allowed.
 - `true` means that any argument list is allowed.
 - `false` means that no arguments are allowed.
 - otherwise an array can be configured. Each item is either a string representing the fixed argument value or a `{ validator: string }` that defines a regex validating the argument value.

Example scope configuration

CLI: `git commit -m the commit message`

Configuration:

```
{
  scope: {
    name: run-git-commit,
    cmd: git,
    args: [commit, -m, { validator: \\\$+ }]
  }
}
```

Usage:

```
import { Command } from '@tauri-apps/api/shell'
new Command('run-git-commit', ['commit', '-m', 'the commit message'])
```

Trying to execute any API with a program not configured on the scope results in a promise rejection due to denied access.

Classes

Child

Since: 1.1.0

Constructors

constructor

```
new Child( pid: number ): Child
```

Parameters

Name	Type
pid	number

Defined in: [shell.ts:323](#)

Properties

pid

```
pid: number
```

The child process `pid`.

Defined in: [shell.ts:321](#)

Methods

`kill`

`kill(): Promise < void >`

Kills the child process.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`write`

`write(data : string | Uint8Array): Promise < void >`

Writes `data` to the `stdin`.

Example

```
import { Command } from '@tauri-apps/api/shell';
const command = new Command('node');
const child = await command.spawn();
await child.write('message');
await child.write([0, 1, 2, 3, 4, 5]);
```

Parameters

Name	Type	Description
<code>data</code>	<code>string Uint8Array</code>	The message to write, either a string or a byte array.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

Command

The entry point for spawning child processes. It emits the `close` and `error` events.

Example

```
import { Command } from '@tauri-apps/api/shell';
const command = new Command('node');
command.on('close', data => {
```

```

    console.log(`command finished with code ${data.code} and signal ${data.signal}`);
};

command.on('error', error => console.error(`command error: ${error}`));
command.stdout.on('data', line => console.log(`command stdout: ${line}`));
command.stderr.on('data', line => console.log(`command stderr: ${line}`));

const child = await command.spawn();
console.log('pid:', child.pid);

```

Since: 1.1.0

Hierarchy

- `EventEmitter < close | error >`
 - **Command**

Constructors

`constructor`

```
new Command( program: string, args?: string | string[], options?: SpawnOptions ): Co
mmand
```

Creates a new `Command` instance.

Parameters

Name	Type	Default value	Description
<code>program</code>	<code>string</code>	<code>undefined</code>	The program name to execute. It must be configured on <code>tauri.conf.json > tauri > allowlist > shell > scope</code> .
<code>args</code>	<code>string string[]</code>	<code>[]</code>	Program arguments.
<code>options?</code>	<code>SpawnOptions</code>	<code>undefined</code>	Spawn options.

Overrides: `EventEmitter.constructor`**Defined in:** `shell.ts:411`

Properties

`stderr`

```
Readonly stderr: EventEmitter < data >
```

Event emitter for the `stderr`. Emits the `data` event.

Defined in: [shell.ts:401](#)

`stdout`

```
 Readonly stdout: EventEmitter<data>
```

Event emitter for the `stdout`. Emits the `data` event.

Defined in: [shell.ts:399](#)

Methods

`addListener`

```
addListener(eventName: error | close, listener: fn): Command
```

Alias for `emitter.on(eventName, listener)`.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command`

`execute`

```
execute(): Promise<ChildProcess>
```

Executes the command as a child process, waiting for it to finish and collecting all of its output.

Example

```
import { Command } from '@tauri-apps/api/shell';
const output = await new Command('echo', 'message').execute();
assert(output.code === 0);
assert(output.signal === null);
assert(output.stdout === 'message');
assert(output.stderr === '')
```

Returns: `Promise < ChildProcess >`

A promise resolving to the child process output.

`listenerCount`**listenerCount**(`eventName : error | close`): `number`

Returns the number of listeners listening to the event named `eventName`.

Since: 1.1.0**Parameters**

Name	Type
<code>eventName</code>	<code>error close</code>

Returns: `number``off`**off**(`eventName : error | close , listener : fn`): `Command`

Removes the all specified listener from the listener array for the event `eventName`. Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0**Parameters**

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command``on`**on**(`eventName : error | close , listener : fn`): `Command`

Adds the `listener` function to the end of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.0.0

Parameters

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command`

`once`

```
once(eventName: error | close, listener: fn): Command
```

Adds a **one-time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command`

`prependListener`

```
prependListener(eventName: error | close, listener: fn): Command
```

Adds the `listener` function to the *beginning* of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command`

`prependOnceListener`

```
prependOnceListener(eventName : error | close, listener : fn): Command
```

Adds a **one-time** `listener` function for the event named `eventName` to the *beginning* of the listeners array. The next time `eventName` is triggered, this listener is removed, and then invoked.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>error close</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `Command`

`removeAllListeners`

```
removeAllListeners(event? : error | close): Command
```

Removes all listeners, or those of the specified `eventName`.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
event?	error close

Returns: Command`removeListener`**removeListener**(eventName: error | close, listener: fn): CommandAlias for `emitter.off(eventName, listener)`.**Since:** 1.1.0**Parameters**

Name	Type
eventName	error close
listener	(... args: any []) => void

Returns: Command`spawn`**spawn()**: Promise < Child >

Executes the command as a child process, returning a handle to it.

Returns: Promise < Child >

A promise resolving to the child process handle.

`sidecar`Static **sidecar**(program: string, args?: string | string[], options?: SpawnOptions): Command

Creates a command to execute the given sidecar program.

Example

```
import { Command } from '@tauri-apps/api/shell';
const command = Command.sidecar('my-sidecar');
const output = await command.execute();
```

Parameters

Name	Type	Default value	Description
<code>program</code>	<code>string</code>	<code>undefined</code>	The program to execute. It must be configured on <code>tauri.conf.json > tauri > allowlist > shell > scope</code> .
<code>args</code>	<code>string string []</code>	<code>[]</code>	-
<code>options?</code>	<code>SpawnOptions</code>	<code>undefined</code>	-

Returns: `Command`

EventEmitter<E>

Since: 1.0.0

Type parameters

- `E extends string`

Hierarchy

- **EventEmitter**
 - `Command`

Constructors

`constructor`

```
new EventEmitter< E >(): EventEmitter < E >
```

Type parameters

- `E extends string`

Methods

`addListener`

```
addListener( eventName : E , listener : fn ): EventEmitter < E >
```

Alias for `emitter.on(eventName, listener)`.

Since: 1.1.0

Parameters

Name	Type
eventName	E
listener	(... args : any []) => void

Returns: EventEmitter < E >

listenerCount

listenerCount(eventName : E): numberReturns the number of listeners listening to the event named `eventName`.**Since:** 1.1.0**Parameters**

Name	Type
eventName	E

Returns: number

off

off(eventName : E , listener : fn): EventEmitter < E >Removes the all specified listener from the listener array for the event `eventName`. Returns a reference to the `EventEmitter`, so that calls can be chained.**Since:** 1.1.0**Parameters**

Name	Type
eventName	E
listener	(... args : any []) => void

Returns: EventEmitter < E >

on

```
on(eventName: E, listener: fn): EventEmitter < E >
```

Adds the `listener` function to the end of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.0.0

Parameters

Name	Type
eventName	E
listener	(... args : any []) => void

Returns: `EventEmitter < E >`

once

```
once(eventName: E, listener: fn): EventEmitter < E >
```

Adds a **one-time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
eventName	E
listener	(... args : any []) => void

Returns: `EventEmitter < E >`

prependListener

```
prependListener( eventName : E , listener : fn ): EventEmitter < E >
```

Adds the `listener` function to the *beginning* of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>E</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `EventEmitter < E >`

prependOnceListener

```
prependOnceListener( eventName : E , listener : fn ): EventEmitter < E >
```

Adds a **one-time** `listener` function for the event named `eventName` to the*beginning* of the listeners array. The next time `eventName` is triggered, this listener is removed, and then invoked.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>E</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `EventEmitter < E >`

`removeAllListeners``removeAllListeners(event? : E): EventEmitter < E >`

Removes all listeners, or those of the specified eventName.

Returns a reference to the `EventEmitter`, so that calls can be chained.

Since: 1.1.0

Parameters

Name	Type
<code>event?</code>	<code>E</code>

Returns: `EventEmitter < E >`

`removeListener``removeListener(eventName : E , listener : fn): EventEmitter < E >`

Alias for `emitter.off(eventName, listener)`.

Since: 1.1.0

Parameters

Name	Type
<code>eventName</code>	<code>E</code>
<code>listener</code>	<code>(... args : any []) => void</code>

Returns: `EventEmitter < E >`

Interfaces

`ChildProcess`

Since: 1.0.0

Properties

`code`

code: `null | number`

Exit code of the process. `null` if the process was terminated by a signal on Unix.

Defined in: [shell.ts:107](#)

`signal`

signal: `null | number`

If the process was terminated by a signal, represents that signal.

Defined in: [shell.ts:109](#)

`stderr`

stderr: `string`

The data that the process wrote to `stderr`.

Defined in: [shell.ts:113](#)

`stdout`

stdout: `string`

The data that the process wrote to `stdout`.

Defined in: [shell.ts:111](#)

SpawnOptions

Since: 1.0.0

Properties

`cwd`

```
Optional cwd: string
```

Current working directory.

Defined in: [shell.ts:86](#)

`encoding`

```
Optional encoding: string
```

Character encoding for stdout/stderr

Since: 1.1.0

Defined in: [shell.ts:94](#)

`env`

```
Optional env: { [name: string]: string; }
```

Environment variables. set to `null` to clear the process env.

Defined in: [shell.ts:88](#)

Functions

`open`

```
open( path: string, openWith?: string ): Promise < void >
```

Opens a path or URL with the system's default app, or the one specified with `openWith`.

The `openWith` value must be one of `firefox`, `google chrome`, `chromium` `safari`, `open`, `start`, `xdg-open`, `gio`, `gnome-open`, `kde-open` or `wslview`.

Example

```
import { open } from '@tauri-apps/api/shell';
// opens the given URL on the default browser:
await open('https://github.com/tauri-apps/tauri');
// opens the given URL using `firefox`:
await open('https://github.com/tauri-apps/tauri', 'firefox');
// opens a file using the default program:
await open('/path/to/file');
```

Since: 1.0.0

Parameters

Name	Type	Description
path	string	The path or URL to open. This value is matched against the string regex defined on <code>tauri.conf.json > tauri > allowlist > shell > open</code> , which defaults to <code>^https?:\/\/</code> .
open-With?	string	The app to open the file or URL with. Defaults to the system default application for the specified path type.

Returns: `Promise < void >`

[Go to TOC](#)

tauri

Invoke your custom commands.

This package is also accessible with `window.__TAURI__.tauri` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

Interfaces

InvokeArgs

Command arguments.

Since: 1.0.0

Functions

convertFileSrc

```
convertFileSrc( filePath: string, protocol?: string ): string
```

Convert a device file path to an URL that can be loaded by the webview. Note that `asset:` and `https://asset.localhost` must be added to `tauri.security.csp` in `tauri.conf.json`.

Example CSP value: `csp: default-src 'self'; img-src 'self' asset: https://asset.localhost` to use the asset protocol on image sources.

Additionally, `asset` must be added to `tauri.allowlist.protocol` in `tauri.conf.json` and its access scope must be defined on the `assetScope` array on the same `protocol` object.

Example

```
import { appDataDir, join } from '@tauri-apps/api/path';
import { convertFileSrc } from '@tauri-apps/api/tauri';
const appDataDirPath = await appDataDir();
const filePath = await join(appDataDirPath, 'assets/video.mp4');
const assetUrl = convertFileSrc(filePath);

const video = document.getElementById('my-video');
const source = document.createElement('source');
source.type = 'video/mp4';
source.src = assetUrl;
video.appendChild(source);
video.load();
```

Since: 1.0.0

Parameters

Name	Type	Default value	Description
filePath	string	undefined	The file path.
protoCol	string	'asset'	The protocol to use. Defaults to <code>asset</code> . You only need to set this when using a custom protocol.

Returns: string

the URL that can be used as source on the webview.

invoke

invoke< T >(cmd: string , args?: InvokeArgs): Promise < T >

Sends a message to the backend.

Example

```
import { invoke } from '@tauri-apps/api/tauri';
await invoke('login', { user: 'tauri', password: 'poiwe3h4r5ip3yrhtew9ty' });
```

Since: 1.0.0

Type parameters

- T

Parameters

Name	Type	Description
cmd	string	The command name.
args	InvokeArgs	The optional arguments to pass to the command.

Returns: Promise < T >

A promise resolving or rejecting to the backend response.

transformCallback

transformCallback(callback?: fn , once?: boolean): number

Transforms a callback function to a string identifier that can be passed to the backend. The backend uses the identifier to `eval()` the callback.

Since: 1.0.0

Parameters

Name	Type	Default value
<code>callback?</code>	<code>(response: any) => void</code>	<code>undefined</code>
<code>once</code>	<code>boolean</code>	<code>false</code>

Returns: `number`

A unique identifier associated with the callback function.

[Go to TOC](#)

updater

Customize the auto updater flow.

This package is also accessible with `window.__TAURI__.updater` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

Interfaces

UpdateManifest

Since: 1.0.0

Properties

body

body: string

Defined in: [updater.ts:34](#)

date

date: string

Defined in: [updater.ts:33](#)

version

version: string

Defined in: [updater.ts:32](#)

UpdateResult

Since: 1.0.0

Properties

`manifest`

Optional **manifest**: UpdateManifest

Defined in: [updater.ts:41](#)

`shouldUpdate`

shouldUpdate: boolean

Defined in: [updater.ts:42](#)

UpdateStatusResult

Since: 1.0.0

Properties

`error`

Optional **error**: string

Defined in: [updater.ts:24](#)

`status`

status: UpdateStatus

Defined in: [updater.ts:25](#)

Type Aliases

UpdateStatus

UpdateStatus: PENDING | ERROR | DONE | UPTODATE

Since: 1.0.0

Defined in: [updater.ts:18](#)

Functions

checkUpdate

checkUpdate(): Promise < UpdateResult >

Checks if an update is available.

Example

```
import { checkUpdate } from '@tauri-apps/api/updater';
const update = await checkUpdate();
// now run installUpdate() if needed
```

Since: 1.0.0

Returns: `Promise < UpdateResult >`

Promise resolving to the update status.

installUpdate

installUpdate(): Promise < void >

Install the update if there's one available.

Example

```
import { checkUpdate, installUpdate } from '@tauri-apps/api/updater';
const update = await checkUpdate();
if (update.shouldUpdate) {
  console.log(`Installing update ${update.manifest?.version}, ${update.manifest?.date}, ${update.manifest?.body}`);
  await installUpdate();
}
```

Since: 1.0.0

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

onUpdaterEvent

```
onUpdaterEvent( handler : fn ): Promise < UnlistenFn >
```

Listen to an updater event.

Example

```
import { onUpdaterEvent } from '@tauri-apps/api/updater';
const unlisten = await onUpdaterEvent(({ error, status }) => {
  console.log('Updater event', error, status);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	(status : UpdateStatusResult) => void

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

[Go to TOC](#)

window

Provides APIs to create windows, communicate with other windows and manipulate the current window.

This package is also accessible with `window.__TAURI__.window` when `build.withGlobalTauri` in `tauri.conf.json` is set to `true`.

The APIs must be added to `tauri.allowlist.window` in `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      window: {
        all: true, // enable all window APIs
        create: true, // enable window creation
        center: true,
        requestUserAttention: true,
        setResizable: true,
        setTitle: true,
        maximize: true,
        unmaximize: true,
        minimize: true,
        unminimize: true,
        show: true,
        hide: true,
        close: true,
        setDecorations: true,
        setAlwaysOnTop: true,
        setSize: true,
        setMinSize: true,
        setMaxSize: true,
        setPosition: true,
        setFullscreen: true,
        setFocus: true,
        setIcon: true,
        setSkipTaskbar: true,
        setCursorGrab: true,
        setCursorVisible: true,
        setCursorIcon: true,
        setCursorPosition: true,
        setIgnoreCursorEvents: true,
        startDragging: true,
        print: true
      }
    }
  }
}
```

It is recommended to allowlist only the APIs you use for optimal bundle size and security.

Window events

Events can be listened using `appWindow.listen`:

```
import { appWindow } from '@tauri-apps/api/window';
appWindow.listen(my-window-event, ({ event, payload }) => { });
```

Enumerations

UserAttentionType

Attention type to request on a window.

Since: 1.0.0

Enumeration Members

Name	Type	Description	Defined in
Critical	1	<p>#### Platform-specific</p> <ul style="list-style-type: none"> - macOS: Bounces the dock icon until the application is in focus. - Windows: Flashes both the window and the taskbar button until the application is in focus. 	window.ts:224
Informational	2	<p>#### Platform-specific</p> <ul style="list-style-type: none"> - macOS: Bounces the dock icon once. - Windows: Flashes the taskbar button until the application is in focus. 	window.ts:230

Classes

CloseRequestedEvent

Since: 1.0.2

Constructors

constructor

```
new CloseRequestedEvent( event : Event < null >): CloseRequestedEvent
```

Parameters

Name	Type
event	Event < null >

Defined in: [window.ts:1843](#)

Properties

`event`

event: string

Event name

Defined in: [window.ts:1836](#)

`id`

id: number

Event identifier used to unlisten

Defined in: [window.ts:1840](#)

`windowLabel`

windowLabel: string

The label of the window that emitted this event.

Defined in: [window.ts:1838](#)

Methods

`isPreventDefault`

isPreventDefault(): boolean

Returns: boolean

`preventDefault`

preventDefault(): void

Returns: void

LogicalPosition

A position represented in logical pixels.

Since: 1.0.0

Constructors

`constructor`

```
new LogicalPosition( x : number , y : number ): LogicalPosition
```

Parameters

Name	Type
x	number
y	number

Defined in: [window.ts:162](#)

Properties

`type`

```
type: string = 'Logical'
```

Defined in: [window.ts:158](#)

`x`

```
x: number
```

Defined in: [window.ts:159](#)

`y`

```
y: number
```

Defined in: [window.ts:160](#)

LogicalSize

A size represented in logical pixels.

Since: 1.0.0

Constructors

`constructor`

```
new LogicalSize(width : number , height : number ): LogicalSize
```

Parameters

Name	Type
width	number
height	number

Defined in: [window.ts:116](#)

Properties

`height`

```
height: number
```

Defined in: [window.ts:114](#)

`type`

```
type: string = 'Logical'
```

Defined in: [window.ts:112](#)

`width`

```
width: number
```

Defined in: [window.ts:113](#)

PhysicalPosition

A position represented in physical pixels.

Since: 1.0.0

Constructors

`constructor`

```
new PhysicalPosition( x : number , y : number ): PhysicalPosition
```

Parameters

Name	Type
x	number
y	number

Defined in: [window.ts:178](#)

Properties

`type`

```
type: string = 'Physical'
```

Defined in: [window.ts:174](#)

`x`

```
x: number
```

Defined in: [window.ts:175](#)

`y`

```
y: number
```

Defined in: [window.ts:176](#)

Methods

`toLogical`

```
toLogical( scaleFactor : number ): LogicalPosition
```

Converts the physical position to a logical one.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const factor = await appWindow.scaleFactor();
const position = await appWindow.innerPosition();
const logical = position.toLogical(factor);
```

Parameters

Name	Type
scaleFactor	number

Returns: `LogicalPosition`

PhysicalSize

A size represented in physical pixels.

Since: 1.0.0

Constructors

`constructor`

```
new PhysicalSize( width : number , height : number ): PhysicalSize
```

Parameters

Name	Type
width	number
height	number

Defined in: [window.ts:132](#)

Properties

`height`

height: number

Defined in: [window.ts:130](#)

`type`

type: string = 'Physical'

Defined in: [window.ts:128](#)

`width`

width: number

Defined in: [window.ts:129](#)

Methods

`toLogical`

toLogical(`scaleFactor` : number): LogicalSize

Converts the physical size to a logical one.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const factor = await appWindow.scaleFactor();
const size = await appWindow.innerSize();
const logical = size.toLogical(factor);
```

Parameters

Name	Type
<code>scaleFactor</code>	number

Returns: LogicalSize

WebviewWindow

Create new webview windows and get a handle to existing ones.

Windows are identified by a *label* a unique identifier that can be used to reference it later. It may only contain alphanumeric characters `a-zA-Z` plus the following special characters `-`, `/`, `:` and `_`.

Example

```
// loading embedded asset:
const webview = new WebviewWindow('theUniqueLabel', {
  url: 'path/to/page.html'
});
// alternatively, load a remote URL:
const webview = new WebviewWindow('theUniqueLabel', {
  url: 'https://github.com/tauri-apps/tauri'
});

webview.once('tauri://created', function () {
  // webview window successfully created
});
webview.once('tauri://error', function (e) {
  // an error happened creating the webview window
});

// emit an event to the backend
await webview.emit(some event, data);
// listen to an event from the backend
const unlisten = await webview.listen(event name, e => {});
unlisten();
```

Since: 1.0.2

Hierarchy

- `WindowManager`
 - **WebviewWindow**

Constructors

constructor

```
new WebviewWindow( label : string , options?: WindowOptions ): WebviewWindow
```

Creates a new WebviewWindow.

Example

```
import { WebviewWindow } from '@tauri-apps/api/window';
const webview = new WebviewWindow('my-label', {
  url: 'https://github.com/tauri-apps/tauri'
});
webview.once('tauri://created', function () {
  // webview window successfully created
```

```
});  
webview.once('tauri://error', function (e) {  
    // an error happened creating the webview window  
});
```

- **Parameters**

Name	Type	Description
label	string	The unique webview window label. Must be alphanumeric: <code>a-zA-Z-/_</code> .
options	WindowOptions	-

Overrides: [WindowManager.constructor](#)

Defined in: [window.ts:1911](#)

Properties

`label`

label: string

The window label. It is a unique identifier for the window, can be used to reference it later.

Inherited from: [WindowManager.label](#)

Defined in: [window.ts:313](#)

`listeners`

listeners: { [key: string]: EventCallback<any>[]; }

Local event listeners.

Inherited from: [WindowManager.listeners](#)

Defined in: [window.ts:315](#)

Methods

`_handleTauriEvent`

_handleTauriEvent< T >(event: string , handler: EventCallback< T >): boolean

Type parameters

- `T`

Parameters

Name	Type
<code>event</code>	<code>string</code>
<code>handler</code>	<code>EventCallback < T ></code>

Returns: `boolean`

`center`

```
center(): Promise < void >
```

Centers the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.center();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`close`

```
close(): Promise < void >
```

Closes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.close();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`emit`

```
emit( event : string , payload? : unknown ): Promise < void >
```

Emits an event to the backend, tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.emit('window-loaded', { loggedIn: true, token: 'authToken' });
```

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
payload?	unknown	Event payload.

Returns: `Promise < void >`

`hide`

hide(): Promise < void >

Sets the window visibility to false.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.hide();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`innerPosition`

innerPosition(): Promise < PhysicalPosition >

The position of the top-left hand corner of the window's client area relative to the top-left hand corner of the desktop.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const position = await appWindow.innerPosition();
```

Returns: `Promise < PhysicalPosition >`

The window's inner position.

innerSize**innerSize()**: Promise < PhysicalSize >

The physical size of the window's client area. The client area is the content of the window, excluding the title bar and borders.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const size = await appWindow.innerSize();
```

Returns: Promise < PhysicalSize >

The window's inner size.

isDecorated**isDecorated()**: Promise < boolean >

Gets the window's current decorated state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const decorated = await appWindow.isDecorated();
```

Returns: Promise < boolean >

Whether the window is decorated or not.

isFullscreen**isFullscreen()**: Promise < boolean >

Gets the window's current fullscreen state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const fullscreen = await appWindow.isFullscreen();
```

Returns: Promise < boolean >

Whether the window is in fullscreen mode or not.

isMaximized**isMaximized()**: Promise < boolean >

Gets the window's current maximized state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const maximized = await appWindow.isMaximized();
```

Returns: Promise < boolean >

Whether the window is maximized or not.

isResizable**isResizable()**: Promise < boolean >

Gets the window's current resizable state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const resizable = await appWindow.isResizable();
```

Returns: Promise < boolean >

Whether the window is resizable or not.

isVisible**isVisible()**: Promise < boolean >

Gets the window's current visible state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const visible = await appWindow.isVisible();
```

Returns: Promise < boolean >

Whether the window is visible or not.

listen

```
listen< T >( event : string , handler : EventCallback < T >): Promise < UnlistenFn >
```

Listen to an event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.listen<string>('state-changed', (event) => {
  console.log(`Got error: ${payload}`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Type parameters

- **T**

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>handler</code>	<code>EventCallback < T ></code>	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

maximize

```
maximize(): Promise < void >
```

Maximizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.maximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
minimize
```

```
minimize(): Promise < void >
```

Minimizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.minimize();
```

Returns: Promise < void >

A promise indicating the success or failure of the operation.

```
onCloseRequested
```

```
onCloseRequested( handler : fn ): Promise < UnlistenFn >
```

Listen to window close requested. Emitted when the user requests to closes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
import { confirm } from '@tauri-apps/api/dialog';
const unlisten = await appWindow.onCloseRequested(async (event) => {
  const confirmed = await confirm('Are you sure?');
  if (!confirmed) {
    // user did not confirm closing the window; let's prevent it
    event.preventDefault();
  }
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	(event : CloseRequestedEvent) => void

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onFileDropEvent`

onFileDropEvent(handler : EventCallback < FileDropEvent >): Promise < UnlistenFn >

Listen to a file drop event. The listener is triggered when the user hovers the selected files on the window, drops the files or cancels the operation.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onFileDropEvent((event) => {
  if (event.payload.type === 'hover') {
    console.log('User hovering', event.payload.paths);
  } else if (event.payload.type === 'drop') {
    console.log('User dropped', event.payload.paths);
  } else {
    console.log('File drop cancelled');
  }
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
<code>handler</code>	<code>EventCallback < FileDropEvent ></code>

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onFocusChanged`

onFocusChanged(handler : EventCallback < boolean >): Promise < UnlistenFn >

Listen to window focus change.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onFocusChanged(({ payload: focused }) => {
  console.log('Focus changed, window is focused? ' + focused);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2**Parameters**

Name	Type
handler	EventCallback<boolean>

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onMenuItemClicked`

```
onMenuItemClicked( handler: EventCallback< string > ): Promise < UnlistenFn >
```

Listen to the window menu item click. The payload is the item id.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onMenuItemClicked(({ payload: menuItem }) => {
  console.log('Menu clicked: ' + menuItem);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2**Parameters**

Name	Type
handler	EventCallback< string >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onMoved

```
onMoved( handler : EventCallback < PhysicalPosition >): Promise < UnlistenFn >
```

Listen to window move.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onMoved(({ payload: position }) => {
  console.log('Window moved', position);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < PhysicalPosition >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onResized

```
onResized( handler : EventCallback < PhysicalSize >): Promise < UnlistenFn >
```

Listen to window resize.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onResized(({ payload: size }) => {
  console.log('Window resized', size);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < PhysicalSize >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onScaleChanged

onScaleChanged(handler : EventCallback < ScaleFactorChanged >): Promise < UnlistenFn >

Listen to window scale change. Emitted when the window's scale factor has changed. The following user actions can cause DPI changes:

- Changing the display's resolution.
- Changing the display's scale factor (e.g. in Control Panel on Windows).
- Moving the window to a display with a different scale factor.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onScaleChanged(({ payload }) => {
  console.log('Scale changed', payload.scaleFactor, payload.size);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < ScaleFactorChanged >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onThemeChanged

```
onThemeChanged( handler : EventCallback < Theme >): Promise < UnlistenFn >
```

Listen to the system theme change.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onThemeChanged(({ payload: theme }) => {
  console.log('New theme: ' + theme);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < Theme >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

once

```
once< T >( event : string , handler : EventCallback < T >): Promise < UnlistenFn >
```

Listen to an one-off event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.once<null>('initialized', (event) => {
  console.log(`Window initialized!`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Type parameters

- `T`

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>handler</code>	<code>EventCallback < T ></code>	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`outerPosition`

outerPosition(): Promise < PhysicalPosition >

The position of the top-left hand corner of the window relative to the top-left hand corner of the desktop.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const position = await appWindow.outerPosition();
```

Returns: `Promise < PhysicalPosition >`

The window's outer position.

`outerSize`

outerSize(): Promise < PhysicalSize >

The physical size of the entire window. These dimensions include the title bar and borders. If you don't want that (and you usually don't), use `inner_size` instead.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const size = await appWindow.outerSize();
```

Returns: `Promise < PhysicalSize >`

The window's outer size.

requestUserAttention

```
requestUserAttention( requestType : null | UserAttentionType ): Promise < void >
```

Requests user attention to the window, this has no effect if the application is already focused. How requesting for user attention manifests is platform dependent, see [UserAttentionType](#) for details.

Providing `null` will unset the request for user attention. Unsetting the request for user attention might not be done automatically by the WM when the window receives input.

Platform-specific

- **macOS:** `null` has no effect.
- **Linux:** Urgency levels have the same effect.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.requestUserAttention();
```

Parameters

Name	Type
<code>requestType</code>	<code>null UserAttentionType</code>

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

scaleFactor

```
scaleFactor(): Promise < number >
```

The scale factor that can be used to map physical pixels to logical pixels.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const factor = await appWindow.scaleFactor();
```

Returns: `Promise < number >`

The window's monitor scale factor.

`setAlwaysOnTop`

```
setAlwaysOnTop( alwaysOnTop : boolean ): Promise < void >
```

Whether the window should always be on top of other windows.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setAlwaysOnTop(true);
```

Parameters

Name	Type	Description
<code>alwaysOnTop</code>	<code>boolean</code>	Whether the window should always be on top of other windows or not.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setCursorGrab`

```
setCursorGrab( grab : boolean ): Promise < void >
```

Grabs the cursor, preventing it from leaving the window.

There's no guarantee that the cursor will be hidden. You should hide it by yourself if you want so.

Platform-specific

- **Linux:** Unsupported.
- **macOS:** This locks the cursor in a fixed location, which looks visually awkward.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorGrab(true);
```

Parameters

Name	Type	Description
<code>grab</code>	<code>boolean</code>	<code>true</code> to grab the cursor icon, <code>false</code> to release it.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setCursorIcon
```

```
setCursorIcon( icon : CursorIcon ): Promise < void >
```

Modifies the cursor icon of the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorIcon('help');
```

Parameters

Name	Type	Description
icon	CursorIcon	The new cursor icon.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setCursorPosition
```

```
setCursorPosition( position : PhysicalPosition | LogicalPosition ): Promise < void >
```

Changes the position of the cursor in window coordinates.

Example

```
import { appWindow, LogicalPosition } from '@tauri-apps/api/window';
await appWindow.setCursorPosition(new LogicalPosition(600, 300));
```

Parameters

Name	Type	Description
position	PhysicalPosition LogicalPosition	The new cursor position.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setCursorVisible`

```
setCursorVisible( visible : boolean ): Promise < void >
```

Modifies the cursor's visibility.

Platform-specific

- **Windows:** The cursor is only hidden within the confines of the window.
- **macOS:** The cursor is hidden as long as the window has input focus, even if the cursor is outside of the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorVisible(false);
```

Parameters

Name	Type	Description
<code>visible</code>	<code>boolean</code>	If <code>false</code> , this will hide the cursor. If <code>true</code> , this will show the cursor.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setDecorations`

```
setDecorations( decorations : boolean ): Promise < void >
```

Whether the window should have borders and bars.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setDecorations(false);
```

Parameters

Name	Type	Description
<code>decorations</code>	<code>boolean</code>	Whether the window should have borders and bars.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

setFocus

```
setFocus(): Promise < void >
```

Bring the window to front and focus.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setFocus();
```

Returns: Promise < void >

A promise indicating the success or failure of the operation.

setFullscreen

```
setFullscreen( fullscreen : boolean ): Promise < void >
```

Sets the window fullscreen state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setFullscreen(true);
```

Parameters

Name	Type	Description
fullscreen	boolean	Whether the window should go to fullscreen or not.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

setIcon

```
setIcon( icon : string | Uint8Array ): Promise < void >
```

Sets the window icon.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setIcon('/tauri/awesome.png');
```

Note that you need the `icon-ico` or `icon-png` Cargo features to use this API.

