Windows Installer

Tauri applications for Windows are either distributed as Microsoft Installers (`.msi` files) using the WiX Toolset v3 or starting with Tauri v1.3 as setup executables (`-setup.exe` files) using NSIS. 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. Cross-compilation for NSIS installers is experimental and being worked on.

This guide provides information about available customization options for the installer.

To build and bundle your whole Tauri application into a single installer simply run the following command:

npm Yarn pnpm Cargo

```
$ cargo tauri build
```

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:

```
PS C:\> 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:

```
PS C:\> tauri build --target aarch64-pc-windows-msvc
```

> **INFO**
>
> Only the NSIS target supports ARM64 targets, so if you configured tauri to compile all bundle types you may want to change the above command to `tauri build --target aarch64-pc-windows-msvc --bundles nsis` to only build the NSIS installer.
>
> Note that the installer itself will still be x86 running on the ARM machine via emulation. The app itself will be a native ARM64 binary.

Supporting Windows 7

By default, the Microsoft Installer (`.msi`) 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). The NSIS based installer (`-setup.exe`) also supports the `downloadBootstrapper` mode on Windows 7.

Additionally, to use the Notification API in Windows 7, you need to enable the `windows7-compat` Cargo feature:

Cargo.toml

```toml
[dependencies]
tauri = { version = "1", features = [ "windows7-compat" ] }
```

FIPS Compliance

If your system requires the MSI bundle to be FIPS compliant you can set the `TAURI_FIPS_COMPLIANT` environment variable to `true` before running `tauri build`. In PowerShell you can set it for the current terminal session like this:

```
PS C:\> $env:TAURI_FIPS_COMPLIANT="true"
```

WebView2 Installation Options

The installers by default download 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	`Default` Results in a smaller installer size, but is not recommended for Windows 7 deployment via `.msi` files.
`embedBootstrapper`	Yes	~1.8MB	Better support on Windows 7 for `.msi` installers.
`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 runs it. Requires an internet connection but results in a smaller installer size. This is not recommended if you're going to be distributing to Windows 7 via `.msi` installers.

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:

```
PS C:\> Expand .\Microsoft.WebView2.FixedVersionRuntime.98.0.1108.50.x64.cab -F:* ./src-tauri
```

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.

> **DANGER**
>
> 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 WiX Installer Template

The `.msi` Windows Installer package is built using the WiX Toolset v3. Currently, apart from pre-defined configurations, 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 webview-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.

Customizing the NSIS Installer Template

The NSIS Installer's `.nsi` script defined by Tauri is configured to work for the common use case of simple webview-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.nsis.template` on Tauri v1.4 and above.

Internationalization

The NSIS Installer is a multi-language installer, which means you always have a single installer which contains all the selected translations. You can specify which languages to include using the `tauri.bundle.windows.nsis.languages` property. A list of languages supported by NSIS is available in the NSIS GitHub project. There are a few Tauri-specific translations required, so if you see untranslated texts feel free to open a feature request in Tauri's main repo. Starting with v1.4 you can also provide custom translation files.

The WiX 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 a WiX 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 a WiX 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 WiX 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>
```

> **NOTE**
>
> 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.