Context Isolation

What is it?

Context Isolation is a feature that ensures that both your `preload` scripts and Electron's internal logic run in a separate context to the website you load in a `webContents`. This is important for security purposes as it helps prevent the website from accessing Electron internals or the powerful APIs your preload script has access to.

This means that the `window` object that your preload script has access to is actually a **different** object than the website would have access to. For example, if you set `window.hello = 'wave'` in your preload script and context isolation is enabled, `window.hello` will be undefined if the website tries to access it.

Context isolation has been enabled by default since Electron 12, and it is a recommended security setting for *all applications*.

Migration

> *Without context isolation, I used to provide APIs from my preload script using* `window.X = apiObject`*. Now what?*

Before: context isolation disabled

Exposing APIs from your preload script to a loaded website in the renderer process is a common use-case. With context isolation disabled, your preload script would share a common global `window` object with the renderer. You could then attach arbitrary properties to a preload script:

```
// preload with contextIsolation disabled
window.myAPI = {
  doAThing: () => {}
}
```

The `doAThing()` function could then be used directly in the renderer process:

```
// use the exposed API in the renderer
window.myAPI.doAThing()
```

After: context isolation enabled

There is a dedicated module in Electron to help you do this in a painless way. The `contextBridge` module can be used to **safely** expose APIs from your preload script's isolated context to the context the website is running in. The API will also be accessible from the website on `window.myAPI` just like it was before.

```
// preload with contextIsolation enabled
const { contextBridge } = require('electron')

contextBridge.exposeInMainWorld('myAPI', {
  doAThing: () => {}
})
```

```
// use the exposed API in the renderer
window.myAPI.doAThing()
```

Please read the `contextBridge` documentation linked above to fully understand its limitations. For instance, you can't send custom prototypes or symbols over the bridge.

Security considerations

Just enabling `contextIsolation` and using `contextBridge` does not automatically mean that everything you do is safe. For instance, this code is **unsafe**.

```
// ❌ Bad code
contextBridge.exposeInMainWorld('myAPI', {
  send: ipcRenderer.send
})
```

It directly exposes a powerful API without any kind of argument filtering. This would allow any website to send arbitrary IPC messages, which you do not want to be possible. The correct way to expose IPC-based APIs would instead be to provide one method per IPC message.

```
// ✅ Good code
contextBridge.exposeInMainWorld('myAPI', {
  loadPreferences: () => ipcRenderer.invoke('load-prefs')
})
```

Usage with TypeScript

If you're building your Electron app with TypeScript, you'll want to add types to your APIs exposed over the context bridge. The renderer's `window` object won't have the correct typings unless you extend the types with a declaration file.

For example, given this `preload.ts` script:

```
contextBridge.exposeInMainWorld('electronAPI', {
  loadPreferences: () => ipcRenderer.invoke('load-prefs')
})
```

You can create a `renderer.d.ts` declaration file and globally augment the `Window` interface:

```
export interface IElectronAPI {
  loadPreferences: () => Promise<void>,
}

declare global {
  interface Window {
    electronAPI: IElectronAPI
  }
}
```

Doing so will ensure that the TypeScript compiler will know about the `electronAPI` property on your global `window` object when writing scripts in your renderer process:

```
window.electronAPI.loadPreferences()
```