To enable it, change your `Cargo.toml` file:

```
[dependencies]
tauri = { version = ..., features = [..., icon-png] }
```

Parameters

Name	Type	Description
<code>icon</code>	<code>string</code> <code>Uint8Array</code>	Icon bytes or path to the icon file.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setIgnoreCursorEvents`

```
setIgnoreCursorEvents( ignore : boolean ): Promise < void >
```

Changes the cursor events behavior.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setIgnoreCursorEvents(true);
```

Parameters

Name	Type	Description
<code>ignore</code>	<code>boolean</code>	<code>true</code> to ignore the cursor events; <code>false</code> to process them as usual.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setMaxSize`

```
setMaxSize( size : undefined | null | PhysicalSize | LogicalSize ): Promise < void >
```

Sets the window maximum inner size. If the `size` argument is undefined, the constraint is unset.

Example

```
import { appWindow, LogicalSize } from '@tauri-apps/api/window';
await appWindow.setMaxSize(new LogicalSize(600, 500));
```

Parameters

Name	Type	Description
size	undefined null PhysicalSize LogicalSize	The logical or physical inner size, or <code>null</code> to unset the constraint.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setMinSize`

```
setMinSize( size: undefined | null | PhysicalSize | LogicalSize ): Promise < void >
```

Sets the window minimum inner size. If the `size` argument is not provided, the constraint is unset.

Example

```
import { appWindow, PhysicalSize } from '@tauri-apps/api/window';
await appWindow.setMinSize(new PhysicalSize(600, 500));
```

Parameters

Name	Type	Description
size	undefined null PhysicalSize LogicalSize	The logical or physical inner size, or <code>null</code> to unset the constraint.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setPosition`

```
setPosition( position: PhysicalPosition | LogicalPosition ): Promise < void >
```

Sets the window outer position.

Example

```
import { appWindow, LogicalPosition } from '@tauri-apps/api/window';
await appWindow.setPosition(new LogicalPosition(600, 500));
```

Parameters

Name	Type	Description
position	PhysicalPosition LogicalPosition	The new position, in logical or physical pixels.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

`setResizable`

```
setResizable(resizable: boolean): Promise < void >
```

Updates the window resizable flag.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setResizable(false);
```

Parameters

Name	Type
resizable	boolean

Returns: Promise < void >

A promise indicating the success or failure of the operation.

`setSize`

```
setSize(size: PhysicalSize | LogicalSize): Promise < void >
```

Resizes the window with a new inner size.

Example

```
import { appWindow, LogicalSize } from '@tauri-apps/api/window';
await appWindow.setSize(new LogicalSize(600, 500));
```

Parameters

Name	Type	Description
size	PhysicalSize LogicalSize	The logical or physical inner size.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setSkipTaskbar`

setSkipTaskbar(skip : boolean): Promise < void >

Whether the window icon should be hidden from the taskbar or not.

Platform-specific

- **macOS:** Unsupported.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setSkipTaskbar(true);
```

Parameters

Name	Type	Description
skip	boolean	true to hide window icon, false to show it.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setTitle`

setTitle(title : string): Promise < void >

Sets the window title.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setTitle('Tauri');
```

Parameters

Name	Type	Description
title	string	The new title

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`show`

`show(): Promise < void >`

Sets the window visibility to true.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.show();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`startDragging`

`startDragging(): Promise < void >`

Starts dragging the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.startDragging();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`theme`

`theme(): Promise < null | Theme >`

Gets the window's current theme.

Platform-specific

- **macOS:** Theme was introduced on macOS 10.14. Returns `light` on macOS 10.13 and below.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const theme = await appWindow.theme();
```

Returns: `Promise < null | Theme >`

The window theme.

`toggleMaximize`

toggleMaximize(): Promise < void >

Toggles the window maximized state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.toggleMaximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`unmaximize`

unmaximize(): Promise < void >

Unmaximizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.unmaximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`unminimize`

unminimize(): Promise < void >

Unminimizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.unminimize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

getByLabel

```
Static getByLabel( label : string ): null | WebviewWindow
```

Gets the WebviewWindow for the webview associated with the given label.

Example

```
import { WebviewWindow } from '@tauri-apps/api/window';
const mainWindow = WebviewWindow.getByLabel('main');
```

Parameters

Name	Type	Description
<code>label</code>	<code>string</code>	The webview window label.

Returns: `null | WebviewWindow`

The WebviewWindow instance to communicate with the webview or null if the webview doesn't exist.

WebviewWindowHandle

A webview window handle allows emitting and listening to events from the backend that are tied to the window.

Since: 1.0.0

Hierarchy

- **WebviewWindowHandle**

- `WindowManager`

Constructors

`constructor`

```
new WebviewWindowHandle( label : string ): WebviewWindowHandle
```

Parameters

Name	Type
<code>label</code>	<code>string</code>

Defined in: [window.ts:317](#)

Properties

`label`

label: `string`

The window label. It is a unique identifier for the window, can be used to reference it later.

Defined in: [window.ts:313](#)

`listeners`

listeners: { [key: string] : EventCallback < any >[]; }

Local event listeners.

Defined in: [window.ts:315](#)

Methods

`_handleTauriEvent`

_handleTauriEvent< T >(event : string , handler : EventCallback < T >): boolean

Type parameters

- `T`

Parameters

Name	Type
event	string
handler	EventCallback < T >

Returns: boolean

emit

emit(event : string , payload? : unknown): Promise < void >

Emits an event to the backend, tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.emit('window-loaded', { loggedIn: true, token: 'authToken' });
```

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, - , / , : and _ .
payload?	unknown	Event payload.

Returns: Promise < void >

listen

listen< T >(event : string , handler : EventCallback < T >): Promise < UnlistenFn >

Listen to an event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.listen<string>('state-changed', (event) => {
  console.log(`Got error: ${payload}`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Type parameters

- T

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
handler	EventCallback < T >	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`once`

```
once< T >( event : string , handler : EventCallback < T >): Promise < UnlistenFn >
```

Listen to an one-off event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.once<null>('initialized', (event) => {
  console.log(`Window initialized!`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Type parameters

- `T`

Parameters

Name	Type	Description
event	string	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
handler	EventCallback < T >	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

WindowManager

Manage the current window object.

Since: 1.0.0

Hierarchy

- [WebviewWindowHandle](#)
 - **WindowManager**
 - [WebviewWindow](#)

Constructors

constructor

```
new WindowManager(label : string): WindowManager
```

Parameters

Name	Type
label	string

Inherited from: [WebviewWindowHandle.constructor](#)

Defined in: [window.ts:317](#)

Properties

label

```
label: string
```

The window label. It is a unique identifier for the window, can be used to reference it later.

Inherited from: [WebviewWindowHandle.label](#)

Defined in: [window.ts:313](#)

listeners

```
listeners: { [key: string] : EventCallback < any >[]; }
```

Local event listeners.

Inherited from: [WebviewWindowHandle.listeners](#)

Defined in: [window.ts:315](#)

Methods

`_handleTauriEvent`

```
_handleTauriEvent< T >( event : string , handler : EventCallback < T >): boolean
```

Type parameters

- `T`

Parameters

Name	Type
<code>event</code>	<code>string</code>
<code>handler</code>	<code>EventCallback < T ></code>

Returns: `boolean`

`center`

```
center(): Promise < void >
```

Centers the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.center();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`close`

```
close(): Promise < void >
```

Closes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.close();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`emit`

```
emit( event : string , payload? : unknown ): Promise < void >
```

Emits an event to the backend, tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.emit('window-loaded', { loggedIn: true, token: 'authToken' });
```

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>payload?</code>	<code>unknown</code>	Event payload.

Returns: `Promise < void >`

`hide`

```
hide(): Promise < void >
```

Sets the window visibility to false.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.hide();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`innerPosition`

```
innerPosition(): Promise < PhysicalPosition >
```

The position of the top-left hand corner of the window's client area relative to the top-left hand corner of the desktop.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const position = await appWindow.innerPosition();
```

Returns: `Promise < PhysicalPosition >`

The window's inner position.

`innerSize`

innerSize(): Promise < PhysicalSize >

The physical size of the window's client area. The client area is the content of the window, excluding the title bar and borders.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const size = await appWindow.innerSize();
```

Returns: `Promise < PhysicalSize >`

The window's inner size.

`isDecorated`

isDecorated(): Promise < boolean >

Gets the window's current decorated state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const decorated = await appWindow.isDecorated();
```

Returns: `Promise < boolean >`

Whether the window is decorated or not.

`isFullscreen`

isFullscreen(): Promise < boolean >

Gets the window's current fullscreen state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const fullscreen = await appWindow.isFullscreen();
```

Returns: `Promise < boolean >`

Whether the window is in fullscreen mode or not.

```
isMaximized
```

```
isMaximized(): Promise < boolean >
```

Gets the window's current maximized state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const maximized = await appWindow.isMaximized();
```

Returns: `Promise < boolean >`

Whether the window is maximized or not.

```
isResizable
```

```
isResizable(): Promise < boolean >
```

Gets the window's current resizable state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const resizable = await appWindow.isResizable();
```

Returns: `Promise < boolean >`

Whether the window is resizable or not.

```
isVisible
```

```
isVisible(): Promise < boolean >
```

Gets the window's current visible state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const visible = await appWindow.isVisible();
```

Returns: `Promise < boolean >`

Whether the window is visible or not.

`listen`

```
listen< T >( event : string , handler : EventCallback < T >): Promise < UnlistenFn >
```

Listen to an event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.listen<string>('state-changed', (event) => {
  console.log(`Got error: ${payload}`);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Type parameters

- `T`

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>handler</code>	<code>EventCallback < T ></code>	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`maximize`

```
maximize(): Promise < void >
```

Maximizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.maximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`minimize`

```
minimize(): Promise < void >
```

Minimizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.minimize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`onCloseRequested`

```
onCloseRequested( handler: fn ): Promise < UnlistenFn >
```

Listen to window close requested. Emitted when the user requests to closes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
import { confirm } from '@tauri-apps/api/dialog';
const unlisten = await appWindow.onCloseRequested(async (event) => {
  const confirmed = await confirm('Are you sure?');
  if (!confirmed) {
    // user did not confirm closing the window; let's prevent it
    event.preventDefault();
  }
});
// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	(event : CloseRequestedEvent) => void

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onFileDropEvent`

onFileDropEvent(handler : EventCallback < FileDropEvent >) : Promise < UnlistenFn >

Listen to a file drop event. The listener is triggered when the user hovers the selected files on the window, drops the files or cancels the operation.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onFileDropEvent((event) => {
  if (event.payload.type === 'hover') {
    console.log('User hovering', event.payload.paths);
  } else if (event.payload.type === 'drop') {
    console.log('User dropped', event.payload.paths);
  } else {
    console.log('File drop cancelled');
  }
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < FileDropEvent >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onFocusChanged

```
onFocusChanged( handler : EventCallback < boolean >): Promise < UnlistenFn >
```

Listen to window focus change.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onFocusChanged(({ payload: focused }) => {
  console.log('Focus changed, window is focused? ', + focused);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < boolean >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onMenuItemClicked

```
onMenuItemClicked( handler : EventCallback < string >): Promise < UnlistenFn >
```

Listen to the window menu item click. The payload is the item id.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onMenuItemClicked(({ payload: menuItem }) => {
  console.log('Menu clicked: ', menuItem);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < string >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onMoved

```
onMoved( handler : EventCallback < PhysicalPosition > ): Promise < UnlistenFn >
```

Listen to window move.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onMoved(({ payload: position }) => {
    console.log('Window moved', position);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < PhysicalPosition >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

onResized

```
onResized( handler : EventCallback < PhysicalSize > ): Promise < UnlistenFn >
```

Listen to window resize.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onResized(({ payload: size }) => {
  console.log('Window resized', size);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < PhysicalSize >

Returns: Promise < UnlistenFn >

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onScaleChanged`

```
onScaleChanged( handler : EventCallback < ScaleFactorChanged >): Promise < UnlistenFn >
```

Listen to window scale change. Emitted when the window's scale factor has changed. The following user actions can cause DPI changes:

- Changing the display's resolution.
- Changing the display's scale factor (e.g. in Control Panel on Windows).
- Moving the window to a display with a different scale factor.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onScaleChanged(({ payload }) => {
  console.log('Scale changed', payload.scaleFactor, payload.size);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
handler	EventCallback < ScaleFactorChanged >

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`onThemeChanged`

```
onThemeChanged( handler : EventCallback < Theme > ): Promise < UnlistenFn >
```

Listen to the system theme change.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.onThemeChanged(({ payload: theme }) => {
  console.log('New theme: ' + theme);
});

// you need to call unlisten if your handler goes out of scope e.g. the component
// is unmounted
unlisten();
```

Since: 1.0.2

Parameters

Name	Type
<code>handler</code>	<code>EventCallback < Theme ></code>

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`once`

```
once< T >( event : string , handler : EventCallback < T > ): Promise < UnlistenFn >
```

Listen to an one-off event emitted by the backend that is tied to the webview window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const unlisten = await appWindow.once<null>('initialized', (event) => {
  console.log(`Window initialized!`);
});
```

```
// you need to call unlisten if your handler goes out of scope e.g. the component
is unmounted
unlisten();
```

Type parameters

- `T`

Parameters

Name	Type	Description
<code>event</code>	<code>string</code>	Event name. Must include only alphanumeric characters, <code>-</code> , <code>/</code> , <code>:</code> and <code>_</code> .
<code>han-</code> <code>dler</code>	<code>EventCallback < T ></code>	Event handler.

Returns: `Promise < UnlistenFn >`

A promise resolving to a function to unlisten to the event. Note that removing the listener is required if your listener goes out of scope e.g. the component is unmounted.

`outerPosition`

outerPosition(): Promise < PhysicalPosition >

The position of the top-left hand corner of the window relative to the top-left hand corner of the desktop.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const position = await appWindow.outerPosition();
```

Returns: `Promise < PhysicalPosition >`

The window's outer position.

`outerSize`

outerSize(): Promise < PhysicalSize >

The physical size of the entire window. These dimensions include the title bar and borders. If you don't want that (and you usually don't), use `inner_size` instead.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const size = await appWindow.outerSize();
```

Returns: `Promise < PhysicalSize >`

The window's outer size.

`requestUserAttention`

```
requestUserAttention( requestType : null | UserAttentionType ): Promise < void >
```

Requests user attention to the window, this has no effect if the application is already focused. How requesting for user attention manifests is platform dependent, see `UserAttentionType` for details.

Providing `null` will unset the request for user attention. Unsetting the request for user attention might not be done automatically by the WM when the window receives input.

Platform-specific

- **macOS:** `null` has no effect.
- **Linux:** Urgency levels have the same effect.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.requestUserAttention();
```

Parameters

Name	Type
<code>requestType</code>	<code>null UserAttentionType</code>

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`scaleFactor`

```
scaleFactor(): Promise < number >
```

The scale factor that can be used to map physical pixels to logical pixels.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const factor = await appWindow.scaleFactor();
```

Returns: `Promise < number >`

The window's monitor scale factor.

`setAlwaysOnTop`**setAlwaysOnTop(** `alwaysOnTop : boolean` **):** `Promise < void >`

Whether the window should always be on top of other windows.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setAlwaysOnTop(true);
```

Parameters

Name	Type	Description
<code>alwaysOnTop</code>	<code>boolean</code>	Whether the window should always be on top of other windows or not.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setCursorGrab`**setCursorGrab(** `grab : boolean` **):** `Promise < void >`

Grabs the cursor, preventing it from leaving the window.

There's no guarantee that the cursor will be hidden. You should hide it by yourself if you want so.

Platform-specific

- **Linux:** Unsupported.
- **macOS:** This locks the cursor in a fixed location, which looks visually awkward.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorGrab(true);
```

Parameters

Name	Type	Description
grab	boolean	true to grab the cursor icon, false to release it.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

setCursorIcon

```
setCursorIcon( icon : CursorIcon ): Promise < void >
```

Modifies the cursor icon of the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorIcon('help');
```

Parameters

Name	Type	Description
icon	CursorIcon	The new cursor icon.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

setCursorPosition

```
setCursorPosition( position : PhysicalPosition | LogicalPosition ): Promise < void >
```

Changes the position of the cursor in window coordinates.

Example

```
import { appWindow, LogicalPosition } from '@tauri-apps/api/window';
await appWindow.setCursorPosition(new LogicalPosition(600, 300));
```

Parameters

Name	Type	Description
position	PhysicalPosition LogicalPosition	The new cursor position.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

```
setCursorVisible
```

```
setCursorVisible( visible : boolean ): Promise < void >
```

Modifies the cursor's visibility.

Platform-specific

- **Windows:** The cursor is only hidden within the confines of the window.
- **macOS:** The cursor is hidden as long as the window has input focus, even if the cursor is outside of the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setCursorVisible(false);
```

Parameters

Name	Type	Description
visible	boolean	If <code>false</code> , this will hide the cursor. If <code>true</code> , this will show the cursor.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setDecorations
```

```
setDecorations( decorations : boolean ): Promise < void >
```

Whether the window should have borders and bars.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setDecorations(false);
```

Parameters

Name	Type	Description
decorations	boolean	Whether the window should have borders and bars.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setFocus
```

```
setFocus(): Promise < void >
```

Bring the window to front and focus.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setFocus();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setFullscreen
```

```
setFullscreen( fullscreen: boolean ): Promise < void >
```

Sets the window fullscreen state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setFullscreen(true);
```

Parameters

Name	Type	Description
<code>fullscreen</code>	<code>boolean</code>	Whether the window should go to fullscreen or not.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

```
setIcon
```

```
setIcon( icon: string | Uint8Array ): Promise < void >
```

Sets the window icon.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setIcon('/tauri/awesome.png');
```

Note that you need the `icon-ico` or `icon-png` Cargo features to use this API.

To enable it, change your `Cargo.toml` file:

```
[dependencies]
tauri = { version = ..., features = [..., icon-png] }
```

Parameters

Name	Type	Description
<code>icon</code>	<code>string</code> <code>Uint8Array</code>	Icon bytes or path to the icon file.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setIgnoreCursorEvents`

```
setIgnoreCursorEvents( ignore : boolean ): Promise < void >
```

Changes the cursor events behavior.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setIgnoreCursorEvents(true);
```

Parameters

Name	Type	Description
<code>ignore</code>	<code>boolean</code>	<code>true</code> to ignore the cursor events; <code>false</code> to process them as usual.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setMaxSize`

```
setMaxSize( size : undefined | null | PhysicalSize | LogicalSize ): Promise < void >
```

Sets the window maximum inner size. If the `size` argument is undefined, the constraint is unset.

Example

```
import { appWindow, LogicalSize } from '@tauri-apps/api/window';
await appWindow.setMaxSize(new LogicalSize(600, 500));
```

Parameters

Name	Type	Description
size	undefined null PhysicalSize LogicalSize	The logical or physical inner size, or <code>null</code> to unset the constraint.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setMinSize`

```
setMinSize( size: undefined | null | PhysicalSize | LogicalSize ): Promise < void >
```

Sets the window minimum inner size. If the `size` argument is not provided, the constraint is unset.

Example

```
import { appWindow, PhysicalSize } from '@tauri-apps/api/window';
await appWindow.setMinSize(new PhysicalSize(600, 500));
```

Parameters

Name	Type	Description
size	undefined null PhysicalSize LogicalSize	The logical or physical inner size, or <code>null</code> to unset the constraint.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setPosition`

```
setPosition( position: PhysicalPosition | LogicalPosition ): Promise < void >
```

Sets the window outer position.

Example

```
import { appWindow, LogicalPosition } from '@tauri-apps/api/window';
await appWindow.setPosition(new LogicalPosition(600, 500));
```

Parameters

Name	Type	Description
position	PhysicalPosition LogicalPosition	The new position, in logical or physical pixels.

Returns: Promise < void >

A promise indicating the success or failure of the operation.

`setResizable`

```
setResizable(resizable: boolean): Promise < void >
```

Updates the window resizable flag.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setResizable(false);
```

Parameters

Name	Type
resizable	boolean

Returns: Promise < void >

A promise indicating the success or failure of the operation.

`setSize`

```
setSize(size: PhysicalSize | LogicalSize): Promise < void >
```

Resizes the window with a new inner size.

Example

```
import { appWindow, LogicalSize } from '@tauri-apps/api/window';
await appWindow.setSize(new LogicalSize(600, 500));
```

Parameters

Name	Type	Description
size	PhysicalSize LogicalSize	The logical or physical inner size.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setSkipTaskbar`

```
setSkipTaskbar( skip : boolean ): Promise < void >
```

Whether the window icon should be hidden from the taskbar or not.

Platform-specific

- **macOS:** Unsupported.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setSkipTaskbar(true);
```

Parameters

Name	Type	Description
skip	boolean	true to hide window icon, false to show it.

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`setTitle`

```
setTitle( title : string ): Promise < void >
```

Sets the window title.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.setTitle('Tauri');
```

Parameters

Name	Type	Description
title	string	The new title

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`show`

`show(): Promise < void >`

Sets the window visibility to true.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.show();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`startDragging`

`startDragging(): Promise < void >`

Starts dragging the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.startDragging();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`theme`

`theme(): Promise < null | Theme >`

Gets the window's current theme.

Platform-specific

- **macOS:** Theme was introduced on macOS 10.14. Returns `light` on macOS 10.13 and below.

Example

```
import { appWindow } from '@tauri-apps/api/window';
const theme = await appWindow.theme();
```

Returns: `Promise < null | Theme >`

The window theme.

`toggleMaximize`

toggleMaximize(): Promise < void >

Toggles the window maximized state.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.toggleMaximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`unmaximize`

unmaximize(): Promise < void >

Unmaximizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.unmaximize();
```

Returns: `Promise < void >`

A promise indicating the success or failure of the operation.

`unminimize`

unminimize(): Promise < void >

Unminimizes the window.

Example

```
import { appWindow } from '@tauri-apps/api/window';
await appWindow.unminimize();
```

Returns: `Promise<void>`

A promise indicating the success or failure of the operation.

Interfaces

Monitor

Allows you to retrieve information about a given monitor.

Since: 1.0.0

Properties

`name`

name: `null | string`

Human-readable name of the monitor

Defined in: [window.ts:79](#)

`position`

position: `PhysicalPosition`

the Top-left corner position of the monitor relative to the larger full screen area.

Defined in: [window.ts:83](#)

`scaleFactor`

scaleFactor: `number`

The scale factor that can be used to map physical pixels to logical pixels.

Defined in: [window.ts:85](#)

size**size:** PhysicalSize

The monitor's resolution.

Defined in: [window.ts:81](#)

ScaleFactorChanged

The payload for the `scaleChange` event.

Since: 1.0.2

Properties**scaleFactor****scaleFactor:** number

The new window scale factor.

Defined in: [window.ts:95](#)

size**size:** PhysicalSize

The new window size

Defined in: [window.ts:97](#)

WindowOptions

Configuration for the window to create.

Since: 1.0.0

Properties**acceptFirstMouse**Optional **acceptFirstMouse:** boolean

Whether clicking an inactive window also clicks through to the webview on macOS.

Defined in: [window.ts:2051](#)

`alwaysOnTop`

Optional **alwaysOnTop**: boolean

Whether the window should always be on top of other windows or not.

Defined in: [window.ts:2025](#)

`center`

Optional **center**: boolean

Show window in the center of the screen..

Defined in: [window.ts:1987](#)

`decorations`

Optional **decorations**: boolean

Whether the window should have borders and bars or not.

Defined in: [window.ts:2023](#)

`fileDropEnabled`

Optional **fileDropEnabled**: boolean

Whether the file drop is enabled or not on the webview. By default it is enabled.

Disabling it is required to use drag and drop on the frontend on Windows.

Defined in: [window.ts:2033](#)

`focus`

Optional **focus**: boolean

Whether the window will be initially focused or not.

Defined in: [window.ts:2011](#)

`fullscreen`

Optional **fullscreen**: boolean

Whether the window is in fullscreen mode or not.

Defined in: [window.ts:2009](#)

`height`

Optional **height**: number

The initial height.

Defined in: [window.ts:1995](#)

`hiddenTitle`

Optional **hiddenTitle**: boolean

If `true`, sets the window title to be hidden on macOS.

Defined in: [window.ts:2047](#)

`maxHeight`

Optional **maxHeight**: number

The maximum height. Only applies if `maxWidth` is also set.

Defined in: [window.ts:2003](#)

`maxWidth`

Optional **maxWidth**: number

The maximum width. Only applies if `maxHeight` is also set.

Defined in: [window.ts:2001](#)

`maximized`

Optional **maximized**: boolean

Whether the window should be maximized upon creation or not.

Defined in: [window.ts:2019](#)

`minHeight`

Optional **minHeight**: number

The minimum height. Only applies if `minWidth` is also set.

Defined in: [window.ts:1999](#)

`minWidth`

Optional **minWidth**: number

The minimum width. Only applies if `minHeight` is also set.

Defined in: [window.ts:1997](#)

`resizable`

Optional **resizable**: boolean

Whether the window is resizable or not.

Defined in: [window.ts:2005](#)

`skipTaskbar`

Optional **skipTaskbar**: boolean

Whether or not the window icon should be added to the taskbar.

Defined in: [window.ts:2027](#)

tabbingIdentifier

Optional **tabbingIdentifier**: string

Defines the window [tabbing identifier](#) on macOS.

Windows with the same tabbing identifier will be grouped together. If the tabbing identifier is not set, automatic tabbing will be disabled.

Defined in: [window.ts:2058](#)

theme

Optional **theme**: Theme

The initial window theme. Defaults to the system theme.

Only implemented on Windows and macOS 10.14+.

Defined in: [window.ts:2039](#)

title

Optional **title**: string

Window title.

Defined in: [window.ts:2007](#)

titleBarStyle

Optional **titleBarStyle**: TitleBarStyle

The style of the macOS title bar.

Defined in: [window.ts:2043](#)

transparent

Optional **transparent**: boolean

Whether the window is transparent or not. Note that on `macOS` this requires the `macos-private-api` feature flag, enabled under `tauri.conf.json > tauri > macOSPrivateApi`.

WARNING: Using private APIs on `macOS` prevents your application from being accepted to the [App Store](#).

Defined in: [window.ts:2017](#)

`url`

Optional `url`: `string`

Remote URL or local file path to open.

- URL such as `https://github.com/tauri-apps` is opened directly on a Tauri window.
- data: URL such as `data:text/html,<html>...` is only supported with the `window-data-url` Cargo feature for the `tauri` dependency.
- local file path or route such as `/path/to/page.html` or `/users` is appended to the application URL (the devServer URL on development, or `tauri://localhost/` and `https://tauri.localhost/` on production).

Defined in: [window.ts:1985](#)

`userAgent`

Optional `userAgent`: `string`

The user agent for the webview.

Defined in: [window.ts:2062](#)

`visible`

Optional `visible`: `boolean`

Whether the window should be immediately visible upon creation or not.

Defined in: [window.ts:2021](#)

`width`

Optional `width`: `number`

The initial width.

Defined in: [window.ts:1993](#)

x

Optional **x**: number

The initial vertical position. Only applies if **y** is also set.

Defined in: [window.ts:1989](#)

y

Optional **y**: number

The initial horizontal position. Only applies if **x** is also set.

Defined in: [window.ts:1991](#)

Type Aliases

CursorIcon

```
CursorIcon: default | crosshair | hand | arrow | move | text | wait | help | progress
| notAllowed | contextMenu | cell | verticalText | alias | copy | noDrop | grab | grabbing
| allScroll | zoomIn | zoomOut | eResize | nResize | neResize | nwResize | sResize
| seResize | swResize | wResize | ewResize | nsResize | neswResize | nwseResize | colR
esize | rowResize
```

Defined in: [window.ts:233](#)

FileDropEvent

```
FileDropEvent: { paths: string[]; type: hover } | { paths: string[]; type: drop } | {
type: cancel }
```

The file drop event types.

Defined in: [window.ts:101](#)

Theme

Theme: light | dark

Defined in: [window.ts:69](#)

TitleBarStyle

TitleBarStyle: visible | transparent | overlay

Defined in: [window.ts:70](#)

Variables

appWindow

appWindow: WebviewWindow

The WebviewWindow for the current window.

Defined in: [window.ts:1953](#)

Functions

availableMonitors

availableMonitors(): Promise < Monitor []>

Returns the list of all the monitors available on the system.

Example

```
import { availableMonitors } from '@tauri-apps/api/window';
const monitors = availableMonitors();
```

Since: 1.0.0

Returns: Promise < Monitor []>

currentMonitor

```
currentMonitor(): Promise < Monitor | null >
```

Returns the monitor on which the window currently resides. Returns `null` if current monitor can't be detected.

Example

```
import { currentMonitor } from '@tauri-apps/api/window';
const monitor = currentMonitor();
```

Since: 1.0.0

Returns: `Promise < Monitor | null >`

getAll

```
getAll(): WebviewWindow []
```

Gets a list of instances of `WebviewWindow` for all available webview windows.

Since: 1.0.0

Returns: `WebviewWindow []`

getCurrent

```
getCurrent(): WebviewWindow
```

Get an instance of `WebviewWindow` for the current webview window.

Since: 1.0.0

Returns: `WebviewWindow`

primaryMonitor

```
primaryMonitor(): Promise < Monitor | null >
```

Returns the primary monitor of the system. Returns `null` if it can't identify any monitor as a primary one.

Example

