contextBridge

History

> Create a safe, bi-directional, synchronous bridge across isolated contexts

Process: **Renderer**

An example of exposing an API to a renderer from an isolated preload script is given below:

```
// Preload (Isolated World)
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld(
  'electron',
  {
    doThing: () => ipcRenderer.send('do-a-thing')
  }
)
```

```
// Renderer (Main World)

window.electron.doThing()
```

Glossary

Main World

The "Main World" is the JavaScript context that your main renderer code runs in. By default, the page you load in your renderer executes code in this world.

Isolated World

When `contextIsolation` is enabled in your `webPreferences` (this is the default behavior since Electron 12.0.0), your `preload` scripts run in an "Isolated World". You can read more about context isolation and what it affects in the **security** docs.

Methods

The `contextBridge` module has the following methods:

`contextBridge.exposeInMainWorld(apiKey, api)`

- `apiKey` string - The key to inject the API onto `window` with. The API will be accessible on `window[apiKey]`.
- `api` any - Your API, more information on what this API can be and how it works is available below.

`contextBridge.exposeInIsolatedWorld(worldId, apiKey, api)`

- `worldId` Integer - The ID of the world to inject the API into. `0` is the default world, `999` is the world used by Electron's `contextIsolation` feature. Using 999 would expose the object for preload context. We recommend using 1000+ while creating isolated world.
- `apiKey` string - The key to inject the API onto `window` with. The API will be accessible on `window[apiKey]`.
- `api` any - Your API, more information on what this API can be and how it works is available below.

`contextBridge.executeInMainWorld(executionScript)` *Experimental*

- `executionScript` Object
  - `func` (...args: any[]) => any - A JavaScript function to execute. This function will be serialized which means that any bound parameters and execution context will be lost.
  - `args` any[] (optional) - An array of arguments to pass to the provided function. These arguments will be copied between worlds in accordance with **the table of supported types.**

Returns `any` - A copy of the resulting value from executing the function in the main world. **Refer to the table** on how values are copied between worlds.

Usage

API

The `api` provided to **`exposeInMainWorld`** must be a `Function`, `string`, `number`, `Array`, `boolean`, or an object whose keys are strings and values are a `Function`, `string`, `number`, `Array`, `boolean`, or another nested object that meets the same conditions.

`Function` values are proxied to the other context and all other values are **copied** and **frozen**. Any data / primitives sent in the API become immutable and updates on either side of the bridge do not result in an update on the other side.

An example of a complex API is shown below:

```
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld(
  'electron',
  {
    doThing: () => ipcRenderer.send('do-a-thing'),
    myPromises: [Promise.resolve(), Promise.reject(new Error('whoops'))],
    anAsyncFunction: async () => 123,
    data: {
      myFlags: ['a', 'b', 'c'],
      bootTime: 1234
    },
    nestedAPI: {
      evenDeeper: {
        youCanDoThisAsMuchAsYouWant: {
          fn: () => ({
            returnData: 123
          })
        }
      }
    }
```

```
  }
)
```

An example of `exposeInIsolatedWorld` is shown below:

```
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInIsolatedWorld(
  1004,
  'electron',
  {
    doThing: () => ipcRenderer.send('do-a-thing')
  }
)
```

```
// Renderer (In isolated world id1004)

window.electron.doThing()
```

API Functions

`Function` values that you bind through the `contextBridge` are proxied through Electron to ensure that contexts remain isolated. This results in some key limitations that we've outlined below.

Parameter / Error / Return Type support

Because parameters, errors and return values are **copied** when they are sent over the bridge, there are only certain types that can be used. At a high level, if the type you want to use can be serialized and deserialized into the same object it will work. A table of type support has been included below for completeness:

Type	Complexity	Parameter Support	Return Value Support	Limitations
`string`	Simple	✅	✅	N/A

Type	Complexity	Parameter Support	Return Value Support	Limitations
`number`	Simple	✅	✅	N/A
`boolean`	Simple	✅	✅	N/A
`Object`	Complex	✅	✅	Keys must be supported using only "Simple" types in this table. Values must be supported in this table. Prototype modifications are dropped. Sending custom classes will copy values but not the prototype.
`Array`	Complex	✅	✅	Same limitations as the `Object` type
`Error`	Complex	✅	✅	Errors that are thrown are also copied, this can result in the message and stack trace of the error changing slightly due to being thrown in a different context, and any custom properties on the Error object **will be lost**
`Promise`	Complex	✅	✅	N/A
`Function`	Complex	✅	✅	Prototype modifications are dropped. Sending classes or constructors will not work.
Cloneable Types	Simple	✅	✅	See the linked document on cloneable types
`Element`	Complex	✅	✅	Prototype modifications are dropped. Sending custom elements

Type	Complexity	Parameter Support	Return Value Support	Limitations
				will not work.
`Blob`	Complex	✅	✅	N/A
`Symbol`	N/A	❌	❌	Symbols cannot be copied across contexts so they are dropped

If the type you care about is not in the above table, it is probably not supported.

Exposing ipcRenderer

Attempting to send the entire `ipcRenderer` module as an object over the `contextBridge` will result in an empty object on the receiving side of the bridge. Sending over `ipcRenderer` in full can let any code send any message, which is a security footgun. To interact through `ipcRenderer`, provide a safe wrapper like below:

```
// Preload (Isolated World)
contextBridge.exposeInMainWorld('electron', {
  onMyEventName: (callback) => ipcRenderer.on('MyEventName', (e, ...args) =>
callback(args))
})
```

```
// Renderer (Main World)
window.electron.onMyEventName(data => { /* ... */ })
```

Exposing Node Global Symbols

The `contextBridge` can be used by the preload script to give your renderer access to Node APIs. The table of supported types described above also applies to Node APIs that you expose through `contextBridge`. Please note that many Node APIs grant access to local system resources. Be very cautious about which globals and APIs you expose to untrusted remote content.

```
const { contextBridge } = require('electron')
const crypto = require('node:crypto')
contextBridge.exposeInMainWorld('nodeCrypto', {
  sha256sum (data) {
    const hash = crypto.createHash('sha256')
    hash.update(data)
    return hash.digest('hex')
  }
})
```

Edit this page