```
import { primaryMonitor } from '@tauri-apps/api/window';
const monitor = primaryMonitor();
```

Since: 1.0.0

Returns: `Promise<Monitor | null>`

[Go to TOC](#)

Introduction

The Tauri Bundler is a Rust harness to compile your binary, package assets, and prepare a final bundle.

It will detect your operating system and build a bundle accordingly. It currently supports:

- [Windows](#): .msi
- [macOS](#): .app, .dmg
- [Linux](#): .deb, .appimage

```
import Command from '@theme/Command'
```

To build and bundle your Tauri application into a single executable simply run the following command:

It will build your frontend (if configured, see `beforeBuildCommand`), compile the Rust binary, collect all external binaries and resources and finally produce neat platform-specific bundles and installers.

[Go to TOC](#)

Reducing App Size

With Tauri, we are working to reduce the environmental footprint of applications by using fewer system resources where available, providing compiled systems that don't need runtime evaluation, and offering guides so that engineers can go even smaller without sacrificing performance or security. By saving resources we are doing our part to help you help us save the planet -- which is the only bottom line that companies in the 21st Century should care about.

So if you are interested in learning how to improve your app size and performance, read on!

You can't improve what you can't measure

Before you can optimize your app, you need to figure out what takes up space in your app! Here are a couple of tools that can assist you with that:

- `cargo-bloat` - A Rust utility to determine what takes the most space in your app. It gives you an excellent, sorted overview of the most significant Rust functions.
- `cargo-expand` - `Macros` make your rust code more concise and easier to read, but they are also hidden size traps! Use `cargo-expand` to see what those macros generate under the hood.
- `rollup-plugin-visualizer` - A tool that generates beautiful (and insightful) graphs from your rollup bundle. Very convenient for figuring out what JavaScript dependencies contribute to your final bundle size the most.
- `rollup-plugin-graph` - You noticed a dependency included in your final frontend bundle, but you are unsure why? `rollup-plugin-graph` generates Graphviz-compatible visualizations of your entire dependency graph.

These are just a couple of tools that you might use. Make sure to check your frontend bundlers plugin list for more!

Checklist

1. [Minify Javascript](#)
2. [Optimize Dependencies](#)
3. [Optimize Images](#)
4. [Remove Unnecessary Custom Fonts](#)
5. [Allowlist Config](#)
6. [Rust Build-time Optimizations](#)
7. [Stripping](#)
8. [UPX](#)

Minify JavaScript

JavaScript makes up a large portion of a typical Tauri app, so it's important to make the JavaScript as light-weight as possible.

You can choose from a plethora of JavaScript bundlers; popular choices are [Vite](#), [webpack](#), and [rollup](#). All of them can produce minified JavaScript if configured correctly, so consult your bundler documentation for specific options. Generally speaking, you should make sure to:

Enable tree shaking

This option removes unused JavaScript from your bundle. All popular bundlers enable this by default.

Enable minification

Minification removes unnecessary whitespace, shortens variable names, and applies other optimizations. Most bundlers enable this by default; a notable exception is [rollup](#), where you need plugins like [rollup-plugin-terser](#) or [rollup-plugin-uglify](#).

Note: You can use minifiers like [terser](#) and [esbuild](#) as standalone tools.

Disable source maps

Source maps provide a pleasant developer experience when working with languages that compile to JavaScript, such as [TypeScript](#). As source maps tend to be quite large, you must disable them when building for production. They have no benefit to your end-user, so it's effectively dead weight.

Optimize Dependencies

Many popular libraries have smaller and faster alternatives that you can choose from instead.

Most libraries you use depend on many libraries themselves, so a library that looks inconspicuous at first glance might add **several megabytes** worth of code to your app.

You can use [Bundlephobia](#) to find the cost of JavaScript dependencies. Inspecting the cost of Rust dependencies is generally harder since the compiler does many optimizations.

If you find a library that seems excessively large, Google around, chances are someone else already had the same thought and created an alternative. A good example is [Moment.js](#) and it's [many alternatives](#).

But keep in mind: **The best dependency is no dependency**, meaning that you should always prefer language builtins over 3rd party packages.

Optimize Images

According to the [Http Archive](#), images are the [biggest contributor to website weight](#). So if your app includes images or icons, make sure to optimize them!

You can choose between a variety of manual options ([GIMP](#), [Photoshop](#), [Squoosh](#)) or plugins for your favorite frontend build tools ([vite-imagetools](#), [vite-plugin-imagemin](#), [image-minimizer-webpack-plugin](#)).

Do note that the `imagemin` library most of the plugins use is [officially unmaintained](#).

Use Modern Image Formats

Formats such as `webp` or `avif` offer size reductions of **up to 95%** compared to jpeg while maintaining excellent visual accuracy. You can use tools such as [Squoosh](#) to try different formats on your images.

Size Images Accordingly

No one appreciates you shipping the 6K raw image with your app, so make sure to size your image accordingly. Images that appear large on-screen should be sized larger than images that take up less screen space.

Don't Use Responsive Images

In a Web Environment, you are supposed to use [Responsive Images](#) to load the correct image size for each user dynamically. Since you are not dynamically distributing images over the web, using Responsive Images only needlessly bloats your app with redundant copies.

Remove Metadata

Images that were taken straight from a camera or stock photo side often include metadata about the camera and lens model or photographer. Not only are those wasted bytes, but metadata properties can also hold potentially sensitive information such as the time, day, and location of the photo.

Remove Unnecessary Custom Fonts

Consider not shipping custom fonts with your app and relying on system fonts instead. If you must ship custom fonts, make sure they are in modern, optimized formats such as `woff2`.

Fonts can be pretty big, so using the fonts already included in the Operating System reduces the footprint of your app. It also avoids FOUT (Flash of Unstyled Text) and makes your app feel more native since it uses the same font as all other apps.

If you must include custom fonts, make sure you include them in modern formats such as `woff2` as those tend to be much smaller than legacy formats.

Use so-called **System Font Stacks** in your CSS. There are a number of variations, but here are 3 basic ones to get you started:

Sans-Serif

```
font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Helvetica, Arial,
  sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji';
```

Serif

```
font-family: Iowan Old Style, Apple Garamond, Baskerville, Times New Roman, Droid
  Serif, Times, Source Serif Pro, serif, Apple Color Emoji, Segoe UI Emoji,
  Segoe
  UI Symbol;
```

Monospace

```
font-family: ui-monospace, SFMono-Regular, SF Mono, Menlo, Consolas, Liberation
Mono, monospace;
```

Allowlist Config

You can reduce the size of your app by only enabling the Tauri API features you need in the `allowlist` config.

The `allowlist` config determines what API features to enable; disabled features will **not be compiled into your app**. This is an easy way of shedding some extra weight.

An example from a typical `tauri.conf.json`:

```
{
  tauri: {
    allowlist: {
      all: false,
      fs: {
        writeFile: true
      },
      shell: {
        execute: true
      },
      dialog: {
        save: true
      }
    }
  }
}
```

Rust Build-Time Optimizations

Configure your cargo project to take advantage of Rust's size optimization features. [Why is a rust executable large?](#) provides an excellent explanation of why this matters and an in-depth walkthrough. At the same time, [Minimizing Rust Binary Size](#) is more up-to-date and has a couple of extra recommendations.

Rust is notorious for producing large binaries, but you can instruct the compiler to optimize the final executable's size.

Cargo exposes several options that determine how the compiler generates your binary. The recommended options for Tauri apps are these:

```
[profile.release]
panic = abort # Strip expensive panic clean-up logic
codegen-units = 1 # Compile crates one after another so the compiler can optimize
better
lto = true # Enables link to optimizations
opt-level = s # Optimize for binary size
```

There is also `opt-level = z` available to reduce the resulting binary size. `s` and `z` can sometimes be smaller than the other, so test it with your application!

We've seen smaller binary sizes from `s` for Tauri example applications, but real-world applications can always differ.

For a detailed explanation of each option and a bunch more, refer to the [Cargo books Profiles section](#).

Unstable Rust Compression Features

caution The following suggestions are all unstable features and require a nightly toolchain. See the [Unstable Features][cargo unstable features] documentation for more information on what this involves.

The following methods involve using unstable compiler features and require the rust nightly toolchain. If you don't have the nightly toolchain + `rust-src` nightly component added, try the following:

```
rustup toolchain install nightly
rustup component add rust-src --toolchain nightly
```

The Rust Standard Library comes precompiled. This means Rust is faster to install, but also that the compiler can't optimize the Standard Library. You can apply the optimization options for the rest of your binary + dependencies to the std with an unstable flag. This flag requires specifying your target, so know the target triple you are targeting.

```
cargo +nightly build --release -Z build-std --target x86_64-unknown-linux-gnu
```

If you are using `panic = abort` in your release profile optimizations, you need to make sure the `panic_abort` crate is compiled with std. Additionally, an extra std feature can further reduce the binary size. The following applies to both:

```
cargo +nightly build --release -Z build-std=std,panic_abort -Z build-std-features=panic_immediate_abort --target x86_64-unknown-linux-gnu
```

See the unstable documentation for more details about `-Z build-std` and `-Z build-std-features`.

Stripping

Use strip utilities to remove debug symbols from your compiled app.

Your compiled app includes so-called Debug Symbols that include function and variable names. Your end-users will probably not care about Debug Symbols, so this is a pretty surefire way to save some bytes!

The easiest way is to use the famous `strip` utility to remove this debugging information.

```
strip target/release/my_application
```

See your local `strip` manpage for more information and flags that can be used to specify what information gets stripped out from the binary.

info

Rust 1.59 now has a builtin version of `strip`! It can be enabled by adding the following to your `Cargo.toml`:

```
[profile.release]
strip = true # Automatically strip symbols from the binary.
```

UPX

UPX, **Ultimate Packer for eXecutables**, is a dinosaur amongst the binary packers. This 23-year old, well-maintained piece of kit is GPL-v2 licensed with a pretty liberal usage declaration. Our understanding of the licensing is that you can use it for any purposes (commercial or otherwise) without needing to change your license unless you modify the source code of UPX.

Maybe your target audience has very slow internet, or your app needs to fit on a tiny USB stick, and all the above steps haven't resulted in the savings you need. Fear not, as we have one last trick up our sleeves:

[UPX](#) compresses your binary and creates a self-extracting executable that decompresses itself at runtime.

caution You should know that this technique might flag your binary as a virus on Windows and macOS - so use at your own discretion, and as always, validate with [Frida] and do real distribution testing!

Usage on macOS

```
brew install upx
yarn tauri build
upx --ultra-brute src-tauri/target/release/bundle/macos/app.app/Contents/macOS/app
```

```
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2018
UPX 3.95           Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018
```

File size	Ratio	Format	Name
963140 ->	274448	28.50%	macho/amd64 app

[Go to TOC](#)

Cross-Platform Compilation

Tauri relies heavily on native libraries and toolchains, so meaningful cross-compilation is **not possible** at the current moment. The next best option is to compile utilizing a CI/CD pipeline hosted on something like [GitHub Actions](#), Azure Pipelines, GitLab, or other options. The pipeline can run the compilation for each platform simultaneously making the compilation and release process much easier.

For an easy setup, we currently provide [Tauri Action](#), a GitHub Action that runs on all the supported platforms, compiles your software, generates the necessary artifacts, and uploads them to a new GitHub release.

Tauri GitHub Action

Tauri Action leverages GitHub Actions to simultaneously build your application as a Tauri native binary for macOS, Linux, and Windows, and automates creating a GitHub release.

This GitHub Action may also be used as a testing pipeline for your Tauri app, guaranteeing compilation runs fine on all platforms for each pull request sent, even if you don't wish to create a new release.

info Code Signing

To setup code signing for both Windows and macOS on your workflow, follow the specific guide for each platform:

- [Windows Code Signing with GitHub Actions](#)
- [macOS Code Signing with GitHub Actions](#)

Getting Started

To set up Tauri Action you must first set up a GitHub repository. You can use this action on a repo that doesn't have Tauri configured since it automatically initializes Tauri before building and configuring it to use your artifacts.

Go to the Actions tab on your GitHub project and choose New workflow, then choose Set up a workflow yourself. Replace the file with the [Tauri Action production build workflow example](#). Alternatively, you may set up the workflow based on the [example lower on this page](#)

Configuration

You can configure Tauri with the `configPath`, `distPath` and `iconPath` options. See the actions Readme for details.

Custom Tauri CLI scripts can be run with the `tauriScript` option. So instead of running `yarn tauri build` or `npx tauri build`, `${tauriScript}` will be executed. This can be useful when you need custom build functionality such as when creating Tauri apps e.g. a `desktop:build` script.

When your app isn't on the root of the repo, use the `projectPath` input. You may modify the workflow name, change the triggers, and add more steps such as `npm run lint` or `npm run test`. The important part is that you keep the below line at the end of the workflow, since this runs the build script and releases the artifacts:

```
- uses: tauri-apps/tauri-action@v0
```

How to Trigger

The release workflow by default is triggered by pushes on the release branch. The action automatically creates a tag and title for the GitHub release using the application version specified in `tauri.config.json`.

You can also trigger the workflow on the push of a version tag such as `app-v0.7.0`. For this you can change the start of the release workflow:

```
name: publish
on:
  push:
    tags:
      - 'app-v*'
  workflow_dispatch:
```

Example Workflow

Below is an example workflow that has been setup to run every time a new version is created on git.

This workflow sets up the environment on Windows, Ubuntu, and macOS latest versions. Note under `jobs.release.strategy.matrix` the `platform` array which contains `macos-latest`, `ubuntu-latest`, and `windows-latest`.

The steps this workflow takes are:

1. Checkout the repository using `actions/checkout@v2`
2. Set up Node 16 using `actions/setup-node@v1`
3. Set up Rust using `actions-rs/toolchain@v1`
4. Installs all the dependencies and run the build script (for the web app)
5. Finally, it uses `tauri-apps/tauri-action@v0` to run `tauri build`, generate the artifacts, and create the GitHub release

```
name: Release
on:
  push:
    tags:
      - 'v*'
  workflow_dispatch:

jobs:
  release:
    strategy:
      fail-fast: false
      matrix:
        platform: [macos-latest, ubuntu-latest, windows-latest]
    runs-on: ${{ matrix.platform }}
```

```

steps:
  - name: Checkout repository
    uses: actions/checkout@v2

  - name: Node.js setup
    uses: actions/setup-node@v1
    with:
      node-version: 16

  - name: Rust setup
    uses: actions-rs/toolchain@v1
    with:
      toolchain: stable

  - name: Install dependencies (ubuntu only)
    if: matrix.platform == 'ubuntu-latest'
    run:
      sudo apt-get update
      sudo apt-get install -y libgtk-3-dev webkit2gtk-4.0 libappindicator3-dev
      librsvg2-dev patchelf

  - name: Install app dependencies and build web
    run: yarn && yarn build

  - name: Build the app
    uses: tauri-apps/tauri-action@v0

    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

    with:
      tagName: v__VERSION__ # tauri-action replaces \_\_VERSION\_\_ with the
      app version
      releaseName: 'v__VERSION__'
      releaseBody: 'See the assets to download this version and install.'
      releaseDraft: true
      prerelease: false

```

GitHub Environment Token

The GitHub Token is automatically issued by GitHub for each workflow run without further configuration, which means there is no risk of secret leakage. This token however only has read permissions by default and you may get a Resource not accessible by integration error when running the workflow. If this happens, you may need to add write permissions to this token. To do this go to your GitHub Project Settings, and then select Actions, scroll down to Workflow permissions and check Read and write permissions.

You can see the GitHub Token being passed to the workflow below:

```

env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

```

Usage Notes

Make sure to check the [documentation for GitHub Actions](#) to understand better how this workflow works. Take care to read the [Usage limits, billing, and administration](#) documentation for GitHub Actions. Some project templates may already implement this GitHub action workflow, such as [tauri-svelte-template](#). You can use this action on a repo that doesn't have Tauri configured. Tauri automatically initializes before building and configuring it to use your web artifacts.


```
import TauriBuild from './tauri-build.md'
```

Linux Bundle

Limitations

Core libraries such as glibc frequently break compatibility with older systems. For this reason, you must build your Tauri application using the oldest base system you intend to support. A relatively old system such as Ubuntu 18.04 is more suited than Ubuntu 22.04, as the binary compiled on Ubuntu 22.04 will have a higher requirement of the glibc version, so when running on an older system, you will face a runtime error like `/usr/lib/libc.so.6: version 'GLIBC_2.33' not found`. We recommend using a Docker container or GitHub Actions to build your Tauri application for Linux.

See the issues [rust-lang/rust#1355](#) and [rust-lang/rust#57497](#), in addition to the [AppImage guide](#) for more information.

Debian

Tauri allows your app to be packaged as a `.deb` (Debian package) file. The Tauri CLI bundles your application binary and additional resources in this format if you build on Linux. Please note that `.deb` packages can **only be created on Linux** as cross-compilation doesn't work yet.

The stock Debian package generated by the Tauri bundler has everything you need to ship your application to Debian-based Linux distributions, defining your application's icons, generating a Desktop file, and specifying the dependencies `libwebkit2gtk-4.0-37` and `libgtk-3-0`, along with `libappindicator3-1` if your app uses the system tray.

GUI apps on macOS and Linux do not inherit the `\$PATH` from your shell dotfiles (`~/.bashrc`, `~/.bash_profile`, `~/.zshrc`, etc). Check out Tauri's [fix-path-env-rs](<https://github.com/tauri-apps/fix-path-env-rs>) crate to fix this issue.

Custom Files

Tauri exposes a few configurations for the Debian package in case you need more control.

If your app depends on additional system dependencies you can specify them in `tauri.conf.json > tauri > bundle > deb > depends`.

To include custom files in the Debian package, you can provide a list of files or folders in `tauri.conf.json > tauri > bundle > deb > files`. The configuration object maps the path in the Debian package to the path to the file on your filesystem, relative to the `tauri.conf.json` file. Here's an example configuration:

```
{
  tauri: {
    bundle: {
      deb: {
```

```
        files: {
          '/usr/share/README.md': '../README.md', // copies the README.md file to
          '/usr/share/README.md'
          'usr/share/assets': '../assets/' // copies the entire assets directory to
          '/usr/share/assets'
        }
      }
    }
  }
}
```

If you need to bundle files in a cross-platform way, check Tauri's [resource](#) and [sidecar](#) mechanisms.

AppImage

AppImage is a distribution format that does not rely on the system installed packages and instead bundles all dependencies and files needed by the application. For this reason, the output file is larger but easier to distribute since it is supported on many Linux distributions and can be executed without installation. The user just needs to make the file executable (`chmod a+x MyProject.AppImage`) and can then run it (`./MyProject.AppImage`).

AppImages are convenient, simplifying the distribution process if you cannot make a package targeting the distribution's package manager. Still, you should carefully use it as the file size grows from the 2-6MBs range to 70+MBs.

caution

If your app plays audio/video you need to enable `tauri.conf.json > tauri > bundle > appimage > bundleMediaFramework`. This will increase the size of the AppImage bundle to include additional `gstreamer` files needed for media playback. This flag is currently only supported on Ubuntu build systems.

```
import TauriBuild from './tauri-build.md'
```

macOS Bundle

Tauri applications for macOS are distributed either with an [Application Bundle](#) (`.app` file) or an Apple Disk Image (`.dmg` file). The Tauri CLI automatically bundles your application code in these formats, providing options to codesign and notarize your application. Please note that `.app` and `.dmg` bundles can **only be created on macOS** as cross-compilation doesn't work yet.

GUI apps on macOS and Linux do not inherit the `$PATH` from your shell dotfiles (`.bashrc`, `.bash_profile`, `.zshrc`, etc). Check out Tauri's [fix-path-env-rs](#) crate to fix this issue.

Setting a Minimum System Version

The minimum version of the operating system required for a Tauri app to run in macOS is `10.13`. If you need support for newer macOS APIs like `window.print` that is only supported from macOS version `11.0` onwards, you can change the `tauri.bundle.macOS.minimumSystemVersion`. This will in turn set the `Info.plist LSMinimumSystemVersion` property and the `MACOSX_DEPLOYMENT_TARGET` environment variable.

Binary Targets

You can compile your application targeting Apple Silicon, Intel-based Mac computers, or universal macOS binaries. By default, the CLI builds a binary targeting your machine's architecture, but you can configure a different target using the `--target` flag:

- `tauri build --target aarch64-apple-darwin`: targets Apple silicon machines.
- `tauri build --target x86_64-apple-darwin`: targets Intel-based machines.
- `tauri build --target universal-apple-darwin`: produces a [universal macOS binary](#) which runs on both Apple silicon and Intel-based Macs.

While Apple silicon machines can run applications compiled for Intel-based Macs through a translation layer called [Rosetta](#), this leads to a reduction in performance due to processor instruction translations. It is common practice to let the user choose the correct target when downloading the app, but you can also choose to distribute a [Universal Binary](#). Universal Binaries include both `aarch64` and `x86_64` executables, giving you the best experience on both architectures. Note, however, that this increases your bundle size significantly.

Application Bundle Customization

The Tauri configuration file provides the following options to customize your application bundle:

- **Bundle name:** Your app's human-readable name. Configured by the `package.productName` property.

- **Bundle version:** Your app's version. Configured by the `package.version` property.
- **Application category:** The category that describes your app. Configured by the `tauri.bundle.category` property. You can see a list of macOS categories [here](#).
- **Copyright:** A copyright string associated with your app. Configured by the `tauri.bundle.copyright` property.
- **Bundle icon:** Your app's icon. Uses the first `.icns` file listed in the `tauri.bundle.icon` array.
- **Minimum system version:** Configured by the `tauri.bundle.macOS.minimumSystemVersion` property.
- **DMG license file:** A license that is added to the `.dmg` file. Configure by the `tauri.bundle.macOS.license` property.
- **Entitlements.plist file:** Entitlements control what APIs your app will have access to. Configured by the `tauri.bundle.macOS.entitlements` property.
- **Exception domain:** an insecure domain that your application can access such as a `localhost` or a remote `http` domain. It is a convenience configuration around `NSAppTransportSecurity > NSExceptionDomains` setting `NSEExceptionAllowsInsecureHTTPLoads` and `NSIncludesSubdomains` to true. See `tauri.bundle.macOS.exceptionDomain` for more information.

info

These options generate the application bundle [Info.plist file](#). You can extend the generated file with your own `Info.plist` file stored in the Tauri folder (`src-tauri` by default). The CLI merges both `.plist` files in production, and the core layer embeds it in the binary during development.

[Go to TOC](#)

Embedding Additional Files

You may need to include additional files in your application bundle that aren't part of your frontend (your `distDir`) directly or which are too big to be inlined into the binary. We call these files `resources`.

To bundle the files of your choice, you can add the `resources` property to the `tauri > bundle` object in your `tauri.conf.json` file.

See more about `tauri.conf.json` configuration [here](#).

`resources` expects a list of strings targeting files either with absolute or relative paths. It supports glob patterns in case you need to include multiple files from a directory.

Here is a sample to illustrate the configuration. This is not a complete `tauri.conf.json` file:

tauri.conf.json

```
{
  tauri: {
    bundle: {
      resources: [
        '/absolute/path/to/textfile.txt',
        'relative/path/to/jsonfile.json',
        'resources/*'
      ],
      allowlist: {
        fs: {
          scope: ['$RESOURCE/*']
        }
      }
    }
  }
}
```

Absolute paths and paths containing parent components (`..`) can only be allowed via `$RESOURCE/*`. Relative paths like `path/to/file.txt` can be allowed explicitly via `$RESOURCE/path/to/file.txt`.

Accessing files in JavaScript

In this example we want to bundle additional i18n json files that look like this:

de.json

```
{
  hello: Guten Tag!,
  bye: Auf Wiedersehen!
}
```

In this case, we store these files in a `lang` directory next to the `tauri.conf.json`. For this we add `lang/*` to `resources` and `$RESOURCE/lang/*` to the `fs` scope as shown above.

Note that you must configure the allowlist to enable `path > all` and the `fs APIs` you need, in this example `fs > readTextFile`.

```
import { resolveResource } from '@tauri-apps/api/path'
// alternatively, use `window.__TAURI__.path.resolveResource`
import { readTextFile } from '@tauri-apps/api/fs'
// alternatively, use `window.__TAURI__.fs.readTextFile`

// `lang/de.json` is the value specified on `tauri.conf.json > tauri > bundle > resources`
const resourcePath = await resolveResource('lang/de.json')
const langDe = JSON.parse(await readTextFile(resourcePath))

console.log(langDe.hello) // This will print 'Guten Tag!' to the devtools console
```

Accessing files in Rust

This is based on the example above. On the Rust side you need an instance of the `PathResolver` which you can get from `App` and `AppHandle`:

```
tauri::Builder::default()
    .setup(|app| {
        let resource_path = app.path_resolver()
            .resolve_resource(lang/de.json)
            .expect("failed to resolve resource");

        let file = std::fs::File::open(&resource_path).unwrap();
        let lang_de: serde_json::Value = serde_json::from_reader(file).unwrap();

        println!("{}: {}", lang_de.get("hello").unwrap()); // This will print 'Guten Tag!' to
        // the terminal
    })
    .ok()
```

```
#[tauri::command]
fn hello(handle: tauri::AppHandle) -> String {
    let resource_path = handle.path_resolver()
        .resolve_resource(lang/de.json)
        .expect("failed to resolve resource");

    let file = std::fs::File::open(&resource_path).unwrap();
    let lang_de: serde_json::Value = serde_json::from_reader(file).unwrap();

    lang_de.get("hello").unwrap()
}
```

[Go to TOC](#)

Embedding External Binaries

You may need to embed depending binaries to make your application work or prevent users from installing additional dependencies (e.g., Node.js or Python). We call this binary a `sidecar`.

To bundle the binaries of your choice, you can add the `externalBin` property to the `tauri > bundle` object in your `tauri.conf.json`.

See more about `tauri.conf.json` configuration [here](#).

`externalBin` expects a list of strings targeting binaries either with absolute or relative paths.

Here is a sample to illustrate the configuration. This is not a complete `tauri.conf.json` file:

```
{
  tauri: {
    bundle: {
      externalBin: [
        '/absolute/path/to/sidecar',
        'relative/path/to/binary',
        'binaries/my-sidecar'
      ]
    },
    allowlist: {
      shell: {
        sidecar: true,
        scope: [
          { name: '/absolute/path/to/sidecar', sidecar: true },
          { name: 'relative/path/to/binary', sidecar: true },
          { name: 'binaries/my-sidecar', sidecar: true }
        ]
      }
    }
  }
}
```

A binary with the same name and a `-$TARGET_TRIPLE` suffix must exist on the specified path. For instance, `externalBin: [binaries/my-sidecar]` requires a `src-tauri/binaries/my-sidecar-x86_64-unknown-linux-gnu` executable on Linux. You can find the current platform's target triple by running the following command:

```
rustc -Vv | grep host | cut -f2 -d' '
```

Here's a Node.js script to append the target triple to a binary:

```
const execa = require('execa')
const fs = require('fs')

let extension = ''
if (process.platform === 'win32') {
  extension = '.exe'
}

async function main() {
```

```

const rustInfo = (await execa('rustc', ['-vV'])).stdout
const targetTriple = /host: (\S+)/g.exec(rustInfo)[1]
if (!targetTriple) {
  console.error('Failed to determine platform target triple')
}
fs.renameSync(
  `src-tauri/binaries/sidecar${extension}`,
  `src-tauri/binaries/sidecar-${targetTriple}${extension}`
)
}

main().catch((e) => {
  throw e
})
}

```

Running it from JavaScript

In the JavaScript code, import the `Command` class on the `shell` module and use the `sidecar` static method.

Note that you must configure the allowlist to enable `shell > sidecar` and configure all binaries in `shell > scope`.

```

import { Command } from '@tauri-apps/api/shell'
// alternatively, use `window.__TAURI__.shell.Command`
// `binaries/my-sidecar` is the EXACT value specified on `tauri.conf.json > tauri
> bundle > externalBin`
const command = Command.sidecar('binaries/my-sidecar')
const output = await command.execute()

```

Running it from Rust

On the Rust side, import the `Command` struct from the `tauri::api::process` module:

```

// `new_sidecar()` expects just the filename, NOT the whole path like in
// JavaScript
let (mut rx, mut child) = Command::new_sidecar(my-sidecar)
  .expect("Failed to create `my-sidecar` binary command")
  .spawn()
  .expect("Failed to spawn sidecar");

tauri::async_runtime::spawn(async move {
  // read events such as stdout
  while let Some(event) = rx.recv().await {
    if let CommandEvent::Stdout(line) = event {
      window
        .emit(message, Some(format!("{}: {}", line)))
        .expect("Failed to emit event");
      // write to stdin
      child.write(message from Rust\n.as_bytes()).unwrap();
    }
  }
});

```

Note that you must enable the **process-command-api** Cargo feature (Tauri's CLI will do this for you once you changed the config):

```
# Cargo.toml
[dependencies]
tauri = { version = 1, features = [process-command-api, ...] }
```

Using Node.js on a Sidecar

The Tauri [sidecar example](#) demonstrates how to use the sidecar API to run a Node.js application on Tauri. It compiles the Node.js code using [pkg](#) and uses the scripts above to run it.

```
import Command from '@theme/Command'
```

Windows Installer

Tauri applications for Windows are distributed as Microsoft Installers (`.msi` files). The Tauri CLI bundles your application binary and additional resources. Please note that `.msi` installers can **only be created on Windows** as cross-compilation doesn't work yet. This guide provides information about available customization options for the installer.

To build and bundle your Tauri application into a single executable simply run the following command:

It will build your Frontend, compile the Rust binary, collect all external binaries and resources and finally produce neat platform-specific bundles and installers.

Building for 32-bit or ARM

The Tauri CLI compiles your executable using your machine's architecture by default. Assuming that you're developing on a 64-bit machine, the CLI will produce 64-bit applications.

If you need to support **32-bit** machines, you can compile your application with a **different Rust target** using the `--target` flag:

```
tauri build --target i686-pc-windows-msvc
```

By default, Rust only installs toolchains for your machine's target, so you need to install the 32-bit Windows toolchain first: `rustup target add i686-pc-windows-msvc`.

If you need to build for **ARM64** you first need to install additional build tools. To do this, open `Visual Studio Installer`, click on Modify, and in the Individual Components tab install the C++ ARM64 build tools. At the time of writing, the exact name in VS2022 is `MSVC v143 - VS 2022 C++ ARM64 build tools (Latest)`.

Now you can add the rust target with `rustup target add aarch64-pc-windows-msvc` and then use the above-mentioned method to compile your app:

```
tauri build --target aarch64-pc-windows-msvc
```

Supporting Windows 7

By default, the Microsoft Installer does not work on Windows 7 because it needs to download the Webview2 bootstrapper if not installed (which might fail if TLS 1.2 is not enabled in the operating system). Tauri includes an option to embed the Webview2 bootstrapper (see the [Embedding the Webview2 Bootstrapper](#) section below).

Additionally, to use the Notification API in Windows 7, you need to enable the `windows7-compat` Cargo feature:

Cargo.toml

```
[dependencies]
tauri = { version = 1, features = [ windows7-compat ] }
```

Webview2 Installation Options

The Windows Installer by default downloads the Webview2 bootstrapper and executes it if the runtime is not installed. Alternatively, you can embed the bootstrapper, embed the offline installer, or use a fixed Webview2 runtime version. See the following table for a comparison between these methods:

Installation Method	Requires Internet Connection?	Additional Installer Size	Notes
---------------------	-------------------------------	---------------------------	-------

downloadBootstrapper	Yes	0MB	<code>Default</code>
----------------------	-----	-----	----------------------

Results in a smaller installer size, but is not recommended for Windows 7 deployment.

embedBootstrapper	Yes	~1.8MB	Better support on Windows 7.
offlineInstaller	No	~127MB	Embeds Webview2 installer. Recommended for offline environments
fixedVersion	No	~180MB	Embeds a fixed Webview2 version
skip	No	0MB	⚠ Not recommended

Does not install the Webview2 as part of the Windows Installer.

info

On Windows 10 (April 2018 release or later) and Windows 11, the Webview2 runtime is distributed as part of the operating system.

Downloaded Bootstrapper

This is the default setting for building the Windows Installer. It downloads the bootstrapper and run it. Requires internet connection but results in a smaller installer size. This is not recommended if you're going to be distributing to Windows 7.

tauri.config.json

```
{
  tauri: {
    bundle: {
      windows: {
        webviewInstallMode: {
          type: downloadBootstrapper
        }
      }
    }
  }
}
```

Embedded Bootstrapper

To embed the Webview2 Bootstrapper, set the `webviewInstallMode` to `embedBootstrapper`. This increases the installer size by around 1.8MB, but increases compatibility with Windows 7 systems.

tauri.config.json

```
{
  tauri: {
    bundle: {
      windows: {
        webviewInstallMode: {
          type: embedBootstrapper
        }
      }
    }
  }
}
```

Offline Installer

To embed the Webview2 Bootstrapper, set the `webviewInstallMode` to `offlineInstaller`. This increases the installer size by around 127MB, but allows your application to be installed even if an internet connection is not available.

tauri.config.json

```
{
  tauri: {
    bundle: {
      windows: {
        webviewInstallMode: {
          type: offlineInstaller
        }
      }
    }
  }
}
```

Fixed Version

Using the runtime provided by the system is great for security as the webview vulnerability patches are managed by Windows. If you want to control the Webview2 distribution on each of your applications (either to manage the release patches yourself or distribute applications on environments where an internet connection might not be available) Tauri can bundle the runtime files for you.

caution Distributing a fixed Webview2 Runtime version increases the Windows Installer by around 180MB.

1. Download the Webview2 fixed version runtime from [Microsoft's website](#). In this example, the downloaded filename is `Microsoft.WebView2.FixedVersionRuntime.98.0.1108.50.x64.cab`
2. Extract the file to the core folder:

```
Expand .\Microsoft.WebView2.FixedVersionRuntime.98.0.1108.50.x64.cab -F:* ./src-
```

3. Configure the Webview2 runtime path in `tauri.conf.json`:

tauri.config.json

```
{
  tauri: {
    bundle: {
      windows: {
        webviewInstallMode: {
          type: fixedRuntime,
          path: ./Microsoft.WebView2.FixedVersionRuntime.98.0.1108.50.x64/
        }
      }
    }
  }
}
```

4. Run `tauri build` to produce the Windows Installer with the fixed Webview2 runtime.

Skipping Installation

You can remove the Webview2 Runtime download check from the installer by setting `webviewInstallMode` to `skip`. Your application WILL NOT work if the user does not have the runtime installed.

warning Your application WILL NOT work if the user does not have the runtime installed and won't attempt to install it.

`tauri.config.json`

```
{
  tauri: {
    bundle: {
      windows: {
        webviewInstallMode: {
          type: skip
        }
      }
    }
  }
}
```

Customizing the Installer

The Windows Installer package is built using the [WiX Toolset v3](#). Currently, you can change it by using a custom WiX source code (an XML file with a `.wxs` file extension) or through WiX fragments.

Replacing the Installer Code with a Custom WiX File

The Windows Installer XML defined by Tauri is configured to work for the common use case of simple web-view-based applications (you can find it [here](#)). It uses `handlebars` so the Tauri CLI can brand your installer according to your `tauri.conf.json` definition. If you need a completely different installer, a custom template file can be configured on `tauri.bundle.windows.wix.template`.

Extending the Installer with WiX Fragments

A [WiX fragment](#) is a container where you can configure almost everything offered by WiX. In this example, we will define a fragment that writes two registry entries:

```

<?xml version=1.0 encoding=utf-8?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Fragment>
    <!-- these registry entries should be installed
        to the target user's machine -->
    <DirectoryRef Id="TARGETDIR">
      <!-- groups together the registry entries to be installed -->
      <!-- Note the unique `Id` we provide here -->

      <Component Id="MyFragmentRegistryEntries" Guid="*">
        <!-- the registry key will be under
            HKEY_CURRENT_USER\Software\MyCompany\MyApplicationName -->
        <!-- Tauri uses the second portion of the
            bundle identifier as the `MyCompany` name
            (e.g. `tauri-apps` in `com.tauri-apps.test`) -->
        <RegistryKey
          Root="HKCU"
          Key="Software\MyCompany\MyApplicationName"
          Action="createAndRemoveOnUninstall"
        >
          <!-- values to persist on the registry -->
          <RegistryValue
            Type="integer"
            Name="SomeIntegerValue"
            Value="1"
            KeyPath="yes"
          />
          <RegistryValue Type="string" Value="Default Value" />
        </RegistryKey>
      </Component>
    </DirectoryRef>
  </Fragment>
</Wix>

```

Save the fragment file with the `.wxs` extension somewhere in your project and reference it on `tauri.conf.json`:

```
{
  "tauri": {
    "bundle": {
      "windows": {
        "wix": {
          "fragmentPaths": ["./path/to/registry.wxs"],
          "componentRefs": ["MyFragmentRegistryEntries"]
        }
      }
    }
  }
}
```

Note that `ComponentGroup`, `Component`, `FeatureGroup`, `Feature` and `Merge` element ids must be referenced on the `wix` object of `tauri.conf.json` on the `componentGroupRefs`, `componentRefs`, `featureGroupRefs`, `featureRefs` and `mergeRefs` respectively to be included in the installer.

Internationalization

The Windows Installer is built using the `en-US` language by default. Internationalization (i18n) can be configured using the `tauri.bundle.windows.wix.language` property, defining the languages Tauri should build an installer against. You can find the language names to use in the Language-Culture column on [Microsoft's website](#).

Compiling an Installer for a Single Language

To create a single installer targeting a specific language, set the `language` value to a string:

```
{
  tauri: {
    bundle: {
      windows: {
        wix: {
          language: "fr-FR"
        }
      }
    }
  }
}
```

Compiling an Installer for Each Language in a List

To compile an installer targeting a list of languages, use an array. A specific installer for each language will be created, with the language key as a suffix:

```
{
  tauri: {
    bundle: {
      windows: {
        wix: {
          language: ["en-US", "pt-BR", "fr-FR"]
        }
      }
    }
  }
}
```

Configuring the Installer for Each Language

A configuration object can be defined for each language to configure localization strings:

```
{
  tauri: {
    bundle: {
      windows: {
        wix: {
          language: {
            "en-US": null,
            "pt-BR": {
              localePath: "./wix/locales/pt-BR.wxl"
            }
          }
        }
      }
    }
  }
}
```

```

        }
    }
}

```

The `localePath` property defines the path to a language file, a XML configuring the language culture:

```

<WixLocalization
    Culture=en-US
    xmlns="http://schemas.microsoft.com/wix/2006/localization"
>
    <String Id="LaunchApp"> Launch MyApplicationName </String>
    <String Id="DowngradeErrorMessage">
        A newer version of MyApplicationName is already installed.
    </String>
    <String Id="PathEnvVarFeature">
        Add the install location of the MyApplicationName executable to
        the PATH system environment variable. This allows the
        MyApplicationName executable to be called from any location.
    </String>
    <String Id="InstallAppFeature">
        Installs MyApplicationName.
    </String>
</WixLocalization>

```

The `WixLocalization` element's `Culture` field must match the configured language.

Currently, Tauri references the following locale strings: `LaunchApp`, `DowngradeErrorMessage`, `PathEnvVarFeature` and `InstallAppFeature`. You can define your own strings and reference them on your custom template or fragments with `!(loc.TheStringId)`. See the [WiX localization documentation](#) for more information.

[Go to TOC](#)

```
import Command from '@theme/Command'
```

Application Debugging

With all the moving pieces in Tauri, you may run into a problem that requires debugging. There are many locations where error details are printed, and Tauri includes some tools to make the debugging process more straightforward.

Rust Console

The first place to look for errors is in the Rust Console. This is in the terminal where you ran, e.g., `tauri dev`. You can use the following code to print something to that console from within a Rust file:

```
println!(Message from Rust: {}, msg);
```

Sometimes you may have an error in your Rust code, and the Rust compiler can give you lots of information. If, for example, `tauri dev` crashes, you can rerun it like this on Linux and macOS:

```
RUST_BACKTRACE=1 tauri dev
```

or like this on Windows:

```
set RUST_BACKTRACE=1
tauri dev
```

This command gives you a granular stack trace. Generally speaking, the Rust compiler helps you by giving you detailed information about the issue, such as:

```
error[E0425]: cannot find value `sun` in this scope
--> src/main.rs:11:5
   |
11 |     sun += i.to_string().parse::<u64>().unwrap();
   |     ^^^ help: a local variable with a similar name exists: `sum`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0425`.
```

WebView Console

Right-click in the WebView, and choose `Inspect Element`. This opens up a web-inspector similar to the Chrome or Firefox dev tools you are used to. You can also use the `Ctrl + Shift + i` shortcut on Linux and Windows, and `Command + Option + i` on macOS to open the inspector.

The inspector is platform-specific, rendering the webkit2gtk WebInspector on Linux, Safari's inspector on macOS and the Microsoft Edge DevTools on Windows.

Opening Devtools Programmatically

You can control the inspector window visibility by using the `Window::open_devtools` and `Window::close_devtools` functions:

```
use tauri::Manager;
tauri::Builder::default()
    .setup(|app| {
        #[cfg(debug_assertions)] // only include this code on debug builds
        {
            let window = app.get_window(main).unwrap();
            window.open_devtools();
            window.close_devtools();
        }
        Ok(())
    });
});
```

Using the Inspector in Production

By default, the inspector is only enabled in development and debug builds unless you enable it with a Cargo feature.

Create a Debug Build

To create a debug build, run the `tauri build --debug` command.

<Command name=build --debug />

Like the normal build and dev processes, building takes some time the first time you run this command but is significantly faster on subsequent runs. The final bundled app has the development console enabled and is placed in `src-tauri/target/debug/bundle`.

You can also run a built app from the terminal, giving you the Rust compiler notes (in case of errors) or your `println` messages. Browse to the file `src-tauri/target/(release|debug)/[app name]` and run it directly in your console or double-click the executable itself in the filesystem (note: the console closes on errors with this method).

Enable Devtools Feature

warning

The devtools API is private on macOS. Using private APIs on macOS prevents your application from being accepted to the App Store.

To enable the devtools in production builds, you must enable the `devtools` Cargo feature in the `src-tauri/Cargo.toml` file:

```
[dependencies]
tauri = { version = ..., features = [..., devtools] }
```

Debugging the Core Process

The Core process is powered by Rust so you can use GDB or LLDB to debug it. You can follow the [Debugging in VS Code](#) guide to learn how to use the LLDB VS Code Extension to debug the Core Process of Tauri applications.

Debugging in VS Code

This guide describes how to setup debugging in VS Code for the [Core Process in Tauri applications](#).

Setup

Install the `vscode-lldb` extension.

Configure launch.json

Create a `.vscode/launch.json` file and paste the below JSON contents into it:

`.vscode/launch.json`

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    version: "0.2.0",
    configurations: [
        {
            type: "lldb",
            request: "launch",
            name: "Tauri Development Debug",
            cargo: {
                args: [
                    "build",
                    "--manifest-path=./src-tauri/Cargo.toml",
                    "--no-default-features"
                ],
                // task for the `beforeDevCommand` if used, must be configured in
                // .vscode/tasks.json
                preLaunchTask: "ui:dev"
            },
            {
                type: "lldb",
                request: "launch",
                name: "Tauri Production Debug",
                cargo: {
                    args: ["build", "--release", "--manifest-path=./src-tauri/Cargo.toml"]
                },
                // task for the `beforeBuildCommand` if used, must be configured in
                // .vscode/tasks.json
                preLaunchTask: "ui:build"
            }
        ]
}
```

This uses `cargo` directly to build the Rust application and load it in both development and production modes.

Note that it does not use the Tauri CLI, so exclusive CLI features are not executed. The `beforeDevCommand` and `beforeBuildCommand` scripts must be executed beforehand or configured as a task in the `preLaunchTask` field. Below is an example `.vscode/tasks.json` file that has two tasks, one for a `beforeDevCommand` that spawns a development server and one for `beforeBuildCommand`:

`.vscode/tasks.json`

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  version: "2.0.0",
  tasks: [
    {
      label: "ui:dev",
      type: "shell",
      // `dev` keeps running in the background
      // ideally you should also configure a `problemMatcher`
      // see https://code.visualstudio.com/docs/editor/tasks#_can-a-background-
      task-be-used-as-a-prelaunchtask-in-launchjson
      isBackground: true,
      // change this to your `beforeDevCommand`:
      command: "yarn",
      args: ["dev"]
    },
    {
      label: "ui:build",
      type: "shell",
      // change this to your `beforeBuildCommand`:
      command: "yarn",
      args: ["build"]
    }
  ]
}
```

Now you can set breakpoints in `src-tauri/src/main.rs` or any other Rust file and start debugging by pressing `F5`.

[Go to TOC](#)

```
import Command from '@theme/Command'
```

Development Cycle

1. Start Your Dev server

Now that you have everything set up, you should start your application development server provided by your UI framework or bundler (assuming you're using one, of course).

Every framework has its own development tooling. It is outside of the scope of this document to cover them all or stay up to date.

2. Start Tauri Development Window

The first time you run this command, the Rust package manager takes several minutes to download and build all the required packages. Since they are cached, subsequent builds are much faster, as only your code needs rebuilding.

Once Rust has finished building, the webview opens, displaying your web app. You can make changes to your web app, and if your tooling enables it, the webview should update automatically, just like a browser. When you make changes to your Rust files, they are rebuilt automatically, and your app automatically restarts.

info About Cargo.toml and Source Control

In your project repository, you **SHOULD** commit the `src-tauri/Cargo.lock` along with the `src-tauri/Cargo.toml` to git because Cargo uses the lockfile to provide deterministic builds. As a result, it is recommended that all applications check in their `Cargo.lock`. You **SHOULD NOT** commit the `src-tauri/target` folder or any of its contents.

[Go to TOC](#)

```
import Tabs from '@theme/Tabs' import TabItem from '@theme/TabItem'
```

Updating Dependencies

Update npm Packages

If you are using the `tauri` package:

```
npm install @tauri-apps/cli@latest @tauri-apps/api@latest
yarn upgrade @tauri-apps/cli @tauri-apps/api --latest
yarn up @tauri-apps/cli @tauri-apps/api
pnpm update @tauri-apps/cli @tauri-apps/api --latest
```

You can also detect what the latest version of Tauri is on the command line, using:

```
npm outdated @tauri-apps/cli
yarn outdated @tauri-apps/cli
pnpm outdated @tauri-apps/cli
```

Alternatively, if you are using the `vue-cli-plugin-tauri` approach:

```
npm install vue-cli-plugin-tauri@latest
yarn upgrade vue-cli-plugin-tauri --latest
yarn up vue-cli-plugin-tauri
pnpm update vue-cli-plugin-tauri --latest
```

Update Cargo Packages

You can check for outdated packages with `cargo outdated` or on the crates.io pages: [tauri](#) / [tauri-build](#).

Go to `src-tauri/Cargo.toml` and change `tauri` and `tauri-build` to

```
[build-dependencies]
tauri-build = %version%
[dependencies]
tauri = { version = %version% }
```

where `%version%` is the corresponding version number from above.

Then do the following:

```
cd src-tauri  
cargo update
```

Alternatively, you can run the `cargo upgrade` command provided by `cargo-edit` which does all of this automatically.

[Go to TOC](#)

```
import Command from '@theme/Command'
```

App Publishing

1. Build Your Web App

Now that you are ready to package your project, you need to run your framework's or bundler's build command (assuming you're using one, of course).

Every framework has its publishing tooling. It is outside of the scope of this document to treat them all or keep them up to date.

2. Bundle your application with Tauri

This command embeds your web assets into a single binary with your Rust code. The binary itself will be located in `src-tauri/target/release/[app name]`, and installers will be located in `src-tauri/target/release/bundle/`.

Like the `tauri dev` command, the first time you run this, it takes some time to collect the Rust crates and build everything - but on subsequent runs, it only needs to rebuild your app's code, which is much quicker.

Code Signing Linux packages

This guide provides information on code signing for Linux packages.

Requirements

- gpg or gpg2

A key for signing must be prepared. A new one can be generated using:

```
gpg2 --full-gen-key
```

Please refer to the gpg or gpg2 documentation for additional information. You should take additional care to backup your private and public keys in a secure location.

Signing for AppImages

You can embed a signature in the AppImage by setting the following environment variables:

- **SIGN**: set to `1` to sign the AppImage.
- **SIGN_KEY**: optional variable to use a specific GPG Key ID for signing.
- **APPIMAGETOOL_SIGN_PASSPHRASE**: the signing key password. If unset, gpg shows a dialog so you can input it. You must set this when running automated tasks.

You can display the signature embedded in the AppImage by running the following command:

```
./src-tauri/target/release/bundle/appimage/$APPNAME_$VERSION_amd64.AppImage -- appimage-signature
```

Note that you need to change the `$APPNAME` and `$VERSION` values with the correct ones based on your configuration.

caution The signature is not verified

AppImage does not validate the signature, so you can't rely on it to check whether the file has been tampered with or not. To validate the signature, you must provide an external tool for your users. See [the official AppImage documentation](#) for additional information.

[Go to TOC](#)

Code Signing macOS Applications

This guide provides information on code signing and notarization for macOS applications.

If you are not utilizing GitHub Actions to perform builds of OSX DMGs, you will need to ensure the environment variable `CI=true` exists. For more information refer to [tauri-apps/tauri#592](#).

Requirements

- macOS 10.13.6 or later
- Xcode 10 or later
- An Apple Developer account enrolled in the [Apple Developer Program](#)

For more details please read the developer article on [notarizing macOS software before distribution](#).

tl;dr

The Tauri code signing and notarization process is configured through the following environment variables:

- `APPLE_SIGNING_IDENTITY` : the name of the keychain entry that contains the signing certificate.
- `APPLE_CERTIFICATE` : base64 string of the `.p12` certificate, exported from the keychain. Useful if you don't have the certificate on the keychain (e.g., CI machines).
- `APPLE_CERTIFICATE_PASSWORD` : the password for the `.p12` certificate.
- `APPLE_ID` and `APPLE_PASSWORD` : your Apple account email and an [app-specific password](#). Only required to notarize the app.
- `APPLE_API_ISSUER` and `APPLE_API_KEY` : authentication with an App Store Connect API key instead of the Apple ID. Only required if you notarize the app.
- `APPLE_PROVIDER_SHORT_NAME` : Team provider short name. If your Apple ID is connected to multiple teams, you have to specify the provider short name of the team you want to use to notarize your app. You can list your account providers using `xcrun altool --list-providers -u AC_USERNAME -p AC_PASSWORD` as explained in the notarization [workflow](#).

Signing Tauri apps

The first step to sign a macOS application is getting a signing certificate from the Apple Developer Program.

Creating a signing certificate

To create a new signing certificate, you must generate a Certificate Signing Request (CSR) file from your Mac computer. [Create a certificate signing request](#) describes creating a CSR.

On your Apple Developer account, navigate to the [Certificates, IDs & Profiles page](#) and click on the `Add` button to open the interface to create a new certificate. Choose the appropriate certificate type (`Apple Distribution` to submit apps to the App Store, and `Developer ID Application` to ship apps outside the App Store). Upload your CSR, and the certificate will be created.

Only the Apple Developer Account Holder can create *Developer ID Application* certificates. But it can be associated with a different Apple ID by creating a CSR with a different user email address.

Downloading a certificate

On [Certificates, IDs & Profiles page](#), click on the certificate you want to use and click on the `Download` button. It saves a `.cer` file that installs the certificate on the keychain once opened. The name of the keychain entry represents the `signing identity`, which can also be found by executing `security find-identity -v -p codesigning`.

A signing certificate is only valid if associated with your Apple ID. An invalid certificate won't be listed on the *Keychain Access > My Certificates* tab or the `security find-identity -v -p codesigning` output.

Signing the Tauri application

The signing configuration is provided to the Tauri bundler via environment variables. You need to configure the certificate to use and an optional authentication configuration to notarize the application.

Certificate environment variables

- `APPLE_SIGNING_IDENTITY` : this is the `signing identity` we highlighted above. It must be defined to sign apps both locally and on CI machines.

Additionally, to simplify the code signing process on CI, Tauri can install the certificate on the keychain for you if you define the `APPLE_CERTIFICATE` and `APPLE_CERTIFICATE_PASSWORD` environment variables.

1. Open the `Keychain Access` app and find your certificate's keychain entry.
2. Expand the entry, double click on the key item, and select `Export $KEYNAME`.
3. Select the path to save the `.p12` file and define the exported certificate password.
4. Convert the `.p12` file to base64 running the following script on the terminal: `openssl base64 -in /path/to/certificate.p12 -out certificate-base64.txt`.
5. Set the contents of the `certificate-base64.txt` file to the `APPLE_CERTIFICATE` environment variable.
6. Set the certificate password to the `APPLE_CERTIFICATE_PASSWORD` environment variable.

Authentication environment variables

These variables are only required to notarize the application.

Notarization is required when using a *Developer ID Application* certificate.

- `APPLE_ID` and `APPLE_PASSWORD` : to authenticate with your Apple ID, set the `APPLE_ID` to your Apple account email (example: `export APPLE_ID=tauri@icloud.com`) and the `APPLE_PASSWORD` to an [app-specific password](#) for the Apple account.
- `APPLE_API_ISSUER` and `APPLE_API_KEY` : alternatively, you can authenticate using an App Store Connect API key. Open the App Store Connect's [Users and Access page](#), select the `Keys` tab, click on the `Add` button and select a name and the `Developer` access. The `APPLE_API_ISSUER` (`Issuer ID`) is presented above the keys table, and the `APPLE_API_KEY` is the value on the `Key ID` column on that table. You also need to download the private key, which can only be done once and is only visible after a page reload (the button is shown on the table row for the newly created key). The private key file must be saved on `./private_keys`, `~/private_keys`, `~/.private_keys` or `~/.appstoreconnect/private_keys`, as stated on the `xcrun altool --help` command.

Building the application

The Tauri bundler automatically signs and notarizes your application with all these environment variables set when running the `tauri build` command.

Example

The following example uses GitHub Actions to sign an application using the [Tauri action](#).

We first define the environment variables we listed above as Secrets on GitHub.

You can view [this guide](#) to learn about GitHub secrets.

Once we have established the GitHub Secrets, we create a GitHub publish workflow in `.github/workflows/main.yml`:

```
name: 'publish'
on:
  push:
    branches:
      - release

jobs:
  publish-tauri:
    strategy:
      fail-fast: false
    matrix:
      platform: [macos-latest]

    runs-on: ${{ matrix.platform }}
    steps:
      - uses: actions/checkout@v2
      - name: setup node
        uses: actions/setup-node@v2
        with:
```

```
node-version: 12
- name: install Rust stable
  uses: actions-rs/toolchain@v1
  with:
    toolchain: stable
- name: install app dependencies and build it
  run: yarn && yarn build
- uses: tauri-apps/tauri-action@v0
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    ENABLE_CODE_SIGNING: ${{ secrets.APPLE_CERTIFICATE }}
    APPLE_CERTIFICATE: ${{ secrets.APPLE_CERTIFICATE }}
    APPLE_CERTIFICATE_PASSWORD: ${{ secrets.APPLE_CERTIFICATE_PASSWORD }}
    APPLE_SIGNING_IDENTITY: ${{ secrets.APPLE_SIGNING_IDENTITY }}
    APPLE_ID: ${{ secrets.APPLE_ID }}
    APPLE_PASSWORD: ${{ secrets.APPLE_PASSWORD }}
  with:
    tagName: app-v__VERSION__ # the action automatically replaces
    \_\_VERSION\_\_ with the app version
    releaseName: 'App v__VERSION__'
    releaseBody: 'See the assets to download this version and install.'
    releaseDraft: true
    prerelease: false
```

The workflow pulls the secrets from GitHub and defines them as environment variables before building the application using the Tauri action. The output is a GitHub release with the signed and notarized macOS application.

Windows - Code signing guide locally & with GitHub Actions

Intro

Code signing your application lets users know that they downloaded the official executable of your app and not some 3rd party malware that poses as your app. While it is not required, it improves users' confidence in your app.

Prerequisites

- Windows - you can likely use other platforms, but this tutorial uses Powershell native features.
- A working Tauri application
- Code signing certificate - you can acquire one of these on services listed in [Microsoft's docs](#). There are likely additional authorities for non-EV certificates than included in that list, please compare them yourself and choose one at your own risk.
 - Please make sure to get a **code signing** certificate, SSL certificates do not work!

This guide assumes that you have a standard code signing certificate> If you have an EV certificate, which generally involves a hardware token, please follow your issuer's documentation instead.

If you sign the app with an EV Certificate, it'll receive an immediate reputation with Microsoft SmartScreen and won't show any warnings to users.

If you opt for an OV Certificate, which is generally cheaper and available to individuals, Microsoft SmartScreen will still show a warning to users when they download the app. It might take some time until your certificate builds enough reputation. You may opt for [submitting your app](#) to Microsoft for manual review. Although not guaranteed, if the app does not contain any malicious code, Microsoft may grant additional reputation and potentially remove the warning for that specific uploaded file.

Getting Started

There are a few things we have to do to get Windows prepared for code signing. This includes converting our certificate to a specific format, installing this certificate, and decoding the required information from the certificate.

A. Convert your `.cer` to `.pfx`

1. You will need the following:

- certificate file (mine is `cert.cer`)
- private key file (mine is `private-key.key`)

2. Open up a command prompt and change to your current directory using `cd Documents/Certs`
3. Convert your `.cer` to a `.pfx` using `openssl pkcs12 -export -in cert.cer -inkey private-key.key -out certificate.pfx`
4. You should be prompted to enter an export password **DON'T FORGET IT!**

B. Import your `.pfx` file into the keystore.

We now need to import our `.pfx` file.

1. Assign your export password to a variable using `$WINDOWS_PFX_PASSWORD = 'MYPASSWORD'`
2. Now Import the certificate using `Import-PfxCertificate -FilePath Certs/certificate.pfx -CertStoreLocation Cert:\CurrentUser\My -Password (ConvertTo-SecureString -String $env:WINDOWS_PFX_PASSWORD -Force -AsPlainText)`

C. Prepare Variables

1. We need the SHA-1 thumbprint of the certificate; you can get this using `openssl pkcs12 -info -in certificate.pfx` and look under for following

Bag Attributes

```
localKeyID: A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A0 B0
```

2. You will capture the `localKeyID` but with no spaces, in this example, it would be `A1B1A2B2A3B3A4B4A5B5A6B6A7B7A8B8A9B9A0B0`. This is our `certificateThumbprint`.
3. We need the SHA digest algorithm used for your certificate (Hint: this is likely `sha256`)
4. We also need a timestamp URL; this is a time server used to verify the time of the certificate signing. I'm using `http://timestamp.comodoca.com`, but whoever you got your certificate from likely has one as well.

Prepare `tauri.conf.json` file

1. Now that we have our `certificateThumbprint`, `digestAlgorithm`, & `timestampUrl` we will open up the `tauri.conf.json`.
2. In the `tauri.conf.json` you will look for the `tauri -> bundle -> windows` section. You see, there are three variables for the information we have captured. Fill it out like below.

```
windows: {
  certificateThumbprint: A1B1A2B2A3B3A4B4A5B5A6B6A7B7A8B8A9B9A0B0,
  digestAlgorithm: sha256,
  timestampUrl: http://timestamp.comodoca.com
}
```

3. Save and run `yarn | yarn build`
4. In the console output, you should see the following output.

```
info: signing app
info: running signtool C:\\Program Files (x86)\\Windows
Kits\\10\\bin\\10.0.19041.0\\x64\\signtool.exe
info: Done Adding Additional Store\\r\\nSuccessfully signed: APPLICATION FILE PATH
HERE
```

Which shows you have successfully signed the `.exe`.

And that's it! You have successfully signed your `.exe` file.

BONUS: Sign your application with GitHub Actions.

We can also create a workflow to sign the application with GitHub actions.

GitHub Secrets

We need to add a few GitHub secrets for the proper configuration of the GitHub Action. These can be named however you would like.

- You can view the [encrypted secrets](#) guide on how to add GitHub secrets.

The secrets we used are as follows

GitHub Secrets	Value for Variable
WINDOWS_CERTIFICATE	Base64 encoded version of your <code>.pfx</code> certificate, can be done using this command <code>certutil -encode certificate.pfx base64cert.txt</code>
WINDOWS_CERTIFICATE_PASSWORD	Certificate export password used on creation of certificate <code>.pfx</code>

Workflow Modifications

1. We need to add a step in the workflow to import the certificate into the Windows environment. This workflow accomplishes the following
 1. Assign GitHub secrets to environment variables
 2. Create a new `certificate` directory
 3. Import `WINDOWS_CERTIFICATE` into `tempCert.txt`
 4. Use `certutil` to decode the `tempCert.txt` from `base64` into a `.pfx` file.
 5. Remove `tempCert.txt`
 6. Import the `.pfx` file into the Cert store of Windows & convert the `WINDOWS_CERTIFICATE_PASSWORD` to a secure string to be used in the import command.
2. We will be using the [tauri-action publish template](#).

```
name: 'publish'
on:
  push:
    branches:
      - release
```

```

jobs:
  publish-tauri:
    strategy:
      fail-fast: false
    matrix:
      platform: [macos-latest, ubuntu-latest, windows-latest]

    runs-on: ${{ matrix.platform }}
    steps:
      - uses: actions/checkout@v2
      - name: setup node
        uses: actions/setup-node@v1
        with:
          node-version: 12
      - name: install Rust stable
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
      - name: install webkit2gtk (ubuntu only)
        if: matrix.platform == 'ubuntu-latest'
        run: |
          sudo apt-get update
          sudo apt-get install -y webkit2gtk-4.0
      - name: install app dependencies and build it
        run: yarn && yarn build
      - uses: tauri-apps/tauri-action@v0
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tagName: app-v__VERSION__ # the action automatically replaces
          \__VERSION\_\_ with the app version
          releaseName: 'App v__VERSION__'
          releaseBody: 'See the assets to download this version and install.'
          releaseDraft: true
          prerelease: false

```

3. Right above `-name: install app dependencies and build it` you will want to add the following step

```

- name: import windows certificate
  if: matrix.platform == 'windows-latest'
  env:
    WINDOWS_CERTIFICATE: ${{ secrets.WINDOWS_CERTIFICATE }}
    WINDOWS_CERTIFICATE_PASSWORD: ${{ secrets.WINDOWS_CERTIFICATE_PASSWORD }}
  run: |
    New-Item -ItemType directory -Path certificate
    Set-Content -Path certificate/tempCert.txt -Value $env:WINDOWS_CERTIFICATE
    certutil -decode certificate/tempCert.txt certificate/certificate.pfx
    Remove-Item -path certificate -include tempCert.txt
    Import-PfxCertificate -FilePath certificate/certificate.pfx -CertStoreLocation
    Cert:\CurrentUser\My -Password (ConvertTo-SecureString -String
    $env:WINDOWS_CERTIFICATE_PASSWORD -Force -AsPlainText)

```

4. Save and push to your repo.

5. Your workflow can now import your windows certificate and import it into the GitHub runner, allowing for automated code-signing!

[Go to TOC](#)

Updater

Configuration

Once you have your Tauri project ready, you need to configure the updater.

Add this in tauri.conf.json

```
updater: {  
    active: true,  
    endpoints: [  
        https://releases.myapp.com/{{target}}/{{current_version}}  
    ],  
    dialog: true,  
    pubkey: YOUR_UPDATER_SIGNATURE_PUBKEY_HERE  
}
```

The required keys are active, endpoints and pubkey; others are optional.

active must be a boolean. By default, it's set to false.

endpoints must be an array. The string `{{target}}` and `{{current_version}}` are automatically replaced in the URL allowing you to determine `server-side` if an update is available. If multiple endpoints are specified, the updater will fallback if a server is not responding within the pre-defined timeout.

dialog if present must be a boolean. By default, it's set to true. If enabled, `events` are turned off as the updater handles everything. If you need the custom events, you MUST turn off the built-in dialog.

pubkey must be a valid public-key generated with Tauri CLI. See [Signing updates](#).

Update Requests

Tauri is indifferent to the request the client application provides for update checking.

`Accept: application/json` is added to the request headers because Tauri is responsible for parsing the response.

For the requirements imposed on the responses and the body format of an update, response see [Server Support](#).

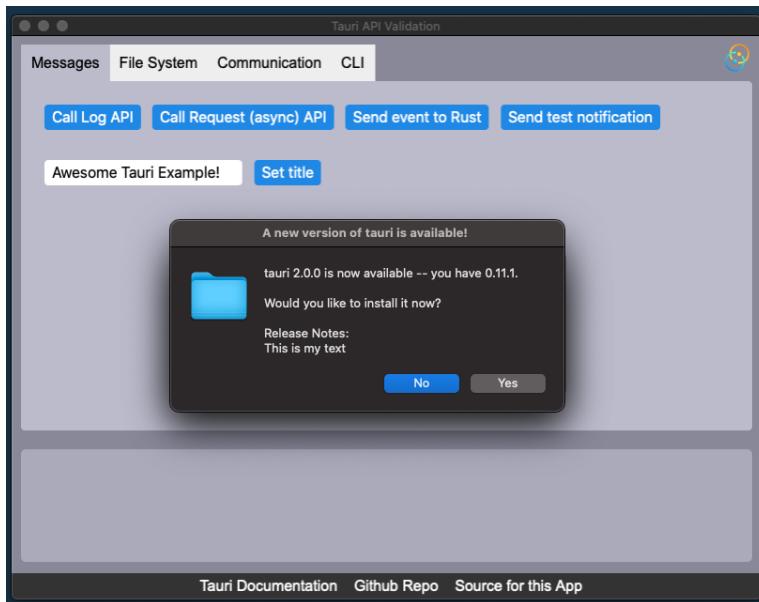
Your update request must *at least* include a version identifier so that the server can determine whether an update for this specific version is required.

It may also include other identifying criteria, such as operating system version, to allow the server to deliver as fine-grained an update as you would like.

How you include the version identifier, or other criteria is specific to the server from which you request updates. A common approach is to use query parameters, [Configuration](#) shows an example.

Built-in dialog

By default, the updater uses a built-in dialog API from Tauri.



New Update

The dialog release notes are represented by the update `note` provided by the `server`. If the user accepts, the update is downloaded and installed. Afterward, the user is prompted to restart the application.

Javascript API

caution You need to `_disable built-in dialog_` in your [tauri configuration](#configuration); Otherwise, the javascript API will NOT work.

```
import { checkUpdate, installUpdate } from '@tauri-apps/api/updater'
import { relaunch } from '@tauri-apps/api/process'
try {
  const { shouldUpdate, manifest } = await checkUpdate()
  if (shouldUpdate) {
    // display dialog
    await installUpdate()
    // install complete, restart the app
    await relaunch()
  }
} catch (error) {
  console.log(error)
}
```

Events

caution

You need to *disable the built-in dialog* in your [tauri configuration](#); Otherwise, events aren't emitted.

To know when an update is ready to be installed, you can subscribe to these events:

Initialize updater and check if a new version is available

If a new version is available, the event `tauri://update-available` is emitted.

Event: `tauri://update`

Rust

```
window.emit(tauri://update.to_string(), None);
```

Javascript

```
import { emit } from '@tauri-apps/api/event'
emit('tauri://update')
```

Listen to the New Update Available event

Event: `tauri://update-available`

Emitted data:

<code>version</code>	Version announced by the server
<code>date</code>	Date announced by the server
<code>body</code>	Note announced by the server

Rust

```
window.listen(tauri://update-available.to_string(), move |msg| {
    println!(New version available: {:?}", msg);
})
```

Javascript

```
import { listen } from '@tauri-apps/api/event'
listen('tauri://update-available', function (res) {
    console.log('New version available: ', res)
})
```

Emit Install and Download events

You need to emit this event to initialize the download and listen to the [install progress](#).

Event: `tauri://update-install`

Rust

```
window.emit(tauri://update-install.to_string(), None);
```

Javascript

```
import { emit } from '@tauri-apps/api/event'
emit('tauri://update-install')
```

Listen to the Install Progress

Event: `tauri://update-status`

Emitted data:

```
status      [ERROR/PENDING/DONE]
error      String/null
```

PENDING is emitted when the download is started and DONE when the installation is complete. You can then ask to restart the application.

ERROR is emitted when there is an error with the updater. We suggest listening to this event even if the dialog is enabled.

Rust

```
window.listen(tauri://update-status.to_string(), move |msg| {
    println!(New status: {:#?}, msg);
})
```

Javascript

```
import { listen } from '@tauri-apps/api/event'
listen('tauri://update-status', function (res) {
    console.log('New status: ', res)
})
```

Server Support

Your server should determine whether an update is required based on the [Update Request](#) your client issues.

If an update is required, your server should respond with a status code of [200 OK](#) and include the [update JSON](#) in the body.

If no update is required your server must respond with a status code of [204 No Content](#).

Update Server JSON Format

When an update is available, Tauri expects the following schema in response to the update request provided:

```
{
    url: https://mycompany.example.com/myapp/releases/myrelease.tar.gz,
    version: 0.0.1,
    notes: These are some release notes,
```

```

pub_date: 2020-09-18T12:29:53+01:00,
signature:
}

```

The required keys are url, version and signature; the others are optional.

pub_date if present must be formatted according to [RFC 3339](#).

signature is the content of the `.sig` file which was generated by Tauri's CLI. See [Signing Updates](#) for instructions on how to set up the required keys.

Update File JSON Format

The alternate update technique uses a plain JSON file, storing your update metadata on S3, gist, or another static file store. Tauri checks against the version field, and if the version of the running process is smaller than the reported one of the JSON and the platform is available, it triggers an update. The format of this file is detailed below:

```

{
  version: v1.0.0,
  notes: Test version,
  pub_date: 2020-06-22T19:25:57Z,
  platforms: {
    darwin-x86_64: {
      signature: ,
      url: https://github.com/lemarier/tauri-
test/releases/download/v1.0.0/app.app.tar.gz
    },
    darwin-aarch64: {
      signature: ,
      url: https://github.com/lemarier/tauri-
test/releases/download/v1.0.0/silicon/app.app.tar.gz
    },
    linux-x86_64: {
      signature: ,
      url: https://github.com/lemarier/tauri-
test/releases/download/v1.0.0/app.AppImage.tar.gz
    },
    windows-x86_64: {
      signature: ,
      url: https://github.com/lemarier/tauri-
test/releases/download/v1.0.0/app.x64.msi.zip
    }
  }
}

```

Note that each platform key is in the `OS-ARCH` format, where `OS` is one of `linux`, `darwin` or `windows`, and `ARCH` is one of `x86_64`, `aarch64`, `i686` or `armv7`.

Bundler (Artifacts)

The Tauri bundler automatically generates update artifacts if the updater is enabled in `tauri.conf.json`. Your update artifacts are automatically signed if the bundler can locate your private and public keys.

The signature is the content of the generated `.sig` file. The signature can be uploaded to GitHub safely or made public if your private key is secure.

You can see how it's [bundled with the CI](#) and a [sample tauri.conf.json](#).

macOS

On macOS, we create a `.tar.gz` from the whole application. (`.app`)

```
target/release/bundle
└── macos
    └── app.app
        └── app.app.tar.gz (update bundle)
            └── app.app.tar.gz.sig
```

Windows

On Windows, we create a `.zip` from the MSI; when downloaded and validated, we run the MSI install.

```
target/release/bundle
└── msi
    └── app.x64.msi
        └── app.x64.msi.zip (update bundle)
            └── app.x64.msi.zip.sig
```

Linux

On Linux, we create a `.tar.gz` from the AppImage.

```
target/release/bundle
└── appimage
    └── app.AppImage
        └── app.AppImage.tar.gz (update bundle)
            └── app.AppImage.tar.gz.sig
```

Signing updates

We offer a built-in signature to ensure your update is safe to be installed.

To sign your updates, you need two things.

The *Public-key* (pubkey) should be added inside your `tauri.conf.json` to validate the update archive before installing.

The *Private key* (privkey) is used to sign your update and should NEVER be shared with anyone. Also, if you lost this key, you'll NOT be able to publish a new update to the current user base. It's crucial to save it in a safe place, and you can always access it.

To generate your keys, you need to use the Tauri CLI:

```
tauri signer generate -w ~/.tauri/myapp.key
```

You have multiple options available

Generate keypair to sign files

USAGE:

```
tauri signer generate [OPTIONS]
```

OPTIONS:

-f, --force	Overwrite private key even if it exists on the specified path
-h, --help	Print help information
-p, --password <PASSWORD>	Set private key password when signing
-V, --version	Print version information
-w, --write-keys <WRITE_KEYS>	Write private key to a file

Environment variables used to sign with the Tauri [bundler](#):

If they are set, the bundler automatically generates and signs the updater artifacts.

`TAURI_PRIVATE_KEY` Path or String of your private key

`TAURI_KEY_PASSWORD` Your private key password (optional)

How can I use unpublished Tauri changes?

To use Tauri from GitHub (bleeding edge version) you need to change your `Cargo.toml` file and update your CLI and API.

Pulling the Rust crate from source Using the Tauri CLI from source Using the Tauri API from source

Should I use Node or Cargo?

Even though installing the CLI through Cargo is the preferred option, it has to compile the whole binary from scratch when you install it. If you're in a CI environment or on a very slow machine you're better off choosing another installation method.

As the CLI is written in Rust, it is naturally available through [crates.io](#) and installable with Cargo.

We also compile the CLI as a native Node.js addon and distribute it [via npm](#). This has several advantages compared to the Cargo installation method:

1. The CLI is pre-compiled, leading to much faster install times
2. You can pin a specific version in your package.json file
3. If you develop custom tooling around Tauri, you can import the CLI as a regular JavaScript module
4. You can install the CLI using a JavaScript manager:

Recommended Browserlist

We recommend using `es2021`, `last 3 Chrome versions`, and `safari13` for your browserlist and build targets. Tauri leverages the OS's native rendering engine (WebKit on macOS, WebView2 on Windows and WebKitGTK on Linux).

Build Conflict with Homebrew on Linux

Homebrew on Linux includes its own `pkg-config` (a utility to find libraries on the system). This can cause conflicts when installing the same `pkg-config` package for Tauri (usually installed through the package manager like `apt`). When you try to build a Tauri app it will try to invoke `pkg-config` and will end up invoking the one from Homebrew. If Homebrew wasn't used to install Tauri's dependencies, this can cause errors.

Errors will *usually* contain messages along the lines of `error: failed to run custom build command for X - Package Y was not found in the pkg-config search path..`. Note that you may see similar errors if the required dependencies are not installed at all.

There are two solutions to this issue:

1. [Uninstall Homebrew](#)
2. Set the `PKG_CONFIG_PATH` environment variable to point to the correct `pkg-config` before building a Tauri app


```
import DocCardList from '@theme/DocCardList'; import {useCurrentSidebarCategory} from '@docusaurus/theme-common';
```

Features

[Go to TOC](#)

Making Your Own CLI

Tauri enables your app to have a CLI through [clap](#), a robust command line argument parser. With a simple CLI definition in your `tauri.conf.json` file, you can define your interface and read its argument matches map on JavaScript and/or Rust.

Base Configuration

Under `tauri.conf.json`, you have the following structure to configure the interface:

`src-tauri/tauri.conf.json`

```
{
  tauri: {
    cli: {
      description: , // command description that's shown on help
      longDescription: , // command long description that's shown on help
      beforeHelp: , // content to show before the help text
      afterHelp: , // content to show after the help text
      args: [], // list of arguments of the command, we'll explain it later
      subcommands: {
        subcommand-name: {
          // configures a subcommand that is accessible
          // with `./app subcommand-name --arg1 --arg2 --etc`
          // configuration as above, with description, args, etc.
        }
      }
    }
  }
}
```

All JSON configurations here are just samples, many other fields have been omitted for the sake of clarity.

Adding Arguments

The `args` array represents the list of arguments accepted by its command or subcommand. You can find more details about the way to configure them [here](#).

Positional Arguments

A positional argument is identified by its position in the list of arguments. With the following configuration:

```
{
  args: [
    {
      name: source,
      index: 1,
      takesValue: true
    }
  ]
}
```

```

    },
    {
      name: destination,
      index: 2,
      takesValue: true
    }
  ]
}

```

Users can run your app as `./app tauri.txt dest.txt` and the arg matches map will define `source` as `tauri.txt` and `destination` as `dest.txt`.

Named Arguments

A named argument is a (key, value) pair where the key identifies the value. With the following configuration:

```

{
  args: [
    {
      name: type,
      short: t,
      takesValue: true,
      multiple: true,
      possibleValues: [foo, bar]
    }
  ]
}

```

Users can run your app as `./app --type foo bar`, `./app -t foo -t bar` or `./app --type=foo,bar` and the arg matches map will define `type` as `[foo, bar]`.

Flag Arguments

A flag argument is a standalone key whose presence or absence provides information to your application. With the following configuration:

```

{
  args: [
    {
      name: verbose,
      short: v,
      multipleOccurrences: true
    }
  ]
}

```

Users can run your app as `./app -v -v -v`, `./app --verbose --verbose --verbose` or `./app -vvv` and the arg matches map will define `verbose` as `true`, with `occurrences = 3`.

Subcommands

Some CLI applications has additional interfaces as subcommands. For instance, the `git` CLI has `git branch`, `git commit` and `git push`. You can define additional nested interfaces with the `subcommands` array:

```
{
  cli: {
    ...
    subcommands: {
      branch: {
        args: []
      },
      push: {
        args: []
      }
    }
  }
}
```

Its configuration is the same as the root application configuration, with the `description`, `longDescription`, `args`, etc.

Reading the matches

Rust

```
fn main() {
    tauri::Builder::default()
        .setup(|app| {
            match app.get_cli_matches() {
                // `matches` here is a Struct with { args, subcommand }.
                // `args` is `HashMap<String, ArgData>` where `ArgData` is a struct with { value, occurrences }.
                // `subcommand` is `Option<Box<SubcommandMatches>>` where `SubcommandMatches` is a struct with { name, matches }.
                Ok(matches) => {
                    println!("{:?}", matches)
                }
                Err(_) => {}
            }
            Ok(())
        })
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}
```

JavaScript

```
import { getMatches } from '@tauri-apps/api/cli'

getMatches().then((matches) => {
    // do something with the { args, subcommand } matches
})
```

Complete documentation

You can find more about the CLI configuration [here](#).

[Go to TOC](#)

Calling Rust from the frontend

Tauri provides a simple yet powerful `command` system for calling Rust functions from your web app. Commands can accept arguments and return values. They can also return errors and be `async`.

Basic Example

Commands are defined in your `src-tauri/src/main.rs` file. To create a command, just add a function and annotate it with `#[tauri::command]`:

```
#[tauri::command]
fn my_custom_command() {
    println!(`I was invoked from JS!`);
```

You will have to provide a list of your commands to the builder function like so:

```
// Also in main.rs
fn main() {
    tauri::Builder::default()
        // This is where you pass in your commands
        .invoke_handler(tauri::generate_handler![my_custom_command])
        .run(tauri::generate_context!())
        .expect("failed to run app");
}
```

Now, you can invoke the command from your JS code:

```
// When using the Tauri API npm package:
import { invoke } from '@tauri-apps/api/tauri'
// When using the Tauri global script (if not using the npm package)
// Be sure to set `build.withGlobalTauri` in `tauri.conf.json` to true
const invoke = window.__TAURI__.invoke

// Invoke the command
invoke('my_custom_command')
```

Passing Arguments

Your command handlers can take arguments:

```
#[tauri::command]
fn my_custom_command(invocation_message: String) {
    println!(`I was invoked from JS, with this message: {}, invocation_message`);
```

Arguments should be passed as a JSON object with camelCase keys:

```
invoke('my_custom_command', { invocationMessage: 'Hello!' })
```

Arguments can be of any type, as long as they implement `serde::Deserialize`.

Returning Data

Command handlers can return data as well:

```
#[tauri::command]
fn my_custom_command() -> String {
    Hello from Rust!.into()
}
```

The `invoke` function returns a promise that resolves with the returned value:

```
invoke('my_custom_command').then((message) => console.log(message))
```

Returned data can be of any type, as long as it implements `serde::Serialize`.

Error Handling

If your handler could fail and needs to be able to return an error, have the function return a `Result`:

```
#[tauri::command]
fn my_custom_command() -> Result<String, String> {
    // If something fails
    Err(This failed!.into())
    // If it worked
    Ok(This worked!.into())
}
```

If the command returns an error, the promise will reject, otherwise, it resolves:

```
invoke('my_custom_command')
    .then((message) => console.log(message))
    .catch((error) => console.error(error))
```

Async Commands

Async commands are executed on a separate thread using `async_runtime::spawn`. Commands without the `async` keyword are executed on the main thread unless defined with `# [tauri::command(async)]`.

If your command needs to run asynchronously, simply declare it as `async`:

```
#[tauri::command]
async fn my_custom_command() {
    // Call another async function and wait for it to finish
    let result = some_async_function().await;
    println!(Result: {}, result);
}
```

Since invoking the command from JS already returns a promise, it works just like any other command:

```
invoke('my_custom_command').then(() => console.log('Completed!'))
```

Accessing the Window in Commands

Commands can access the `Window` instance that invoked the message:

```
#[tauri::command]
async fn my_custom_command(window: tauri::Window) {
    println!(Window: {}, window.label());
}
```

Accessing an AppHandle in Commands

Commands can access an `AppHandle` instance:

```
#[tauri::command]
async fn my_custom_command(app_handle: tauri::AppHandle) {
    let app_dir = app_handle.path_resolver().app_dir();
    use tauri::GlobalShortcutManager;
    app_handle.global_shortcut_manager().register(CTRL + U, move || {});
```

Accessing managed state

Tauri can manage state using the `manage` function on `tauri::Builder`. The state can be accessed on a command using `tauri::State`:

```
struct MyState(String);

#[tauri::command]
fn my_custom_command(state: tauri::State<MyState>) {
    assert_eq!(state.0 == some state value, true);
}

fn main() {
    tauri::Builder::default()
        .manage(MyState(some state value.into()))
        .invoke_handler(tauri::generate_handler![my_custom_command])
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}
```

Creating Multiple Commands

The `tauri::generate_handler!` macro takes an array of commands. To register multiple commands, you cannot call `invoke_handler` multiple times. Only the last call will be used. You must pass each command to a single call of `tauri::generate_handler!`.

```
#[tauri::command]
fn cmd_a() -> String {
    Command a
}
#[tauri::command]
fn cmd_b() -> String {
```

```
Command b
}

fn main() {
    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![cmd_a, cmd_b])
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}
```

Complete Example

Any or all of the above features can be combined:

```
struct Database;

#[derive(serde::Serialize)]
struct CustomResponse {
    message: String,
    other_val: usize,
}

async fn some_other_function() -> Option<String> {
    Some(response.into())
}

#[tauri::command]
async fn my_custom_command(
    window: tauri::Window,
    number: usize,
    database: tauri::State<'_, Database>,
) -> Result<CustomResponse, String> {
    println!(Called from {}, window.label());
    let result: Option<String> = some_other_function().await;
    if let Some(message) = result {
        Ok(CustomResponse {
            message,
            other_val: 42 + number,
        })
    } else {
        Err("No result".into())
    }
}

fn main() {
    tauri::Builder::default()
        .manage(Database {})
        .invoke_handler(tauri::generate_handler![my_custom_command])
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}
```

```
// Invocation from JS

invoke('my_custom_command', {
    number: 42,
})
    .then((res) =>
```

```
    console.log(`Message: ${res.message}, Other Val: ${res.other_val}`)  
  )  
  .catch((e) => console.error(e))
```

Events

The Tauri event system is a multi-producer multi-consumer communication primitive that allows message passing between the frontend and the backend. It is analogous to the command system, but a payload type check must be written on the event handler and it simplifies communication from the backend to the front-end, working like a channel.

A Tauri application can listen and emit global and window-specific events. Usage from the frontend and the backend is described below.

Frontend

The event system is accessible on the frontend on the `event` and `window` modules of the `@tauri-apps/api` package.

Global events

To use the global event channel, import the `event` module and use the `emit` and `listen` functions:

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

// listen to the `click` event and get a function to remove the event listener
// there's also a `once` function that subscribes to an event and automatically
// unsubscribes the listener on the first event
const unlisten = await listen('click', (event) => {
    // event.event is the event name (useful if you want to use a single callback fn
    // for multiple event types)
    // event.payload is the payload object
})

// emits the `click` event with the object payload
emit('click', {
    theMessage: 'Tauri is awesome!',
})
```

Window-specific events

Window-specific events are exposed on the `window` module.

```
import { appWindow, WebviewWindow } from '@tauri-apps/api/window'

// emit an event that are only visible to the current window
appWindow.emit('event', { message: 'Tauri is awesome!' })

// create a new webview window and emit an event only to that window
const webview = new WebviewWindow('window')
webview.emit('event')
```

Backend

On the backend, the global event channel is exposed on the `App` struct, and window-specific events can be emitted using the `Window` trait.

Global events

```
use tauri::Manager;

// the payload type must implement `Serialize` and `Clone`.
#[derive(Clone, serde::Serialize)]
struct Payload {
    message: String,
}

fn main() {
    tauri::Builder::default()
        .setup(|app| {
            // listen to the `event-name` (emitted on any window)
            let id = app.listen_global(event_name, |event| {
                println!(got event-name with payload {:?}, event.payload());
            });
            // unlisten to the event using the `id` returned on the `listen_global` function
            // an `once_global` API is also exposed on the `App` struct
            app.unlisten(id);

            // emit the `event-name` event to all webview windows on the frontend
            app.emit_all(event_name, Payload { message: Tauri is awesome!.into()})
        }).unwrap();
        Ok(())
    }
    .run(tauri::generate_context!())
    .expect("failed to run app");
}
```

Window-specific events

To use the window-specific event channel, a `Window` object can be obtained on a command handler or with the `get_window` function:

```
use tauri::{Manager, Window};

// the payload type must implement `Serialize` and `Clone`.
#[derive(Clone, serde::Serialize)]
struct Payload {
    message: String,
}

// init a background process on the command, and emit periodic events only to the window that used the command
#[tauri::command]
fn init_process(window: Window) {
    std::thread::spawn(move || {
        loop {
            window.emit(event_name, Payload { message: Tauri is awesome!.into()})
        }.unwrap();
    })
}
```

```
    });

fn main() {
    tauri::Builder::default()
        .setup(|app| {
            // `main` here is the window label; it is defined on the window creation or
            // under `tauri.conf.json`
            // the default value is `main`. note that it must be unique
            let main_window = app.get_window(main).unwrap();

            // listen to the `event-name` (emitted on the `main` window)
            let id = main_window.listen(event_name, |event| {
                println!(got window event-name with payload {:{}}?, event.payload());
            });
            // unlisten to the event using the `id` returned on the `listen` function
            // an `once` API is also exposed on the `Window` struct
            main_window.unlisten(id);

            // emit the `event-name` event to the `main` window
            main_window.emit(event_name, Payload { message: Tauri is awesome!.into()
            }).unwrap();
            Ok(())
        })
        .invoke_handler(tauri::generate_handler![init_process])
        .run(tauri::generate_context!())
        .expect(failed to run app);
}
```

```
import Command from '@theme/Command'
```

Icons

Tauri ships with a default iconset based on its logo. This is NOT what you want when you ship your application. To remedy this common situation, Tauri provides the `icon` command that will take an input file (`./app-icon.png` by default) and create all the icons needed for the various platforms.

info Note on filetypes

- `icon.icns` = macOS
- `icon.ico` = Windows
- `*.png` = Linux
- `Square*Logo.png` & `StoreLogo.png` = Currently unused but intended for AppX/MS Store targets.

Note that icon types may be used on platforms other than those listed above (especially `png`). Therefore we recommend including all icons even if you intend to only build for a subset of platforms.

Command Usage

Starting with `@tauri-apps/cli` / `tauri-cli` version 1.1 the `icon` subcommand is part of the main cli:

```
> cargo tauri icon --help
cargo-tauri-icon 1.1.0

Generates various icons for all major platforms

USAGE:
  cargo tauri icon [OPTIONS] [INPUT]

ARGS:
  <INPUT>    Path to the source icon (png, 1240x1240px with transparency)
  [default: ./app-icon.png]

OPTIONS:
  -h, --help            Print help information
  -o, --output <OUTPUT>  Output directory. Default: 'icons' directory next to
  the tauri.conf.json file
  -v, --verbose         Enables verbose logging
  -V, --version          Print version information
```

By default, the icons will be placed in your `src-tauri/icons` folder where they will automatically be included in your built app. If you want to source your icons from a different location, you can edit this part of the `tauri.conf.json` file:

```
{
  tauri: {
    bundle: {
      icon: [
        'icons/32x32.png',
        'icons/128x128.png',
        'icons/128x128@2x.png',
        'icons/icon.icns',
```



Creating the icons manually

If you prefer to build these icons yourself (if you want to have a simpler design for small sizes or because you don't want to depend on the CLI's internal image resizing), the required layer sizes and names for the `icns` file are described [in the Tauri repo](#) and the `ico` file must include layers for 16, 24, 32, 48, 64 and 256 pixels. For an optimal display of the ICO image *in development*, the 32px layer should be the first layer.

Window Menu

Native application menus can be attached to a window.

Creating a menu

To create a native window menu, import the `Menu`, `Submenu`, `MenuItem` and `CustomMenuItem` types. The `MenuItem` enum contains a collection of platform-specific items (currently not implemented on Windows). The `CustomMenuItem` allows you to create your own menu items and add special functionality to them.

```
use tauri::{CustomMenuItem, Menu, MenuItem, Submenu};
```

Create a `Menu` instance:

```
// 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 close = CustomMenuItem::new(close.to_string(), Close);
let submenu = Submenu::new(File, Menu::new().add_item(quit).add_item(close));
let menu = Menu::new()
    .add_native_item(MenuItem::Copy)
    .add_item(CustomMenuItem::new(hide, Hide))
    .add_submenu(submenu);
```

Adding the menu to all windows

The defined menu can be set to all windows using the `menu` API on the `tauri::Builder` struct:

```
use tauri::{CustomMenuItem, Menu, MenuItem, Submenu};

fn main() {
    let menu = Menu::new(); // configure the menu
    tauri::Builder::default()
        .menu(menu)
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}
```

Adding the menu to a specific window

You can create a window and set the menu to be used. This allows defining a specific menu set for each application window.

```
use tauri::{CustomMenuItem, Menu, MenuItem, Submenu};
use tauri::WindowBuilder;

fn main() {
    let menu = Menu::new(); // configure the menu
    tauri::Builder::default()
        .setup(|app| {
            WindowBuilder::new(
                app,
                main_window.to_string(),

```

```

        tauri::WindowUrl::App(index.html.into()),
    )
    .menu(menu)
    .build()?;
    Ok(())
})
.run(tauri::generate_context!())
.expect(error while running tauri application);
}

```

Listening to events on custom menu items

Each `CustomMenuItem` triggers an event when clicked. Use the `on_menu_event` API to handle them, either on the global `tauri::Builder` or on a specific window.

Listening to events on global menus

```

use tauri::{CustomMenuItem, Menu, MenuItem};

fn main() {
    let menu = Menu::new(); // configure the menu
    tauri::Builder::default()
        .menu(menu)
        .on_menu_event(|event| {
            match event.menu_item_id() {
                quit => {
                    std::process::exit(0);
                }
                close => {
                    event.window().close().unwrap();
                }
                _ => {}
            }
        })
        .run(tauri::generate_context!())
        .expect(error while running tauri application);
}

```

Listening to events on window menus

```

use tauri::{CustomMenuItem, Menu, MenuItem};
use tauri::{Manager, WindowBuilder};

fn main() {
    let menu = Menu::new(); // configure the menu
    tauri::Builder::default()
        .setup(|app| {
            let window = WindowBuilder::new(
                app,
                main_window.to_string(),
                tauri::WindowUrl::App(index.html.into()),
            )
            .menu(menu)
            .build()?;
            let window_ = window.clone();
            window.on_menu_event(move |event| {
                match event.menu_item_id() {
                    quit => {
                        std::process::exit(0);
                    }

```

```

        }
        close => {
            window_.close().unwrap();
        }
        - => {}
    }
});
Ok(())
)
.run(tauri::generate_context!())
.expect(error while running tauri application);
}

```

Updating menu items

The `Window` struct has a `menu_handle` method, which allows updating menu items:

```

fn main() {
    let menu = Menu::new(); // configure the menu
    tauri::Builder::default()
        .menu(menu)
        .setup(|app| {
            let main_window = app.get_window(main).unwrap();
            let menu_handle = main_window.menu_handle();
            std::thread::spawn(move || {
                // you can also `set_selected`, `set_enabled` and `set_native_image`
                // (macOS only).
                menu_handle.get_item(item_id).set_title(New title);
            });
        })
    Ok(())
}

```

Multiwindow

Manage multiple windows on a single application.

Creating a window

A window can be created statically from the Tauri configuration file or at runtime.

Static window

Multiple windows can be created with the `tauri.windows` configuration array. The following JSON snippet demonstrates how to statically create several windows through the config:

```
{
  "tauri": {
    "windows": [
      {
        "label": "external",
        "title": "Tauri Docs",
        "url": "https://tauri.app"
      },
      {
        "label": "local",
        "title": "Tauri",
        "url": "index.html"
      }
    ]
  }
}
```

Note that the window label must be unique and can be used at runtime to access the window instance.

The complete list of configuration options available for static windows can be found in the [WindowConfig](#) documentation.

Runtime window

You can also create windows at runtime either via the Rust layer or through the Tauri API.

Create a window in Rust

A window can be created at runtime using the [WindowBuilder](#) struct.

To create a window, you must have an instance of the running [App](#) or an [AppHandle](#).

Create a window using the [App](#) instance

The [App](#) instance can be obtained in the setup hook or after a call to [Builder::build](#).

```
tauri::Builder::default()
  .setup(|app| {
    let docs_window = tauri::WindowBuilder::new(
      app,
```

```

    external, /* the unique window label */
    tauri::WindowUrl::External(https://tauri.app/.parse().unwrap())
).build()?;
let local_window = tauri::WindowBuilder::new(
    app,
    local,
    tauri::WindowUrl::App(index.html.into())
).build()?;
Ok(())
)
}

```

Using the setup hook ensures static windows and Tauri plugins are initialized. Alternatively, you can create a window after building the App:

```

let app = tauri::Builder::default()
    .build(tauri::generate_context!())
    .expect(error while building tauri application);

let docs_window = tauri::WindowBuilder::new(
    &app,
    external, /* the unique window label */
    tauri::WindowUrl::External(https://tauri.app/.parse().unwrap())
).build().expect(failed to build window);

let local_window = tauri::WindowBuilder::new(
    &app,
    local,
    tauri::WindowUrl::App(index.html.into())
).build();

```

This method is useful when you cannot move ownership of values to the setup closure.

Create a window using an [AppHandle](#) instance

An [AppHandle](#) instance can be obtained using the `[App::handle]` function or directly injected in Tauri commands.

```

tauri::Builder::default()
    .setup(|app| {
        let handle = app.handle();
        std::thread::spawn(move || {
            let local_window = tauri::WindowBuilder::new(
                &handle,
                local,
                tauri::WindowUrl::App(index.html.into())
            ).build()?;
        });
        Ok(())
    })
}

```

```

#[tauri::command]
async fn open_docs(handle: tauri::AppHandle) {
    let docs_window = tauri::WindowBuilder::new(
        &handle,
        external, /* the unique window label */
        tauri::WindowUrl::External(https://tauri.app/.parse().unwrap())
    ).build().unwrap();
}

```

info

When creating windows in a Tauri command, ensure the command function is `async` to avoid a deadlock on Windows due to the [wry#583](#) issue.

Create a window in JavaScript

Using the Tauri API you can easily create a window at runtime by importing the `WebviewWindow` class.

```
import { WebviewWindow } from '@tauri-apps/api/window'
const webview = new WebviewWindow('theUniqueLabel', {
  url: 'path/to/page.html',
})
// since the webview window is created asynchronously,
// Tauri emits the `tauri://created` and `tauri://error` to notify you of the
creation response
webview.once('tauri://created', function () {
  // webview window successfully created
})
webview.once('tauri://error', function (e) {
  // an error occurred during webview window creation
})
```

Accessing a window at runtime

The window instance can be queried using its label and the `get_window` method on Rust or `WebviewWindow.getByLabel` on JavaScript.

```
use tauri::Manager;
tauri::Builder::default()
  .setup(|app| {
    let main_window = app.get_window(main).unwrap();
    Ok(())
})
```

Note that you must import `tauri::Manager` to use the `get_window` method on `App` or `AppHandle` instances.

```
import { WebviewWindow } from '@tauri-apps/api/window'
const mainWindow = WebviewWindow.getByLabel('main')
```

Communicating with other windows

Window communication can be done using the event system. See the [Event Guide](#) for more information.

[Go to TOC](#)

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 the `tauri-cli` installed just run:

```
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 attached javascript API package like this:

```
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:

```

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 setup and maintain state, just like your app can:

```

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.

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

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)  
    }  
}
```

Splashscreen

If your webpage could take some time to load, or if you need to run an initialization procedure in Rust before displaying your main window, a splashscreen could improve the loading experience for the user.

Setup

First, create a `splashscreen.html` in your `distDir` that contains the HTML code for a splashscreen. Then, update your `tauri.conf.json` like so:

```
windows: [
  {
    title: Tauri App,
    width: 800,
    height: 600,
    resizable: true,
    fullscreen: false,
    + visible: false // Hide the main window by default
  },
  // Add the splashscreen window
  + {
    + width: 400,
    + height: 200,
    + decorations: false,
    + url: splashscreen.html,
    + label: splashscreen
  }
]
```

Now, your main window will be hidden and the splashscreen window will show when your app is launched. Next, you'll need a way to close the splashscreen and show the main window when your app is ready. How you do this depends on what you are waiting for before closing the splashscreen.

Waiting for Webpage

If you are waiting for your web code, you'll want to create a `close_splashscreen` command.

```
use tauri::Manager;
// Create the command:
// This command must be async so that it doesn't run on the main thread.
#[tauri::command]
async fn close_splashscreen(window: tauri::Window) {
  // Close splashscreen
  if let Some(splashscreen) = window.get_window(splashscreen) {
    splashscreen.close().unwrap();
  }
  // Show main window
  window.get_window(main).unwrap().show().unwrap();
}

// Register the command:
fn main() {
  tauri::Builder::default()
  // Add this line
```

```

    .invoke_handler(tauri::generate_handler![close_splashscreen])
    .run(tauri::generate_context!())
    .expect(failed to run app);
}

```

Then, you can call it from your JS:

```

// With the Tauri API npm package:
import { invoke } from '@tauri-apps/api/tauri'
// With the Tauri global script:
const invoke = window.__TAURI__.invoke

document.addEventListener('DOMContentLoaded', () => {
    // This will wait for the window to load, but you could
    // run this function on whatever trigger you want
    invoke('close_splashscreen')
})

```

Waiting for Rust

If you are waiting for Rust code to run, put it in the `setup` function handler so you have access to the `App` instance:

```

use tauri::Manager;
fn main() {
    tauri::Builder::default()
        .setup(|app| {
            let splashscreen_window = app.get_window(splashscreen).unwrap();
            let main_window = app.get_window(main).unwrap();
            // we perform the initialization code on a new task so the app doesn't
            // freeze
            tauri::async_runtime::spawn(async move {
                // initialize your app here instead of sleeping :)
                println!(Initializing...);
                std::thread::sleep(std::time::Duration::from_secs(2));
                println!(Done initializing);

                // After it's done, close the splashscreen and display the main window
                splashscreen_window.close().unwrap();
                main_window.show().unwrap();
            });
            Ok(())
        })
        .run(tauri::generate_context!())
        .expect(failed to run app);
}

```

[Go to TOC](#)

System Tray

Native application system tray.

Setup

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

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

Initialize a new tray instance:

```
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`, `SystemMenuItem` and `CustomMenuItem` types:

```
use tauri::{CustomMenuItem, SystemTrayMenu, SystemMenuItem};
```

Create the `SystemTrayMenu`:

```
// 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:

```
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:

```
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();
```

Keep the app running in the background after closing all windows

By default, tauri closes the application when the last window is closed. If your app should run in the background, you can call `api.prevent_close()` 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();
    }
    - => {}
});
```

Window Customization

Tauri provides lots of options for customizing the look and feel of your app's window. You can create custom titlebars, have transparent windows, enforce size constraints, and more.

Configuration

There are three ways to change the window configuration:

- [Through tauri.conf.json](#)
- [Through the JS API](#)
- [Through the Window in Rust](#)

Creating a Custom Titlebar

A common use of these window features is creating a custom titlebar. This short tutorial will guide you through that process.

CSS

You'll need to add some CSS for the titlebar to keep it at the top of the screen and style the buttons:

```
.titlebar {
    height: 30px;
    background: #329ea3;
    user-select: none;
    display: flex;
    justify-content: flex-end;
    position: fixed;
    top: 0;
    left: 0;
    right: 0;
}
.titlebar-button {
    display: inline-flex;
    justify-content: center;
    align-items: center;
    width: 30px;
    height: 30px;
}
.titlebar-button:hover {
    background: #5bbec3;
}
```

HTML

Now, you'll need to add the HTML for the titlebar. Put this at the top of your `<body>` tag:

```
<div data-tauri-drag-region class=titlebar>
    <div class=titlebar-button id=titlebar-minimize>
        <img
            src=https://api.iconify.design/mdi:window-minimize.svg>
```

```

        alt=minimize
    />
</div>
<div class=titlebar-button id=titlebar-maximize>
    <img
        src=https://api.iconify.design mdi:window-maximize.svg
        alt=maximize
    />
</div>
<div class=titlebar-button id=titlebar-close>
    <img src=https://api.iconify.design mdi:close.svg alt=close />
</div>
</div>

```

Note that you may need to move the rest of your content down so that the titlebar doesn't cover it.

JS

Finally, you'll need to make the buttons work:

```

import { appWindow } from '@tauri-apps/api/window'
document
    .getElementById('titlebar-minimize')
    .addEventListener('click', () => appWindow.minimize())
document
    .getElementById('titlebar-maximize')
    .addEventListener('click', () => appWindow.toggleMaximize())
document
    .getElementById('titlebar-close')
    .addEventListener('click', () => appWindow.close())

```

```
import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';
```

Prerequisites

Installing

The first step is to install [Rust](#) and system dependencies. Keep in mind that this setup is only needed for *developing Tauri apps*. Your end-users are not required to do any of this.

Setting Up Windows

1. Microsoft Visual Studio C++ Build Tools

You will need to install Microsoft Visual Studio C++ build tools. The easiest way is to install [Build Tools for Visual Studio 2022](#). When asked which workloads to install, ensure C++ build tools and the Windows 10 SDK are selected.

Microsoft Visual Studio Installer Microsoft Visual Studio Installer

Listing 1-1: Selecting C++ build tools and Windows 10 SDK using the Visual Studio Build Tools 2022 installer.

2. WebView2

WebView2 is pre-installed in Windows 11

Tauri heavily depends on WebView2 to render web content on Windows, therefore you must have WebView2 installed. The easiest way is to download and run the Evergreen Bootstrapper from [Microsoft's website](#).

The bootstrapper script will try to determine the correct architecture and version for your system. Still, if you run into issues (especially with Windows on ARM) you can select the correct standalone installer.

3. Rust

Lastly, go to <https://www.rust-lang.org/tools/install> to install `rustup` (the Rust installer). Note that you have to restart your terminal, and in some cases, Windows itself, for the changes to take effect.

Alternatively, you could use `winget` to install rustup using the following command in PowerShell:

```
winget install --id Rustlang.Rustup
```

Setting Up macOS

1. CLang and macOS Development Dependencies

You will need to install CLang and macOS development dependencies. To do this, run the following command in your terminal:

```
xcode-select --install
```

2. Rust

To install Rust on macOS, open a terminal and enter the following command:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

We have audited this bash script, and it does what it says it is supposed to do. Nevertheless, before blindly curl-bashing a script, it is always wise to look at it first. Here is the file as a plain script: [rustup.sh](#)

The command downloads a script and starts the installation of the `rustup` tool, which installs the latest stable version of Rust. You might be prompted for your password. If the installation was successful, the following line will appear:

```
Rust is installed now. Great!
```

Make sure to restart your terminal for the changes to take effect.

Setting Up Linux

1. System Dependencies

You will need to install a couple of system dependencies, such as a C compiler and `webkit2gtk`. Below are commands for a few popular distributions:

```
sudo apt update
sudo apt install libwebkit2gtk-4.0-dev \
    build-essential \
    curl \
    wget \
    libssl-dev \
    libgtk-3-dev \
    libayatana-appindicator3-dev \
    librsvg2-dev
```

```
sudo pacman -Syu
sudo pacman -S --needed \
    webkit2gtk \
    base-devel \
    curl \
    wget \
    openssl \
```

```
appmenu-gtk-module \
gtk3 \
libappindicator-gtk3 \
librsvg \
libvips
```

```
sudo dnf check-update
sudo dnf install webkit2gtk3-devel.x86_64 \
openssl-devel \
curl \
wget \
libappindicator-gtk3 \
librsvg2-devel
sudo dnf group install C Development Tools and Libraries
```

```
sudo zypper up
sudo zypper in webkit2gtk3-soup2-devel \
libopenssl-devel \
curl \
wget \
libappindicator3-1 \
librsvg-devel
sudo zypper in -t pattern devel_basis
```

Working on NixOS requires a slightly different setup, as Tauri needs to find the required system libraries both at compile time and dynamically at runtime. To make them available to Tauri the `LD_LIBRARY_PATH` environment variable needs to be populated with the correct paths.

When using [Nix Flakes](#), copy the following code into `flake.nix` on your repository, then run `nix develop` to activate the development environment. You can also use [direnv's Flakes integration](#) to automatically start the dev shell when entering the project folder.

```
{
  inputs = {
    nixpkgs.url = nixpkgs;
    flake-utils.url = github:numtide/flake-utils;
  };

  outputs = { self, nixpkgs, flake-utils }: {
    flake-utils.lib.eachDefaultSystem (system: {
      let
        pkgs = nixpkgs.legacyPackages.${system};

        libraries = with pkgs; [
          webkitgtk
          gtk3
          cairo
          gdk-pixbuf
          glib.out
          dbus.lib
          openssl_3.out
        ];
      packages = with pkgs; [
        curl
        wget
        pkg-config
        dbus
        openssl_3
      ];
    });
  };
}
```

```

glib
gtk3
libsoup
webkitgtk
];
in
{
  devShell = pkgs.mkShell {
    buildInputs = packages;

    shellHook =
      let
        joinLibs = libs: builtins.concatStringsSep : (builtins.map (x:
${x}/lib) libs);
        libs = joinLibs libraries;
      in
        ''
          export LD_LIBRARY_PATH=${libs}:$LD_LIBRARY_PATH
        '';
  };
}
}

```

If you don't use Nix Flakes, the [Nix Shell](#) can be configured using the following `shell.nix` script. Run `nix-shell` to activate the development environment, or use [direnv's Shell integration] to automatically start the dev shell when entering the project folder.

```

let
pkgs = import <nixpkgs> { };

libraries = with pkgs; [
  webkitgtk
  gtk3
  cairo
  gdk-pixbuf
  glib.out
  dbus.lib
  openssl_3.out
];

packages = with pkgs; [
  pkg-config
  dbus
  openssl_3
  glib
  gtk3
  libsoup
  webkitgtk
  appimagekit
];
in
pkgs.mkShell {
  buildInputs = packages;

  shellHook =
    let
      joinLibs = libs: builtins.concatStringsSep : (builtins.map (x: ${x}/lib)
libs);
      libs = joinLibs libraries;
    in
      ''

```

```
    export LD_LIBRARY_PATH=${libs}:$LD_LIBRARY_PATH
}
'
```

2. Rust

To install Rust on Linux, open a terminal and enter the following command:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

We have audited this bash script, and it does what it says it is supposed to do. Nevertheless, before blindly curl-bashing a script, it is always wise to look at it first. Here is the file as a plain script: [rustup.sh](#)

The command downloads a script and starts the installation of the `rustup` tool, which installs the latest stable version of Rust. You might be prompted for your password. If the installation was successful, the following line will appear:

```
Rust is installed now. Great!
```

Make sure to restart your Terminal for the changes to take effect.

Managing The Rust Installation

You should keep your Rust version up to date whenever possible to always benefit from the latest improvements. To update Rust, open a terminal and run the following command:

```
rustup update
```

`rustup` can also be used to uninstall Rust from your machine fully:

```
rustup self uninstall
```

Troubleshooting

To check whether you have Rust installed correctly, open a shell and enter this command:

```
rustc --version
```

You should see the version number, commit hash, and commit date for the latest stable version that has been released in the following format:

```
rustc x.y.z (abcabca... yyyy-mm-dd)
```

If you don't see this information, your Rust installation might be broken. Please consult [Rust's Troubleshooting Section](#) on how to fix this. If your problems persist, you can get help from the official [Tauri Discord](#) and [GitHub Discussions](#).

[direnv's Shell integration]: https://nixos.wiki/wiki/Development_environment_with_nix-shell#direnv

Mocking Tauri APIs

When writing your frontend tests, having a fake Tauri environment to simulate windows or intercept IPC calls is common, so-called *mocking*. The `@tauri-apps/api/mocks` module provides some helpful tools to make this easier for you:

caution

Remember to clear mocks after each test run to undo mock state changes between runs! See `clearMocks()` docs for more info.

IPC Requests

Most commonly, you want to intercept IPC requests; this can be helpful in a variety of situations:

- Ensure the correct backend calls are made
- Simulate different results from backend functions

Tauri provides the `mockIPC` function to intercept IPC requests. You can find more about the specific API in detail [here](#).

The following examples use [Vitest], but you can use any other frontend testing library such as jest.

```
import { beforeAll, expect, test } from vitest;
import { randomFillSync } from crypto;

import { mockIPC } from '@tauri-apps/api/mocks'
import { invoke } from '@tauri-apps/api/tauri';

// jsdom doesn't come with a WebCrypto implementation
beforeAll(() => {
    // @ts-ignore
    window.crypto = {
        getRandomValues: function (buffer) {
            return randomFillSync(buffer);
        },
    };
});

test(invoke simple, async () => {
    mockIPC((cmd, args) => {
        // simulated rust command called add that just adds two numbers
        if(cmd === add) {
            return (args.a as number) + (args.b as number)
        }
    })
    expect(invoke(add, { a: 12, b: 15 })).resolves.toBe(27)
})
```

Sometimes you want to track more information about an IPC call; how many times was the command invoked? Was it invoked at all? You can use `mockIPC()` with other spying and mocking tools to test this:

```
import { beforeAll, expect, test, vi } from vitest;
import { randomFillSync } from crypto;

import { mockIPC } from '@tauri-apps/api/mocks'
import { invoke } from '@tauri-apps/api/tauri';

// jsdom doesn't come with a WebCrypto implementation
beforeAll(() => {
    // @ts-ignore
    window.crypto = {
        getRandomValues: function (buffer) {
            return randomFillSync(buffer);
        },
    };
});

test(invoke, async () => {
    mockIPC((cmd, args) => {
        // simulated rust command called add that just adds two numbers
        if(cmd === add) {
            return (args.a as number) + (args.b as number)
        }
    })

    // we can use the spying tools provided by vitest to track the mocked function
    const spy = vi.spyOn(window, '__TAURI_IPC__')

    expect(invoke(add, { a: 12, b: 15 })).resolves.toBe(27)
    expect(spy).toHaveBeenCalled()
})
})
```

To mock IPC requests to a sidecar or shell command you need to grab the ID of the event handler when `spawn()` or `execute()` is called and use this ID to emit events the backend would send back:

```
mockIPC(async (cmd, args) => {
    if (args.message.cmd === 'execute') {
        const eventCallbackId = `_${args.message.onEventFn}`;
        const eventEmitter = window[eventCallbackId];

        // 'Stdout' event can be called multiple times
        eventEmitter({
            event: 'Stdout',
            payload: 'some data sent from the process',
        });

        // 'Terminated' event must be called at the end to resolve the promise
        eventEmitter({
            event: 'Terminated',
            payload: {
                code: 0,
                signal: 'kill',
            },
        });
    }
});
```

Windows

Sometimes you have window-specific code (a splash screen window, for example), so you need to simulate different windows. You can use the `mockWindows()` method to create fake window labels. The first string identifies the current window (i.e., the window your JavaScript believes itself in), and all other strings are treated as additional windows.

`mockWindows()` only fakes the existence of windows but no window properties. To simulate window properties, you need to intercept the correct calls using `mockIPC()`

```
import { beforeAll, expect, test } from 'vitest'
import { randomFillSync } from 'crypto'

import { mockWindows } from '@tauri-apps/api/mocks'

// jsdom doesn't come with a WebCrypto implementation
beforeAll(() => {
  // @ts-ignore
  window.crypto = {
    getRandomValues: function (buffer) {
      return randomFillSync(buffer)
    },
  }
})

test('invoke', async () => {
  mockWindows('main', 'second', 'third')

  const { getCurrent, getAll } = await import('@tauri-apps/api/window')

  expect(getCurrent()).toHaveProperty('label', 'main')
  expect(getAll().map((w) => w.label)).toEqual(['main', 'second', 'third'])
})
```

[Go to TOC](#)

Continuous Integration

Utilizing Linux and some programs to create a fake display, it is possible to run [WebDriver](#) tests with [tauri-driver](#) on your CI. The following example uses the [WebdriverIO](#) example we [previously built together](#) and GitHub Actions.

This means the following assumptions:

1. The Tauri application is in the repository root and the binary builds when running `cargo build --release`.
2. The [WebDriverIO](#) test runner is in the `webdriver/webdriverio` directory and runs when `yarn test` is used in that directory.

The following is a commented GitHub Actions workflow file at `.github/workflows/webdriver.yml`

```
# run this action when the repository is pushed to
on: [push]

# the name of our workflow
name: WebDriver

jobs:
  # a single job named test
  test:
    # the display name of the test job
    name: WebDriverIO Test Runner

    # we want to run on the latest linux environment
    runs-on: ubuntu-latest

    # the steps our job runs **in order**
    steps:
      # checkout the code on the workflow runner
      - uses: actions/checkout@v2

      # install system dependencies that Tauri needs to compile on Linux.
      # note the extra dependencies for `tauri-driver` to run which are:
      `webkit2gtk-driver` and `xvfb`
      - name: Tauri dependencies
        run: >-
          sudo apt-get update &&
          sudo apt-get install -y
          libgtk-3-dev
          libayatana-appindicator3-dev
          libwebkit2gtk-4.0-dev
          webkit2gtk-driver
          xvfb

      # install the latest Rust stable
      - name: Rust stable
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable

      # we run our rust tests before the webdriver tests to avoid testing a broken
      # application
```

```
- name: Cargo test
  uses: actions-rs/cargo@v1
  with:
    command: test

# build a release build of our application to be used during our WebdriverIO
tests
- name: Cargo build
  uses: actions-rs/cargo@v1
  with:
    command: build
    args: --release

# install the latest stable node version at the time of writing
- name: Node v16
  uses: actions/setup-node@v2
  with:
    node-version: 16.x

# install our Node.js dependencies with Yarn
- name: Yarn install
  run: yarn install
  working-directory: webdriver/webdriverio

# install the latest version of `tauri-driver`.
# note: the tauri-driver version is independent of any other Tauri versions
- name: Install tauri-driver
  uses: actions-rs/cargo@v1
  with:
    command: install
    args: tauri-driver

# run the WebdriverIO test suite.
# we run it through `xvfb-run` (the dependency we installed earlier) to have
a fake
  # display server which allows our application to run headless without any
changes to the code
- name: WebdriverIO
  run: xvfb-run yarn test
  working-directory: webdriver/webdriverio
```

```
import Tabs from '@theme/Tabs' import TabItem from '@theme/TabItem'
```

Selenium

info Example Application This [Selenium] guide expects you to have already gone through the [example Application setup] to follow step-by-step. The general information may still be helpful otherwise.

This WebDriver testing example will use [Selenium](#) and a popular Node.js testing suite. You are expected to already have Node.js installed, along with `npm` or `yarn` although the [finished example project](#) uses `yarn`.

Create a Directory for the Tests

Let's create a space to write these tests in our project. We will be using a nested directory for this example project as we will later also go over other frameworks, but typically you will only need to use one. Create the directory we will use with `mkdir -p webdriver/selenium`. The rest of this guide will assume you are inside the `webdriver/selenium` directory.

Initializing a Selenium Project

We will be using a pre-existing `package.json` to bootstrap this test suite because we have already chosen specific dependencies to use and want to showcase a simple working solution. The bottom of this section has a collapsed guide on how to set it up from scratch.

`package.json`:

```
{
  name: 'selenium',
  version: '1.0.0',
  private: true,
  scripts: {
    test: 'mocha'
  },
  dependencies: {
    chai: '^4.3.4',
    mocha: '^9.0.3',
    selenium-webdriver: '^4.0.0-beta.4'
  }
}
```

We have a script that runs [Mocha](#) as a test framework exposed as the `test` command. We also have various dependencies that we will be using to run the tests. [Mocha](#) as the testing framework, [Chai](#) as the assertion library, and `selenium-webdriver` which is the Node.js [Selenium](#) package.

Click me if you want to see how to set a project up from scratch

► Details

► Details

Testing

Unlike the [WebdriverIO Test Suite](#), Selenium does not come out of the box with a Test Suite and leaves it up to the developer to build those out. We chose [Mocha](#), which is pretty neutral and not related to WebDrivers, so our script will need to do a bit of work to set up everything for us in the correct order. [Mocha](#) expects a testing file at `test/test.js` by default, so let's create that file now.

`test/test.js`:

```
const os = require('os')
const path = require('path')
const { expect } = require('chai')
const { spawn, spawnSync } = require('child_process')
const { Builder, By, Capabilities } = require('selenium-webdriver')

// create the path to the expected application binary
const application = path.resolve(
  __dirname,
  '..',
  '..',
  '..',
  'target',
  'release',
  'hello-tauri-webdriver'
)

// keep track of the webdriver instance we create
let driver

// keep track of the tauri-driver process we start
let tauriDriver

before(async function () {
  // set timeout to 2 minutes to allow the program to build if it needs to
  this.timeout(120000)

  // ensure the program has been built
  spawnSync('cargo', ['build', '--release'])

  // start tauri-driver
  tauriDriver = spawn(
    path.resolve(os.homedir(), '.cargo', 'bin', 'tauri-driver'),
    [],
    { stdio: [null, process.stdout, process.stderr] }
  )

  const capabilities = new Capabilities()
  capabilities.set('tauri:options', { application })
  capabilities.setBrowserName('wry')

  // start the webdriver client
  driver = await new Builder()
    .withCapabilities(capabilities)
    .usingServer('http://localhost:4444/')
    .build()
})
```

```

after(async function () {
  // stop the webdriver session
  await driver.quit()

  // kill the tauri-driver process
  tauriDriver.kill()
})

describe('Hello Tauri', () => {
  it('should be cordial', async () => {
    const text = await driver.findElement(By.css('body > h1')).getText()
    expect(text).to.match(/^[hH]ello/)
  })

  it('should be excited', async () => {
    const text = await driver.findElement(By.css('body > h1')).getText()
    expect(text).to.match(/!$/)
  })

  it('should be easy on the eyes', async () => {
    // selenium returns color css values as rgb(r, g, b)
    const text = await driver
      .findElement(By.css('body'))
      .getCssValue('background-color')

    const rgb = text.match(/^rgb\(((?:<r>\d+),\s*(?:<g>\d+),\s*(?:<b>\d+)\)\)$/.groups)
    expect(rgb).to.have.all.keys('r', 'g', 'b')

    const luma = 0.2126 * rgb.r + 0.7152 * rgb.g + 0.0722 * rgb.b
    expect(luma).to.be.lessThan(100)
  })
})
}
)

```

If you are familiar with JS testing frameworks, `describe`, `it`, and `expect` should look familiar. We also have semi-complex `before()` and `after()` callbacks to setup and teardown mocha. Lines that are not the tests themselves have comments explaining the setup and teardown code. If you were familiar with the Spec file from the [WebdriverIO example](#), you notice a lot more code that isn't tests, as we have to set up a few more WebDriver related items.

Running the Test Suite

Now that we are all set up with our dependencies and our test script, let's run it!

```
<Tabs groupId=package-manager defaultValue=yarn values=[{label: 'npm', value: 'npm'}, {label: 'Yarn', value: 'yarn'}], ]>
```

`npm test`

`yarn test`

We should see output the following output:

```

→ selenium git:(main) ✘ yarn test
yarn run v1.22.11
$ Mocha

```

```
Hello Tauri
✓ should be cordial (120ms)
✓ should be excited
✓ should be easy on the eyes
```

3 passing (588ms)

Done in 0.93s.

We can see that our `Hello Tauri` sweet we created with `describe` had all 3 items we created with `it` pass their tests!

With [Selenium](#) and some hooking up to a test suite, we just enabled e2e testing without modifying our Tauri application at all!

```
import HelloTauriWebdriver from '@site/static/img/webdriver/hello-tauri-webdriver.png'
```

Setup Example

This example application solely focuses on adding WebDriver testing to an already existing project. To have a project to test in the following two sections, we will set up an extremely minimal Tauri application for use in our testing. We will not use the Tauri CLI, any frontend dependencies or build steps, and not be bundling the application afterward. This is to showcase exactly a minimal suite to show off adding WebDriver testing to an existing application.

If you just want to see the finished example project that utilizes what will be shown in this example guide, then you can see https://github.com/chippers/hello_tauri.

Initializing a Cargo Project

We want to create a new binary Cargo project to house this example application. We can easily do this from the command line with `cargo new hello-tauri-webdriver --bin`, which will scaffold a minimal binary Cargo project for us. This directory will serve as the working directory for the rest of this guide, so make sure commands you run are inside this new `hello-tauri-webdriver/` directory.

Creating a Minimal Frontend

We will create a minimal HTML file to act as our example application's front end. We will also be using a few things from this frontend later during our WebDriver tests.

First, let's create our Tauri `distDir` that we know we will need once building the Tauri portion of the application. `mkdir dist` should create a new directory called `dist/` in which we will be placing the following `index.html` file.

`dist/index.html`:

```
<!DOCTYPE html>
<html lang=en>
<head>
  <meta charset=UTF-8 />
  <title>Hello Tauri!</title>
  <style>
    body {
      /* Add a nice colorscheme */
      background-color: #222831;
      color: #ececce;

      /* Make the body the exact size of the window */
      margin: 0;
      height: 100vh;
      width: 100vw;

      /* Vertically and horizontally center children of the body tag */
      display: flex;
      justify-content: center;
```

```

        align-items: center;
    }
</style>
</head>
<body>
    <h1>Hello, Tauri!</h1>
</body>
</html>

```

Adding Tauri to the Cargo Project

Next, we will add necessary items to turn our Cargo project into a Tauri project. First, is adding the dependencies to the Cargo Manifest (`Cargo.toml`) so that Cargo knows to pull in our dependencies while building.

`Cargo.toml`:

```

[package]
name = hello-tauri-webdriver
version = 0.1.0
edition = 2021
rust-version = 1.56

# Needed to set up some things for Tauri at build time
[build-dependencies]
tauri-build = 1

# The actual Tauri dependency, along with `custom-protocol` to serve the pages.
[dependencies]
tauri = { version = 1, features = [custom-protocol] }

# Make --release build a binary that is small (opt-level = s) and fast (lto = true).
# This is completely optional, but shows that testing the application as close to
the
# typical release settings is possible. Note: this will slow down compilation.

[profile.release]
incremental = false
codegen-units = 1
panic = abort
opt-level = s
lto = true

```

We added a `[build-dependency]` as you may have noticed. To use the build dependency, we must use it from a build script. We will create one now at `build.rs`.

`build.rs`:

```

fn main() {
    // Only watch the `dist/` directory for recompiling, preventing unnecessary
    // changes when we change files in other project subdirectories.
    println!(cargo:rerun-if-changed=dist);

    // Run the Tauri build-time helpers
    tauri_build::build()
}

```

Our Cargo Project now knows how to pull in and build our Tauri dependencies with all that setup. Let's finish making this minimal example a Tauri application by setting up Tauri in the actual project code. We will be editing the `src/main.rs` file to add this Tauri functionality.

`src/main.rs`:

```
fn main() {
    tauri::Builder::default()
        .run(tauri::generate_context!())
        .expect(unable to run Tauri application);
}
```

Pretty simple, right?

Tauri Configuration

We are going to need 2 things to successfully build the application. First, we need an icon file. You can use any PNG for this next part and copy it into `icon.png`. Typically, this will be provided as part of the scaffolding when you use the Tauri CLI to create a project. To get the default Tauri icon, we can download the icon used by the Hello Tauri example repository with the command `curl -L https://github.com/chippers/hello_tauri/raw/main/icon.png --output icon.png`.

We will need a `tauri.conf.json` to set some important configuration values for Tauri. Again, this would typically come from the `tauri init` scaffolding command, but we will be creating our own minimal config here.

`tauri.conf.json`:

```
{
  build: {
    distDir: dist
  },
  tauri: {
    bundle: {
      identifier: studio.tauri.hello_tauri_webdriver,
      icon: [icon.png]
    },
    allowlist: {
      all: false
    },
    windows: [
      {
        width: 800,
        height: 600,
        resizable: true,
        fullscreen: false
      }
    ]
  }
}
```

I'll go over some of these. You can see the `dist/` directory we created earlier specified as the `distDir` property. We set a bundle identifier so that the built application has a unique id and set the `icon.png` as the only icon. We aren't using any Tauri APIs or features, so we disable them in `allowlist` by setting `all: false`. The window values just set a single window to be created with some reasonable default values.

At this point, we have a basic Hello World application that should display a simple greeting when run.

Running the Example Application

To make sure we did it right, let's build this application! We will run this as a `--release` application because we will also run our WebDriver tests with a release profile. Run `cargo run --release`, and after some compiling, we should see the following application pop up.



Note: If you are modifying the application and want to use the Devtools, then run it without `--release` and Inspect

Element should be available in the right-click menu._

We should now be ready to start testing this application with some WebDriver frameworks. This guide will go over both [WebdriverIO](#) and [Selenium](#) in that order.

```
import Tabs from '@theme/Tabs' import TabItem from '@theme/TabItem'
```

WebdriverIO

info Example Application This [WebdriverIO] guide expects you to have already gone through the [example Application setup] to follow step-by-step. The general information may still be helpful otherwise.

This WebDriver testing example will use [WebdriverIO](#), and its testing suite. It is expected to have Node.js already installed, along with `npm` or `yarn` although the [finished example project](#) uses `yarn`.

Create a Directory for the Tests

Let's create a space to write these tests in our project. We will be using a nested directory for this example project as we will later also go over other frameworks, but typically you only need to use one. Create the directory we will use with `mkdir -p webdriver/webdriverio`. The rest of this guide assumes you are inside the `webdriver/webdriverio` directory.

Initializing a WebdriverIO Project

We will be using a pre-existing `package.json` to bootstrap this test suite because we have already chosen specific [WebdriverIO](#) config options and want to showcase a simple working solution. The bottom of this section has a collapsed guide on setting it up from scratch.

`package.json`:

```
{
  name: 'webdriverio',
  version: '1.0.0',
  private: true,
  scripts: {
    test: 'wdio run wdio.conf.js'
  },
  dependencies: {
    '@wdio/cli': '^7.9.1'
  },
  devDependencies: {
    '@wdio/local-runner': '^7.9.1',
    '@wdio/mocha-framework': '^7.9.1',
    '@wdio/spec-reporter': '^7.9.0'
  }
}
```

We have a script that runs a [WebdriverIO](#) config as a test suite exposed as the `test` command. We also have various dependencies added by the `@wdio/cli` command when we first set it up. In short, these dependencies are for the most simple setup using a local WebDriver runner, [Mocha](#) as the test framework, and a simple Spec Reporter.

Click me if you want to see how to set a project up from scratch

Config

You may have noticed that the `test` script in our `package.json` mentions a file `wdio.conf.js`. That's the [WebdriverIO](#) config file which controls most aspects of our testing suite.

`wdio.conf.js`:

```
const os = require('os')
const path = require('path')
const { spawn, spawnSync } = require('child_process')

// keep track of the `tauri-driver` child process
let tauriDriver

exports.config = {
  specs: ['./test/specs/**/*.*js'],
  maxInstances: 1,
  capabilities: [
    {
      maxInstances: 1,
      'tauri:options': {
        application: '../target/release/hello-tauri-webdriver',
      },
    },
  ],
  reporters: ['spec'],
  framework: 'mocha',
  mochaOpts: {
    ui: 'bdd',
    timeout: 60000,
  },
  // ensure the rust project is built since we expect this binary to exist for the
  // webdriver sessions
  onPrepare: () => spawnSync('cargo', ['build', '--release']),

  // ensure we are running `tauri-driver` before the session starts so that we can
  // proxy the webdriver requests
  beforeSession: () =>
    (tauriDriver = spawn(
      path.resolve(os.homedir(), '.cargo', 'bin', 'tauri-driver'),
      [],
      { stdio: [null, process.stdout, process.stderr] }
    )),

  // clean up the `tauri-driver` process we spawned at the start of the session
  afterSession: () => tauriDriver.kill(),
}
```

If you are interested in the properties on the `exports.config` object, I [suggest reading the documentation](#). For non-WDIO specific items, there are comments explaining why we are running commands in `onPrepare`, `beforeSession`, and `afterSession`. We also have our specs set to `./test/specs/**/*.*js`, so let's create a spec now.

Spec

A spec contains the code that is testing your actual application. The test runner will load these specs and automatically run them as it sees fit. Let's create our spec now in the directory we specified.

test/specs/example.e2e.js :

```
// calculates the luma from a hex color `#abcdef`
function luma(hex) {
  if (hex.startsWith('#')) {
    hex = hex.substring(1)
  }

  const rgb = parseInt(hex, 16)
  const r = (rgb >> 16) & 0xff
  const g = (rgb >> 8) & 0xff
  const b = (rgb >> 0) & 0xff
  return 0.2126 * r + 0.7152 * g + 0.0722 * b
}

describe('Hello Tauri', () => {
  it('should be cordial', async () => {
    const header = await $('body > h1')
    const text = await header.getText()
    expect(text).toMatch(/^[hH]ello/)
  })

  it('should be excited', async () => {
    const header = await $('body > h1')
    const text = await header.getText()
    expect(text).toMatch(/!$/)
  })

  it('should be easy on the eyes', async () => {
    const body = await $('body')
    const backgroundColor = await body.getCSSProperty('background-color')
    expect(luma(backgroundColor.parsed.hex)).toBeLessThan(100)
  })
})
```

The `luma` function on top is just a helper function for one of our tests and is not related to the actual testing of the application. If you are familiar with other testing frameworks, you may notice similar functions being exposed that are used, such as `describe`, `it`, and `expect`. The other APIs, such as items like `$` and its exposed methods, are covered by the [WebdriverIO API docs](#).

Running the Test Suite

Now that we are all set up with config and a spec let's run it!

```
<Tabs groupId=package-manager defaultValue=yarn values={[ {label: 'npm', value: 'npm'}, {label: 'Yarn', value: 'yarn'}], }>
```

`npm test`

`yarn test`

We should see output the following output:

```
→ webdriverio git:(main) ✘ yarn test
yarn run v1.22.11
$ wdio run wdio.conf.js

Execution of 1 workers started at 2021-08-17T08:06:10.279Z

[0-0] RUNNING in undefined - /test/specs/example.e2e.js
[0-0] PASSED in undefined - /test/specs/example.e2e.js

spec Reporter:
-----
[wry 0.12.1 linux #0-0] Running: wry (v0.12.1) on linux
[wry 0.12.1 linux #0-0] Session ID: 81e0107b-4d38-4eed-9b10-ee80ca47bb83
[wry 0.12.1 linux #0-0]
[wry 0.12.1 linux #0-0] » /test/specs/example.e2e.js
[wry 0.12.1 linux #0-0] Hello Tauri
[wry 0.12.1 linux #0-0]     ✓ should be cordial
[wry 0.12.1 linux #0-0]     ✓ should be excited
[wry 0.12.1 linux #0-0]     ✓ should be easy on the eyes
[wry 0.12.1 linux #0-0]
[wry 0.12.1 linux #0-0] 3 passing (244ms)

Spec Files:      1 passed, 1 total (100% completed) in 00:00:01

Done in 1.98s.
```

We see the Spec Reporter tell us that all 3 tests from the `test/specs/example.e2e.js` file, along with the final report `Spec Files: 1 passed, 1 total (100% completed) in 00:00:01`.

Using the [WebdriverIO](#) test suite, we just easily enabled e2e testing for our Tauri application from just a few lines of configuration and a single command to run it! Even better, we didn't have to modify the application at all.

caution Currently in pre-alpha Webdriver support for Tauri is still in pre-alpha. Tooling that is dedicated to it, such as [tauri-driver], is still in active development and may change as necessary over time. Additionally, only Windows and Linux are currently supported.

WebDriver is a standardized interface to interact with web documents primarily intended for automated testing. Tauri supports the WebDriver interface by leveraging the native platform's WebDriver server underneath a cross-platform wrapper `tauri-driver`.

System Dependencies

Install the latest `tauri-driver` or update an existing installation by running:

```
cargo install tauri-driver
```

Because we currently utilize the platform's native WebDriver server, there are some requirements for running `tauri-driver` on supported platforms. Platform support is currently limited to Linux and Windows.

Linux

We use `WebKitWebDriver` on Linux platforms. Check if this binary exists already (command `which WebKitWebDriver`) as some distributions bundle it with the regular WebKit package. Other platforms may have a separate package for them, such as `webkit2gtk-driver` on Debian-based distributions.

Windows

Make sure to grab the version of Microsoft Edge Driver that matches your Windows' Edge version that the application is being built and tested on. This should almost always be the latest stable version on up-to-date Windows installs. If the two versions do not match, you may experience your WebDriver testing suite hanging while trying to connect.

The download contains a binary called `msedgedriver.exe`. `tauri-driver` looks for that binary in the `$PATH` so make sure it's either available on the path or use the `--native-driver` option on `tauri-driver`. You may want to download this automatically as part of the CI setup process to ensure the Edge, and Edge Driver versions stay in sync on Windows CI machines. A guide on how to do this may be added at a later date.

Example Application

The next section of the guide shows step-by-step how to create a minimal example application that is tested with WebDriver.

If you prefer to see the result of the guide and look over a finished minimal codebase that utilizes it, you can look at https://github.com/chippers/hello_tauri. That example also comes with a CI script to test with GitHub actions, but you may still be interested in the WebDriver CI guide as it explains the concept a bit more.

[Go to TOC](#)

Brownfield Pattern

This is the default pattern.

This is the simplest and most straightforward pattern to use Tauri with, because it tries to be as compatible as possible with existing frontend projects. In short, it tries to require nothing additional to what an existing web frontend might use inside a browser. Not **everything** that works in existing browser applications will work out-of-the-box; see the [Incompatibility section](#) for more details.

If you are unfamiliar with Brownfield software development in general, the [Brownfield Wikipedia article](#) provides a nice summary. For Tauri, the existing software is current browser support and behavior, instead of legacy systems.

Incompatibilities

The first incompatibility category is simple: any browser-specific APIs will not work properly inside Tauri (even while using the Brownfield pattern). If the API is not widely supported across browsers, it probably won't be supported across all platforms while using Tauri.

The second incompatibility category is features that are planned for Tauri, but are currently not fully implemented. Here is a list of examples:

- [WebRTC support on Linux](#)
- [Some permissions APIs](#)
- [Download Links/Blob as URL](#)
- [Better i18n](#)

Configuration

Because the Brownfield pattern is the default pattern, it doesn't require a configuration option to be set. To explicitly set it, you can use the `tauri > pattern` object in the `tauri.conf.json` configuration file.

```
{
  tauri: {
    pattern: {
      use: brownfield
    }
  }
}
```

There are no additional configuration options for the brownfield pattern.

Isolation Pattern

The Isolation pattern is a way to intercept and modify Tauri API messages sent by the frontend before they get to Tauri Core, all with JavaScript. The secure JavaScript code that is injected by the Isolation pattern is referred to as the Isolation application.

Why

The Isolation pattern's purpose is to provide a mechanism for developers to help protect their application from unwanted or malicious frontend calls to Tauri Core. The need for the Isolation pattern rose out of threats coming from untrusted content running on the frontend, a common case for applications with many dependencies. See [Security: Threat Models](#) for a list of many sources of threats that an application may see.

The largest threat model described above that the Isolation pattern was designed in mind was Development Threats. Not only do many frontend build-time tools consist of many dozen (or hundreds) of often deeply-nested dependencies, but a complex application may also have a large amount of (also often deeply-nested) dependencies that are bundled into the final output.

When

Tauri highly recommends using the isolation pattern whenever it can be used. Because the Isolation application intercepts **all** messages from the frontend, it can *always* be used.

Tauri also strongly suggests locking down your application whenever you use external Tauri APIs. As the developer, you can utilize the secure Isolation application to try and verify IPC inputs, to make sure they are within some expected parameters. For example, you may want to check that a call to read or write a file is not trying to access a path outside your application's expected locations. Another example is making sure that a Tauri API HTTP fetch call is only setting the Origin header to what your application expects it to be.

That said, it intercepts **all** messages from the frontend, so it will even work with always-on APIs such as [Events](#). Since some events may cause your own rust code to perform actions, the same sort of validation techniques can be used with them.

How

The Isolation pattern is all about injecting a secure application in between your frontend and Tauri Core to intercept and modify incoming IPC messages. It does this by using the sandboxing feature of `<iframe>`s to run the JavaScript securely alongside the main frontend application. Tauri enforces the Isolation pattern while loading the page, forcing all IPC calls to Tauri Core to instead be routed through the sandboxed Isolation application first. Once the message is ready to be passed to Tauri Core, it is encrypted using the browser's [SubtleCrypto](#) implementation and passed back to the main frontend application. Once there, it is directly passed to Tauri Core, where it is then decrypted and read like normal.

To ensure that someone cannot manually read the keys for a specific version of your application and use that to modify the messages after being encrypted, new keys are generated each time your application is run.

Approximate Steps of an IPC Message

To make it easier to follow, here's an ordered list with the approximate steps an IPC message will go through when being sent to Tauri Core with the Isolation pattern:

1. Tauri's IPC handler receives a message
2. IPC handler -> Isolation application
3. [sandbox] Isolation application hook runs and potentially modifies the message
4. [sandbox] Message is encrypted with AES-GCM using a runtime-generated key
5. [encrypted] Isolation application -> IPC handler
6. [encrypted] IPC handler -> Tauri Core

Note: Arrows (->) indicate message passing.

Performance Implications

Because encryption of the message does occur, there are additional overhead costs compared to the [Brownfield pattern](#), even if the secure Isolation application doesn't do anything. Aside from performance-sensitive applications (who likely have a carefully-maintained and small set of dependencies, to keep the performance adequate), most applications should not notice the runtime costs of encrypting/decrypting the IPC messages, as they are relatively small and AES-GCM is relatively fast. If you are unfamiliar with AES-GCM, all that is relevant in this context is that it's the only authenticated mode algorithm included in [SubtleCrypto](#) and that you probably already use it every day under the hood with [TLS](#).

There is also a cryptographically secure key generated once each time the Tauri application is started. It is not generally noticeable if the system already has enough entropy to immediately return enough random numbers, which is extremely common for desktop environments. If running in a headless environment to perform some [integration testing with WebDriver](#) then you may want to install some sort of entropy-generating service such as [haveged](#) if your operating system does not have one included. Linux 5.6 (March 2020) now includes entropy generation using speculative execution.

Limitations

There are a few limitations in the Isolation pattern that arose out of platform inconsistencies. The most significant limitation is due to external files not loading correctly inside sandboxed `<iframes>` on Windows. Because of this, we have implemented a simple script inlining step during build time that takes the content of scripts relative to the Isolation application and injects them inline. This means that typical bundling or simple including of files like `<script src=index.js></script>` still works properly, but newer mechanisms such as ES Modules will *not* successfully load.

Recommendations

Because the point of the Isolation application is to protect against Development Threats, we highly recommend keeping your Isolation application as simple as possible. Not only should you strive to keep dependencies minimal, but you should also consider keeping required build steps minimal. This would allow you to not need to worry about supply chain attacks against your Isolation application on top of your frontend application.

Creating the Isolation Application

In this example, we will make a small hello-world style Isolation application and hook it up to an imaginary existing Tauri application. It will do no verification of the messages passing through it, only print the contents to the WebView console.

For the purposes of this example, let's imagine we are in the same directory as `tauri.conf.json`. The existing Tauri application has it's `distDir` set to `../dist`.

```
../dist-isolation/index.html :
```

```
<!DOCTYPE html>
<html lang=en>
  <head>
    <meta charset=UTF-8 />
    <title>Isolation Secure Script</title>
  </head>
  <body>
    <script src=index.js></script>
  </body>
</html>
```

```
../dist-isolation/index.js :
```

```
window.__TAURI_ISOLATION_HOOK__ = (payload) => {
  // let's not verify or modify anything, just print the content from the hook
  console.log('hook', payload)
  return payload
}
```

Now, all we need to do is set up our `tauri.conf.json` configuration to use the Isolation pattern, and have just bootstrapped to the Isolation pattern from the [Brownfield pattern](#).

Configuration

Let's assume that our main frontend `distDir` is set to `../dist`. We also output our Isolation application to `../dist-isolation`.

```
{
  build: {
    distDir: ../dist
  },
  tauri: {
    pattern: {
```

```
use: isolation,
options: {
  dir: '../dist-isolation
}
}
```

Inter-Process Communication

Inter-Process Communication (IPC) allows isolated processes to communicate securely and is key to building more complex applications.

Tauri uses a particular style of Inter-Process Communication called [Asynchronous Message Passing](#), where processes exchange *requests* and *responses* serialized using some simple data representation. Message Passing should sound familiar to anyone with web development experience, as this paradigm is used for client-server communication on the internet.

Message passing is a safer technique than shared memory or direct function access because the recipient is free to reject or discard requests as it sees fit. For example, if the Tauri Core process determines a request to be malicious, it simply discards the requests and never executes the corresponding function.

In the following, we explain Tauri's two IPC primitives - [Events](#) and [Commands](#) - in more detail.

Events

Events are fire-and-forget, one-way IPC messages that are best suited to communicate lifecycle events and state changes. Unlike [Commands](#), Events can be emitted by both the Frontend *and* the Tauri Core.

```
sequenceDiagram
    participant F as Frontend
    participant C as Tauri Core
    C->F: Event
```

Figure 1-2: An event sent from the Core to the Frontend.

Commands

Tauri also provides a [foreign function interface](#)-like abstraction on top of IPC messages¹. The primary API, `invoke`, is similar to the browser's `fetch` API and allows the Frontend to invoke Rust functions, pass arguments, and receive data.

Because this mechanism uses a [JSON-RPC](#) like protocol under the hood to serialize requests and responses, all arguments and return data must be serializable to JSON.

```
sequenceDiagram
    participant F as Frontend
    participant C as Tauri Core
    F->+C: IPC request
    note over C: Perform computation, write to file system, etc.
    C->-F: Response
```

Figure 1-3: IPC messages involved in a command invocation.

1. Commands still use message passing under the hood, so don't share the same security pitfalls as real FFI interfaces.[←](#)

Process Model

Tauri employs a multi-process architecture similar to Electron or many modern web browsers. This guide explores the reasons behind the design choice and why it is key to writing secure applications.

Why Multiple Processes?

In the early days of GUI applications, it was common to use a single process to perform computation, draw the interface and react to user input. As you can probably guess, this meant that a long-running, expensive computation would leave the user interface unresponsive, or worse, a failure in one app component would bring the whole app crashing down.

It became clear that a more resilient architecture was needed, and applications began running different components in different processes. This makes much better use of modern multi-core CPUs and creates far safer applications. A crash in one component doesn't affect the whole system anymore, as components are isolated on different processes. If a process gets into an invalid state, we can easily restart it.

We can also limit the blast radius of potential exploits by handing out only the minimum amount of permissions to each process, just enough so they can get their job done. This pattern is known as the [Principle of Least Privilege](#), and you see it in the real world all the time. If you have a gardener coming over to trim your hedge, you give them the key to your garden. You would **not** give them the keys to your house; why would they need access to that? The same concept applies to computer programs. The less access we give them, the less harm they can do if they get compromised.

The Core Process

Each Tauri application has a core process, which acts as the application's entry point and which is the only component with full access to the operating system.

The Core's primary responsibility is to use that access to create and orchestrate application windows, system-tray menus, or notifications. Tauri implements the necessary cross-platform abstractions to make this easy. It also routes all [Inter-Process Communication](#) through the Core process, allowing you to intercept, filter, and manipulate IPC messages in one central place.

The Core process should also be responsible for managing global state, such as settings or database connections. This allows you to easily synchronize state between windows and protect your business-sensitive data from prying eyes in the Frontend.

We chose Rust to implement Tauri because of its concept of [Ownership](#) guarantees memory safety while retaining excellent performance.

```
flowchart TD
    C[Core]
    W1[WebView]
    W2[WebView]
    W3[WebView]
```

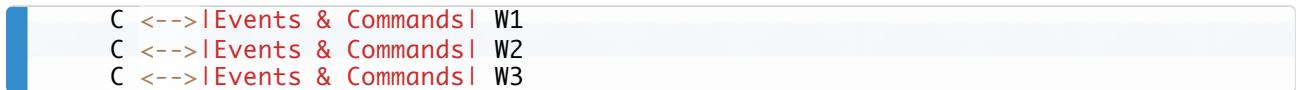


Figure 1-1: Simplified representation of the Tauri process model. A single Core process manages one or more WebView processes.

The WebView Process

The Core process doesn't render the actual user interface (UI) itself; it spins up WebView processes that leverage WebView libraries provided by the operating system. A WebView is a browser-like environment that executes your HTML, CSS, and JavaScript.

This means that most of your techniques and tools used in traditional web development can be used to create Tauri applications. For example, many Tauri examples are written using the [Svelte](#) frontend framework and the [Vite](#) bundler.

Security best practices apply as well; for example, you must always sanitize user input, never handle secrets in the Frontend, and ideally defer as much business logic as possible to the Core process to keep your attack surface small.

Unlike other similar solutions, the WebView libraries are **not** included in your final executable but dynamically linked at runtime¹. This makes your application *significantly* smaller, but it also means that you need to keep platform differences in mind, just like traditional web development.

1. Currently, Tauri uses [Microsoft Edge WebView2](#) on Windows, [WKWebView](#) on macOS and [webkitgtk](#) on Linux. ↩

```
import { colors } from '@theme/Mermaid' import { Mermaid } from 'mdx-mermaid/Mermaid';
```

Tauri Architecture

Introduction

Tauri is a polyglot and generic toolkit that is very composable and allows engineers to make a wide variety of applications. It is used for building applications for desktop computers using a combination of Rust tools and HTML rendered in a Webview. Apps built with Tauri can ship with any number of pieces of an optional JS API and Rust API so that webviews can control the system via message passing. Developers can extend the default API with their own functionality and bridge the Webview and Rust-based backend easily.

Tauri apps can have [custom menus](#) and [tray-type interfaces](#). They can be [updated](#) and are managed by the user's operating system as expected. They are [very small](#) because they use the OS's webview. They do not ship a runtime since the final binary is compiled from Rust. This makes the [reversing of Tauri apps not a trivial task](#).

What Tauri is Not

Tauri is not a lightweight kernel wrapper. Instead, it directly uses [WRY](#) and [TAO](#) to do the heavy lifting in making system calls to the OS.

Tauri is not a VM or virtualized environment. Instead, it is an application toolkit that allows making Webview OS applications.

Core Ecosystem

```
<Mermaid chart={ graph TB; subgraph Core direction LR subgraph tauri direction TB tauri-runtime tauri-macros tauri-utils end %% This section should be organized from top to bottom tauri-build tauri-codegen tauri-runtime-wry end tauri-runtime-wry -.--> WRY subgraph Upstream direction LR WRY TAO WRY -.--> TAO end style Core fill:${colors.blue.light},stroke:${colors.blue.dark},stroke-width:4px style Upstream fill:${colors.blue.light},stroke:${colors.blue.dark},stroke-width:4px style tauri fill:${colors.orange.light},stroke:${colors.orange.dark},stroke-width:4px } />
```

[tauri](#)

This is the major crate that holds everything together. It brings the runtimes, macros, utilities and API into one final product. It reads the `tauri.conf.json` file at compile time to bring in features and undertake the actual configuration of the app (and even the `Cargo.toml` file in the project's folder). It handles script injection (for polyfills / prototype revision) at runtime, hosts the API for systems interaction, and even manages the updating process.

tauri-runtime

The glue layer between Tauri itself and lower-level webview libraries.

tauri-macros

Creates macros for the context, handler, and commands by leveraging the `tauri-codegen` crate.

tauri-utils

Common code that is reused in many places and offers useful utilities like parsing configuration files, detecting platform triples, injecting the CSP, and managing assets.

tauri-build

Applies the macros at build-time to rig some special features needed by `cargo`.

tauri-codegen

Embeds, hashes, and compresses assets, including icons for the app as well as the system tray. Parses `tauri.conf.json` at compile time and generates the Config struct.

tauri-runtime-wry

This crate opens up direct systems-level interactions specifically for WRY, such as printing, monitor detection, and other windowing-related tasks.

Tauri Tooling

API (JavaScript / TypeScript)

A typescript library that creates `cjs` and `esm` JavaScript endpoints for you to import into your frontend framework so that the Webview can call and listen to backend activity. Also ships in pure typescript, because for some frameworks this is more optimal. It uses the message passing of webviews to their hosts.

Bundler (Rust / Shell)

A library that builds a Tauri app for the platform it detects or is told. Currently supports macOS, Windows and Linux - but in the near future will support mobile platforms as well. May be used outside of Tauri projects.

cli.rs (Rust)

This Rust executable provides the full interface to all of the required activities for which the CLI is required. It runs on macOS, Windows, and Linux.

cli.js (JavaScript)

Wrapper around `cli.rs` using `napi-rs` to produce npm packages for each platform.

[create-tauri-app \(JavaScript\)](#)

A toolkit that will enable engineering teams to rapidly scaffold out a new `tauri-apps` project using the frontend framework of their choice (as long as it has been configured).

Upstream Crates

The Tauri-Apps organisation maintains two upstream crates from Tauri, namely TAO for creating and managing application windows, and WRY for interfacing with the Webview that lives within the window.

[TAO](#)

Cross-platform application window creation library in Rust that supports all major platforms like Windows, macOS, Linux, iOS and Android. Written in Rust, it is a fork of `winit` that we have extended for our own needs - like menu bar and system tray.

[WRY](#)

WRY is a cross-platform WebView rendering library in Rust that supports all major desktop platforms like Windows, macOS, and Linux. Tauri uses WRY as the abstract layer responsible to determine which webview is used (and how interactions are made).

Additional Tooling

[tauri-action](#)

GitHub workflow that builds Tauri binaries for all platforms. Even allows creating a (very basic) Tauri app even if Tauri is not set up.

[tauri-vscode](#)

This project enhances the Visual Studio Code interface with several nice-to-have features.

[vue-cli-plugin-tauri](#)

Allows you to very quickly install Tauri in a vue-cli project.

Plugins

[Tauri Plugin Guide](#)

Generally speaking, plugins are authored by third parties (even though there may be official, supported plugins). A plugin generally does 3 things:

1. Enables Rust code to do something.
2. Provides interface glue to make it easy to integrate into an app.
3. Provides a JavaScript API for interfacing with the Rust code.

Here are some examples of Tauri Plugins:

- [tauri-plugin-sql](#)
- [tauri-plugin-stronghold](#)
- [tauri-plugin-authenticator](#)

License

Tauri itself is licensed under MIT or Apache-2.0. If you repackage it and modify any source code, it is your responsibility to verify that you are complying with all upstream licenses. Tauri is provided AS-IS with no explicit claim for suitability for any purpose.

Here you may peruse our [Software Bill of Materials](#).

Next Steps

[Your First Tauri App](#)

[Development Cycle](#)

[Publishing](#)

[Updating](#)

[Go to TOC](#)

Tauri recipes are descriptions of use cases that are entirely configurable within the `src-tauri/tauri.conf.json` file. These are not the limits of what Tauri can do, and there are probably more out there. If you discover one, please get in touch and help us update this collection!

If you haven't read about the general design of Tauri, then it would make the most sense for you to visit the [Quick Start](#) guide and become familiar with the basic architecture and terminology used in these recipes.

[Go to TOC](#)

```
import Rater from '@theme/Rater' import useBaseUrl from '@docusaurus/useBaseUrl'
```

Ease of Use
Extensibility
Performance
Security

Bridge

Pros:

- Highly configurable
- No Rust skills required

Cons:

- Some WebAPIs unavailable
- Challenge to implement

Description

The Bridge recipe is a secure pattern where messages are passed between brokers via an implicit bridge using the API. It isolates functionality to the specific scope and passes messages instead of functionality.

Diagram

```
graph TD
    H==>F
    subgraph WEBVIEW
        F-. .-E
    end
    D--->E
    E--->D
    B--->D
    D--->B
    subgraph RUST
        A==>H
        A--->B
        B-. .-C
        B-. .-G
    end
    A[Binary]
    B{Rust Broker}
    C[Subprocess 2]
    G[Subprocess 1]
    D(( API BRIDGE ))
    E{JS Broker}
    F[Window]
    H{Bootstrap}
    class D apibridge
    class RUST rust
    class WEBVIEW webview
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
{
  tauri: {
    allowlist: {
      all: false,
      clipboard: {
        all: false,
        readText: false,
        writeText: false
      },
      dialog: {
        all: false,
        ask: false,
        confirm: false,
        message: false,
        open: false,
        save: false
      },
      fs: {
        all: false,
        copyFile: false,
        createDir: false,
        readDir: false,
        readFile: false,
        removeDir: false,
        removeFile: false,
        renameFile: false,
        scope: [],
        writeFile: false
      },
      globalShortcut: {
        all: false
      },
      http: {
        all: false,
        request: false,
        scope: []
      },
      notification: {
        all: false
      },
      os: {
        all: false
      },
      path: {
        all: false
      },
      process: {
        all: false,
        exit: false,
        relaunch: false,
        relaunchDangerousAllowSymlinkMacos: false
      },
      protocol: {
        all: false,
        asset: false,
        assetScope: []
      }
  }
}
```

```
shell: {
  all: false,
  execute: false,
  open: false,
  scope: [],
  sidecar: false
},
window: {
  all: false,
  center: false,
  close: false,
  create: false,
  hide: false,
  maximize: false,
  minimize: false,
  print: false,
  requestUserAttention: false,
  setAlwaysOnTop: false,
  setDecorations: false,
  setFocus: false,
  setFullscreen: false,
  setIcon: false,
  setMaxSize: false,
  setMinSize: false,
  setPosition: false,
  setResizable: false,
  setSize: false,
  setSkipTaskbar: false,
  setTitle: false,
  show: false,
  startDragging: false,
  unmaximize: false,
  unminimize: false
}
}
```

```
import Rater from '@theme/Rater' import useBaseUrl from '@docusaurus/useBaseUrl'
```

Ease of Use
Extensibility
Performance
Security

Cloubridge

Pros:

- All available features
- No Rust skills required

Cons:

- Largest bundle size
- Hard to separate concerns

Description

The Cloubridge recipe combines the flexibility of a localhost and the security of the bridge. With so many features, it can be easy to get lost.

Diagram

```
graph TD
    H==>F2
    H==>D2
    D2--->F2
    F2--->D2
    B--->D
    D--->B
    E2--->D
    D--->E2
    subgraph WEBVIEW
        F2
        E2
    end
    subgraph SERVER
        D2
        E--->D2
    end
    subgraph RUST
        A==>H
        A--->B
        B--->C
    end
    A[Binary]
    B{Rust Broker}
    C[Subprocess]
    D(( API BRIDGE ))
    E{JS Broker}
    D2(( localhost ))
```

```
E[bundled resources]
E2{JS Broker}
F2[Window]
H{Bootstrap}
class D apibridge
class RUST rust
class WEBVIEW webview
class SERVER server
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {
  allowlist: {
    all: true           // enable entire API
  }
}
```

```
import Rater from '@theme/Rater' import useBaseUrl from '@docusaurus/useBaseUrl'
```

Ease of Use
Extensibility
Performance
Security

Cloudish

Pros:

- Similar to a SPA web-app
- No Rust skills required

Cons:

- No access to Rust API
- Uses a localhost server

Description

The Cloudish recipe is a pattern for maximum flexibility and app performance. It uses a localhost server, which means that your app will technically be available to other processes, like browsers and potentially other devices on the network. All of your assets are baked into the binary but served as if they were distinct files.

Diagram

```
graph TD
    H==>F
    H==>D
    D-->F
    F-->D
    subgraph RUST
        A==>H
    end
    subgraph WEBVIEW
        F
    end
    subgraph SERVER
        D
        E-->D
    end
    A[Binary]
    D(( localhost ))
    E[bundled resources]
    F[Window]
    H{Bootstrap}
    class RUST rust
    class WEBVIEW webview
    class SERVER server
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {  
    allowlist: {  
        all: false // disable entire API  
    }  
}
```

```
import useBaseUrl from '@docusaurus/useBaseUrl'
```

warning This pattern is not available for now.

```
import Rater from '@theme/Rater'
```

Ease of Use
Extensibility
Performance
Security

GLUI

Pros:

- Framebuffer FTW
- Window events rigged

Cons:

- Broken on your machine

Description

The GLUI is a research pattern that we will use internally to test approaches using a GLUTIN window. We're not sure yet if it will make the final cut as a bona fide alternative to WebView, although early tests with transparent and multiwindow are exciting.

Diagram

```
graph TD
    A==>H
    H==>G
    A-->D
    D-->G
    subgraph GLUTIN
        G
    end
    subgraph RUST
        A
    end
    A[Binary]
    D(Framebuffer)
    G,GL Window
    H,Bootstrap
    class GLUTIN other
    class RUST rust
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {  
    allowlist: {  
        all: false, // all API endpoints are default false  
        // disable the api  
    },  
    window: {  
        glutin: true, // not yet normative  
        webview: false  
    }  
}
```

```
import Rater from '@theme/Rater' import useBaseUrl from '@docusaurus/useBaseUrl'
```

Ease of Use
Extensibility
Performance
Security

□ Hermit

Pros:

- Quick to make
- Smallest size

Cons:

- No remote resources
- No access to API

Description

The Hermit recipe is a pattern for ultimate application isolation where all logic is self-contained in the Window and the binary exists merely to bootstrap the Window. There is no communication back to Rust from the Window, there is no localhost server, and the Window has no access to any remote resources. The Hermit is great for interactive Kiosk Mode and standalone HTML-based games.

Diagram

```
graph LR
A==>H
H==>F
subgraph WEBVIEW
F
end
subgraph RUST
A
end
A[fa:fa-cog Binary ]
F[fa:fa-window-maximize Window]
H{Bootstrap}
class RUST rust
class WEBVIEW webview
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {
  allowlist: {
    all: false,           // disable and tree-shake all api functions
  }
}
```

```
import Rater from '@theme/Rater' import useBaseUrl from '@docusaurus/useBaseUrl'
```

Ease of Use
Extensibility
Performance
Security

Lockdown

Pros:

- Highest security rating
- Elegant and powerful

Cons:

- Rust skills required
- No remote resources

Description

The Lockdown recipe is a minimal usage of the [Bridge pattern](#), which only allows interaction between Rust and the Window via expiring JS Promise Closures that are injected into the Window by Rust and nulled as part of the callback.

Diagram

```
graph TD
    H==>F
    G-.->B
    B-->G
    subgraph WEBVIEW
        G-->F
    end
    subgraph RUST
        A-->B
        A==>H
    end
    A[Binary]
    B[API:Event]
    F[Window]
    G((Promise Closure))
    H{Bootstrap}
    class RUST rust
    class WEBVIEW webview
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {  
    allowlist: []  
} // all API endpoints are default false
```

[Go to TOC](#)

```
import useBaseUrl from '@docusaurus/useBaseUrl' import Rater from '@theme/Rater'
```

Ease of Use

Extensibility

Performance

Security

Multiwin

Pros:

- Windows can be spawned or destroyed at runtime
- Separation of concerns

Cons:

- Somewhat complex

Description

The Multiwin recipe will allow you to have multiple windows.

Diagram

```
graph LR
A==>H
H==>F
H==>G
subgraph WEBVIEW
F
end
subgraph WINIT
G
end
subgraph RUST
A
end
A[Binary]
F[Window]
G[Window]
H{Bootstrap}
class WINIT other
class RUST rust
class WEBVIEW webview
```

Configuration

Here's what you need to add to your tauri.conf.json file:

```
tauri: {
  allowlist: {}, // all API endpoints are default false
  windows: [
    title: Window1,
```

```
    label: main,  
}, {  
    title: Splash,  
    label: splashscreen  
}]  
}
```

Security

This guide seeks to explain the high-level concepts and security features at the core of Tauri's design that make you, your apps, and your users safer by default.

info Note

While we take every opportunity to help you harden your application - there are always underlying threats like BIOS attacks, memory rowhammering and other operating system vulnerabilities that are constantly being discovered and (in the best cases) responsibly disclosed.

Furthermore, there are many ways that development teams can cut corners and either leak sensitive information or leave doors wide open to any of a range of attacks. Security is a never-ending quest, and your users count on you to keep them safe.

Therefore, we highly recommend that you take some time to consider the security ramifications of everything that your application does, especially in the context of running on the semi-hostile platform of end-user devices.

If you need help or want a review, you are welcome to contact the Tauri team for security consultation.

Security Researchers

If you feel that there is a security concern or issue with anything in Tauri, please do not publicly comment on your findings. Instead, reach out directly to our security team: security@tauri.app

Although we do not currently have a budget for Security Bounties, in some cases, we will consider rewarding responsible disclosure with our limited resources.

No Server Required

Tauri enables you to construct an application that uses web technology for the user interface without requiring you to use a server to communicate with the backend. Even if you used advanced techniques of dynamic imports and offload work to the backend, no traffic can be sniffed on TCP ports or external processes - because they aren't there. This reduces not only the physical and virtual footprint of your final binary by a good deal, but it also reduces the surface area of potential attack vectors by removing them from the equation.

Language Features of Rust

By turning to the programming language renowned for its memory safety and speed, Tauri simply erases whole classes of conventional attacks. Use after free just isn't something that can happen with Tauri.

Dynamic Ahead of Time Compilation (AOT)

This compilation process happens several times during the bootstrapping phase of a Tauri app. Using our default dynamic Ahead of Time compiler, you can generate code references that are unique for every session and are still technically static code units.

Function Hardening

Functional ASLR

Functional address Space Layout Randomization techniques randomize function names at runtime and can implement OTP hashing, so no two sessions are ever the same. We propose a novel type of function naming at boot time and optionally after every execution. Using a UID for each function pointer prevents static attacks.

Kamikaze Function Injection

This advanced type of fASLR using the `EVENT` API endpoint is a promise wrapped in a closure (with randomized handle) that Rust inserts at runtime into the WebView, where its interface is locked within the promise resolution handler and is nulled after execution.

Bridge, don't serve

Instead of passing potentially unsafe functions, an event bridge can be used to pass messages and commands to named brokers at each respective side of the application.

One Time Pad Tokenization and Hashing

Hashing important messages with an OTP salt, you are able to encrypt messages between the user interface and the Rust backend. We are currently investigating the use of additional sources of entropy, such as the amazing [Infinite Noise TRNG](#).

System Features

Allowing API

You have the ability to pick and choose which API functions are available to the UI and to Rust. If they are not enabled, the code will not be shipped with your app, which reduces binary size and attack surface. They are opt-in, so you have to consciously choose to progressively enhance your application.

Content Security Policy Management

Preventing unauthorized code execution for websites has long since been resolved by using CSPs. Tauri can inject CSPs into the `index.html` of the user interface, and when using a localhost server, it will also send these headers to the UI or any other clients that connect with it.

Decompilation is Difficult

This means that your apps cannot be easily decompiled, as is the case with Electron ASAR files, which makes the process of reverse engineering your project much more time-intensive and requires specialist training.

Ecosystem

Build Pipelines and Artifact Authenticity

The process of releasing our source-code artifacts is highly automated yet mandates kickoff and review from real humans. Our current release strategy uses a combination of GitHub Actions and IOTA Tangle publication

Resilient PR and Approval Processes

Our WG-TECH reviews code changes, tags PRs with scope, and makes sure that everything stays up to date. And when its time to publish a new version, one of the maintainers tags a new release on dev, which:

- Validates core
- Runs smoke tests
- Audits security for crates and npm
- Generates changelogs
- Creates artifacts
- Publishes checksums to IOTA
- Creates a draft release

Then the maintainer reviews the release notes, edits if necessary, and a new release is forged.

Future Work

Signed Binaries

Because the entire project is shipped within a monolithic binary, code can be signed for all distributables. (Currently using external tooling, but we are actively working on making the bundler a one-stop-shop.) This makes it virtually impossible for hackers to change an installed Application without the operating system noticing. [Reference](#)

Post-Binary Analysis

Use industrial-grade pentester-tooling (via our forthcoming Tauri-Frida GUI) to discover and fix security weaknesses in your final binaries.

Post-Binary Enhancement

After the build is before the delivery and Tauri will provide you with tools never seen before. Stay tuned!

Audits

We are currently in the process of our first external audit. When complete, we will publish the results here.

[Go to TOC](#)

Benchmarks

```
import Chart, { fetchData } from '@theme/BenchmarkChart'
```

```
export var data = fetchData()
```

All benchmarks run on GitHub Actions using the `ubuntu-latest` matrix. Various metrics are measured by the following applications:

Tauri	Wry	Electron
<code>tauri_cpu_intensive</code>	<code>wry_cpu_intensive</code>	<code>electron_cpu_intensive</code>
<code>tauri_hello_world</code>	<code>wry_hello_world</code>	<code>electron_hello_world</code>
<code>tauri_3mb_transfer</code>	<code>wry_custom_protocol</code>	<code>electron_3mb_transfer</code>

The CPU intensive benchmark measures how much time it takes to calculate all the prime numbers under a certain value without blocking the UI and reporting how many have been found so far using web workers.

Execution Time

How much time in total it takes to initialize the application and wait for the `DOMContentLoaded` event. This uses `hyperfine` under the hood and runs 3 warm-up sequence first, then 10 sequences to calculate the average execution time.

Binary Size

All binaries are compiled in release mode.

Memory Usage

Uses `mprof` to get the max memory usage during execution. Smaller is better.

Thread Count

How many threads the application uses. Smaller is better.

Syscall Count

How many total syscalls are performed when executing a given application. Smaller is better.

Dependencies

Configuration Files

Since Tauri is a toolkit for building applications there can be many files to configure project settings. Some common files that you may run across are `tauri.conf.json`, `package.json` and `Cargo.toml`. We briefly explain each on this page to help point you in the right direction for which file to modify.

Tauri Config

The file can be either `tauri.conf.json`, `tauri.conf.json5`, or `Tauri.toml`. The default is `tauri.conf.json`. See the note below for more information.

This is the file used by the Tauri process. You can define build settings (such as the [command run before tauri build](#) or [tauri dev](#)), set the [name and version of your app](#), [control the Tauri process](#), and configure any plugin settings. You can find all of the options in the [tauri.conf.json API reference](#).

The default Tauri config format is `.json`. The `.json5` or `.toml` format can be enabled by adding the `config-json5` or `config-toml` feature flag (respectively) to the `tauri` and `tauri-build` dependencies in `Cargo.toml`. Note that the `.toml` format is only available from Tauri 1.1 and above.

`Cargo.toml`

```
[build-dependencies]
// highlight-next-line
tauri-build = { version = 1.0.0, features = [ config-json5 ] }

[dependencies]
serde_json = 1.0
serde = { version = 1.0, features = [derive] }
// highlight-next-line
tauri = { version = 1.0.0, features = [ api-all, config-json5 ] }
```

The structure and values are the same across all formats, however, the formatting should be consistent with the respective file's format.

Cargo.toml

Cargo's manifest file is used to declare Rust crates your app depends on, metadata about your app, and other Rust-related features. If you do not intend to do backend development using Rust for your app then you may not be modifying it much, but it's important to know that it exists and what it does.

Below is an example of a barebones `Cargo.toml` file for a Tauri project:

`Cargo.toml`

```
[package]
name = app
version = 0.1.0
description = A Tauri App
authors = [you]
license =
repository =
default-run = app
edition = 2021
rust-version = 1.57

[build-dependencies]
tauri-build = { version = 1.0.0 }

[dependencies]
serde_json = 1.0
serde = { version = 1.0, features = [derive] }
tauri = { version = 1.0.0, features = [api-all] }

[features]
# by default Tauri runs in production mode
# when `tauri dev` runs it is executed with `cargo run --no-default-features` if
`devPath` is an URL
default = [custom-protocol]
# this feature is used for production builds where `devPath` points to the
filesystem
# DO NOT remove this
custom-protocol = [tauri/custom-protocol]
```

The most important parts to take note of are the `tauri-build` and `tauri` dependencies. Generally, they must both be on the latest minor versions as the Tauri CLI, but this is not strictly required. If you encounter issues while trying to run your app you should check that any Tauri versions (`tauri` and `tauri-cli`) are on the latest versions for their respective minor releases.

Cargo version numbers use [Semantic Versioning](#). Running `cargo update` will pull the latest available Semver-compatible versions of all dependencies. For example, if you specify `1.0.0` as the version for `tauri-build`, Cargo will detect and download version `1.0.4` because it is the latest Semver-compatible version available. Tauri will update the major version number whenever a breaking change is introduced, meaning you should always be capable of safely upgrading to the latest minor and patch versions without fear of your code breaking.

If you want to use a specific crate version you can use exact versions instead by prepending `=` to the version number of the dependency:

```
tauri-build = { version = =1.0.0 }
```

An additional thing to take note of is the `features=[]` portion of the `tauri` dependency. Running `tauri dev` and `tauri build` will automatically manage which features need to be enabled in your project based on the `allowlist` properties you set in `tauri.conf.json`.

When you build your application a `Cargo.lock` file is produced. This file is used primarily for ensuring that the same dependencies are used across machines during development (similar to `yarn.lock` or `package-lock.json` in Node.js). Since you are developing a Tauri app, this file should be committed to your source repository (only Rust libraries should omit committing this file).

To learn more about `Cargo.toml` you can read more in the [official documentation](#).

package.json

This is the package file used by Node.js. If the frontend of a Tauri app is developed using Node.js-based technologies (such as `npm`, `yarn`, or `pnpm`) this file is used to configure the frontend dependencies and scripts.

An example of a barebones `package.json` file for a Tauri project might look a little something like this:

package.json

```
{  
  scripts: {  
    dev: command-for-your-framework,  
    tauri: tauri  
  },  
  dependencies: {  
    @tauri-apps/api: ^1.0,  
    @tauri-apps/cli: ^1.0  
  }  
}
```

It's common to use the `scripts` section to store the command used to launch the frontend used by your Tauri application. The above file specifies the `dev` command that you can run using `yarn dev` or `npm run dev` to start the frontend framework.

The dependencies object specifies which dependencies Node.js should download when you run either `yarn` or `npm install` (in this case the Tauri CLI and API).

In addition to the `package.json` file you may see either a `yarn.lock` file or a `package-lock.json` file. These files assist in ensuring that when you download the dependencies later you'll get the exact same versions that you have used during development (similar to `Cargo.lock` in Rust).

To learn more about `package.json` you can read more in the [official documentation](#).

Development Security

Whether you like it or not, today's applications live in operating systems that can be (and regularly are) compromised by any number of attacks. When your insecure application is a gateway for such lateral movement into the operating system, you are contributing to the tools that professional hackers have at their disposal. Don't be a tool.

This is why we have taken every opportunity to help you secure your application, prevent undesired access to system-level interfaces, and manufacture bullet-proof applications. Your users assume you are following best practices. We make that easy, but you should still be aware of the information below.

Security Is A Community Responsibility

It is important to remember that the security of your Tauri application is the sum of the overall security of Tauri itself, all Rust and npm dependencies, your code, and the devices that run the final application. The Tauri Team does their best to do their part, the security community does its part, and you too should follow a few important best practices.

Keep Your Application Up-To-Date

When releasing your app into the wild, you are also shipping a bundle that has Tauri in it. Vulnerabilities affecting Tauri may impact the security of your application. By updating Tauri to the latest version, you ensure that critical vulnerabilities are already patched and cannot be exploited in your application. Also be sure to keep your compiler (rustc) and transpilers (nodejs) up to date, because there are often security issues that are resolved.

Evaluate Your Dependencies

While npm and Crates.io provide many convenient packages, it is your responsibility to choose trustworthy third-party libraries - or rewrite them in Rust. If you do use outdated libraries which are affected by known vulnerabilities or are unmaintained, your application security and good night's sleep could be in jeopardy. Use tooling like npm audit and cargo audit to automate this process, and lean on the security community's important work.

Adopt More Secure Coding Practices

The first line of defense for your application is your own code. Although Tauri can protect you from common web vulnerabilities, such as Cross-Site Scripting based Remote Code Execution, improper configurations can have a security impact. Even if this were not the case, it is highly recommended to adopt secure software development best practices and perform security testing. We detail what this means in the next section.

Educate Your Users

True security means that unexpected behaviour cannot happen. So in a sense, being more secure means having the peace of mind of knowing that ONLY those things that you want to happen can happen. In the real world, though, this is a utopian dream. However, by removing as many vectors as possible and building on a solid foundation, your choice of Tauri is a signal to your users that you care about them, their safety, and their devices.

Threat Models

Tauri applications are composed of many pieces at different points of the lifecycle. Here we describe classical threats and what you SHOULD do about them.

Upstream Threats

Tauri is a direct dependency on your project, and we maintain strict authorial control of commits, reviews, pull requests, and releases. We do our best to maintain up-to-date dependencies and take action to either update or fork and fix. Other projects may not be so well maintained, and may not even have ever been audited. Please consider their health when integrating them, otherwise, you may have adopted architectural debt without even knowing it.

Development Threats

We assume that you, the developer, care for your development environment. It is on you to make sure that your operating system, build toolchains, and associated dependencies are kept up to date.

A genuine risk all of us face is what is known as supply-chain attacks, which are usually considered to be attacks on direct dependencies of your project. However, a growing class of attacks in the wild directly target development machines, and you would be well off to address this head-on.

One practice that we highly recommend, is to only ever consume critical dependencies from git using hash revisions at best or named tags as second best. This holds for Rust as well as the Node ecosystem. Also, consider requiring all contributors to sign their commits and protect Git branches and pipelines.

Buildtime Threats

Modern organisations use CI/CD to manufacture binary artifacts. At Tauri, we even provide a GitHub Workflow for building on multiple platforms. If you create your own CI/CD and depend on third-party tooling, be wary of actions whose versions you have not explicitly pinned.

You should sign your binaries for the platform you are shipping to, and while this can be complicated and somewhat costly to setup, end users expect that your app is verifiably from you.

Runtime Threats

We assume the webview is insecure, which has led Tauri to implement several protections regarding webview access to system APIs in the context of loading untrusted userland content.

You can read more in detail below, but using the CSP will lockdown types of communication that the Webview can undertake. Furthermore, [Context Isolation](#) prevents untrusted content or scripts from accessing the API within the Webview.

And please, whatever you do, **DO NOT** trust the results of cryptography using private keys in the Webview. Rust is there for a reason.

Updater Threats

We have done our best to make shipping hot updates to the app as straightforward and secure as possible. However, all bets are off if you lose control of the manifest server, the build server, or the binary hosting service. If you build your own system, consult a professional OPS architect and build it properly.

Secure Content Loading

Tauri restricts the [Content Security Policy](#) (CSP) of your HTML pages. Local scripts are hashed, styles and external scripts are referenced using a cryptographic nonce, which prevents unallowed content from being loaded.

warning Avoid loading remote content such as scripts served over a CDN as they introduce an attack vector. But any untrusted file can introduce new and subtle attack vectors.

The CSP protection is only enabled if `[tauri > security > csp]` is set on the Tauri configuration file. You should make it as restricted as possible, only allowing the webview to load assets from hosts you trust, and preferably own. At compile time, Tauri appends its nonces and hashes to the relevant CSP attributes automatically, so you only need to worry about what is unique to your application.

See `script-src`, `style-src` and [CSP Sources](#) for more information about this protection.

Webview Versions

WebView2 (Windows)

Tauri uses WebView2 which is based on Microsoft Edge and therefore Chromium. WebView2 can update itself, you are guaranteed a relatively recent chromium build on all Windows targets.

WebView2 is supported on Windows 7 and newer and comes preinstalled on Windows 11. On versions older than Windows 11 the installer generated by Tauri takes care of ensuring WebView2 is installed on the system.

WebKit (macOS, iOS, & Linux)

Tauri uses WebKit on macOS (through `WKWebView`) and Linux (through `webkit2gtk`).

Interpreting WebKit Version Numbers

Wekit version numbers are quite complicated, so below is some helpful information to understand them.

WebKit version numbers are made up of 5 segments and a numeric prefix indicating which OS WebKit is built for:

```
$(SYSTEM_VERSION_PREFIX)$(MAJOR_VERSION).$(MINOR_VERSION).$(TINY_VERSION).$(MICRO_VERSION).$(NANO_VERSION)
```

The numeric prefix is called the `SYSTEM_VERSION_PREFIX` and seems to be only present for macOS and iOS builds (not for Linux). Furthermore, if the last two segments are both `0` they can be omitted (so a version like `613.2.7.0.0` would be referred to as `613.2.7`).

As an example, the WebKit version shipped with Safari 15.5 on macOS Monterey (12.x) has the version number `17613.2.7.1.8`. You can interpret it like this:

- `SYSTEM_VERSION_PREFIX` : 17
- `MAJOR_VERSION` : 613
- `MINOR_VERSION` : 2
- `TINY_VERSION` : 7
- `MICRO_VERSION` : 1
- `NANO_VERSION` : 8

Here is what the `SYSTEM_VERSION_PREFIX` values map to:

macOS version	SYSTEM_VERSION_PREFIX
sdk=iphone*	8

macOS version	SYSTEM_VERSION_PREFIX
13.0	18
12.0	17
11.0	16
10.15	15
10.14	14
10.13	13
10.12	12
10.11	11

macOS & iOS

On macOS, Tauri uses the webview that comes preinstalled with macOS since version 10.10 (Yosemite). It is considered a core component and is therefore updated with the regular OS updates. This means unsupported macOS versions **do not** receive WebKit updates.

To find the WebKit version used by `WKWebView` on your version of macOS you can use this command in the terminal:

```
awk '/CFBundleVersion/{getline;gsub(/<[^>]*>/,");print}'  
/System/Library/Frameworks/WebKit.framework/Resources/Info.plist
```

WebKit Versions in Safari

The table below maps an OS version to the corresponding WebKit Safari versions so that you can use sites like [caniuse](#) to figure out if a specific web platform feature is supported.

OS Name	OS Version	WebKit Version	Safari Version	Notes
Ventura	13.0.1	614.2.9.1.12	13.0 16.1	Verified on a 2020 M1 13 MacBook Pro
Monterey	12.6	613.3.9.1.5	12.5 15.6	Verified on a 2020 M1 13 MacBook Pro
M1 13 MacBook Pro	12.5.1	613.3.9.1.16	15.6.1	Verified on a 2020 M1 13 MacBook Pro
M1 13 MacBook Pro	12.3.1	613.2.7.1.8	12.4 15.5	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.3	613.1.17.1.13	12.4 15.5	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.2	613.1.17.1.6	12.3 15.4	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.2	612.4.9.1.8	12.2 15.4	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.2	612.4.9.1.5	12.2 15.3	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.1.1	612.3.6.1.6	12.1 15.2	Verified on a 2020 M1 13 MacBook Pro
Big Sur	12.1	612.3.6.1.6	12.1 15.2	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6.7	612.2.9.1.20	11.6.7 11.6.6	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6.6	612.2.9.1.20	11.6.6 11.6.5	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6.5	612.2.9.1.20	11.6.5 11.6.4	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6.2	612.2.9.1.20	11.6.2 11.6.1	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6.1	612.2.9.1.20	11.6.1 11.6.0	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.6	612.2.9.1.20	11.6 11.5.2	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.5.2	612.2.9.1.20	11.5.2 611.3.10.1.6	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.5	612.2.9.1.20	11.5 611.3.10.1.6	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.4	612.2.9.1.20	11.4 611.2.7.1.4	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.4	612.2.9.1.20	11.4 14.1.1	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.3.1	612.2.9.1.20	11.3.1 11.3	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.3	612.2.9.1.20	11.3 611.1.21.16.1.3	Verified on a 2020 M1 13 MacBook Pro
Big Sur	11.2.3	612.2.9.1.20	11.2.3 14.1	Verified on a 2020 M1 13 MacBook Pro
iMac	24	611.1.21.16.1.3	14.1 24	iMac received a special WebKit version
iMac	M1	611.1.21.1.12	14.0.3 14.0.2	iMac received a special WebKit version
iPhone	11.0.1	610.2.11.51.8	11.0 11.0.1	iPhone received a special WebKit version
Catalina	10.15.7	610.2.11.1.3	14.0.1 14.0.2	Catalina received a special WebKit version
Security Update 2022-004	10.15.7	609.4.1.1.1	14.0.1 14.0.2	Security Update 2022-004 received a special WebKit version
10.15.7	10.15.6	609.4.1.1.1	14.0.1 14.0.2	10.15.7 received a special WebKit version
10.15.4	10.15.5	609.2.9.1.2	13.1.1 13.1.2	10.15.4 received a special WebKit version
10.15.4	10.15.6	609.3.5.1.3	13.1.2 13.1.3	10.15.4 received a special WebKit version

609.1.20.111.8 | 13.1 | | | 10.15.3 | [608.5.11](#) | 13.0.5 | | | 10.15.2 | [608.4.9.1.3](#) | 13.0.4 | | | |
 10.15.1 | [608.3.10.1.4](#) | 13.0.3 | Verified on a 2014 15 MacBook Pro | | | 10.15 | [608.2.30.1.1](#) | 13.0.2 | | |
 | Mojave | 10.14.6 | [608.1.49](#) | 13.0 | | | 10.14.4 | [607.1.40.1.5](#) | 12.1 | | | 10.14.3 | [606.4.5](#) | 12.0.3
 | | | 10.14.2 | [606.3.4](#) | 12.0.2 | | | 10.14.1 | [606.2.104.1.1](#) | 12.0.1 | | | 10.14 | [606.2.11](#) | 12.0 | | |
 | High Sierra | 10.13.6 | [605.3.8](#) | 11.1.2 | | | 10.13.5 | [605.2.8](#) | 11.1.1 | | | 10.13.4 Security Update
 2018-001 | [605.1.33.1.4](#) | 11.1 | | | 10.13.4 | [605.1.33.1.2](#) | 11.1 | | | 10.13.3 | [604.5.6](#) | 11.0.3 | | |
 | 10.13.2 Supplemental Update | [604.4.7.1.6](#) | 11.0.2 | 27 iMac Pro received a special WebKit version
[604.4.7.10.6](#) | | | 10.13.2 | [604.4.7.1.3](#) | 11.0.2 | 27 iMac Pro received a special WebKit version
[604.4.7.10.4](#) | | | 10.13.1 | [604.3.5](#) | 11.0.1 | | | 10.13 | [604.1.38.1.6](#) | 11.0 | |

Linux

The diverse nature of the Linux ecosystem means it is very hard to compile accurate information about WebKitGTK on the various distros. The table below is a very incomplete list of the most commonly used distributions and their WebKit versions. You should always check your distro's repositories for up-to-date information.

Distro	webkitgtk Version	WebKit Version	Safari Equivalent
Debian 11 (with update), Ubuntu 20.04 (with update), Ubuntu 22.04	2.36	614.1.6	TP 140 (16.0)
Debian 10 (with update)	2.34	613.1.1	15.4
Debian 11, Ubuntu 18.04 (with update), centos 8 (non-stream)	2.32	612.1.6	15.0
Ubuntu 20.04	2.28	610.1.1	14.0
Debian 9 (with backport), Debian 10	2.24	608.1.6	13.0
Ubuntu 18.04	2.20	606.1.4	12.0

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- Tauri - English
- GitHub source: [tauri-apps/tauri-docs](https://github.com/tauri-apps/tauri-docs)
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>