session

Manage browser sessions, cookies, cache, proxy settings, etc.

Process: Main

The `session` module can be used to create new `Session` objects.

You can also access the `session` of existing pages by using the `session` property of `WebContents`, or from the `session` module.

```
const { BrowserWindow } = require('electron')

const win = new BrowserWindow({ width: 800, height: 600 })
win.loadURL('http://github.com')

const ses = win.webContents.session
console.log(ses.getUserAgent())
```

Methods

The `session` module has the following methods:

`session.fromPartition(partition[, options])`

- `partition` string
- `options` Object (optional)
  - `cache` boolean - Whether to enable cache.

Returns `Session` - A session instance from `partition` string. When there is an existing `Session` with the same `partition`, it will be returned; otherwise a new `Session` instance will be created with `options`.

If `partition` starts with `persist:`, the page will use a persistent session available to all pages in the app with the same `partition`. if there is no `persist:` prefix, the page will use an in-memory session. If the `partition` is empty then default session of the app will be returned.

To create a `Session` with `options`, you have to ensure the `Session` with the `partition` has never been used before. There is no way to change the `options` of an existing `Session` object.

Properties

The `session` module has the following properties:

`session.defaultSession`

A `Session` object, the default session object of the app.

Class: Session

Get and set properties of a session.

Process: Main *This class is not exported from the `'electron'` module. It is only available as a return value of other methods in the Electron API.*

You can create a `Session` object in the `session` module:

```
const { session } = require('electron')
const ses = session.fromPartition('persist:name')
console.log(ses.getUserAgent())
```

Instance Events

The following events are available on instances of `Session`:

Event: 'will-download' Returns:

- `event` Event
- `item` DownloadItem
- `webContents` WebContents

Emitted when Electron is about to download `item` in `webContents`.

Calling `event.preventDefault()` will cancel the download and `item` will not be available from next tick of the process.

```
const { session } = require('electron')
session.defaultSession.on('will-download', (event, item, webContents) => {
  event.preventDefault()
  require('got')(item.getURL()).then((response) => {
    require('fs').writeFileSync('/somewhere', response.body)
  })
})
```

Event: 'extension-loaded' Returns:

- `event` Event
- `extension` Extension

Emitted after an extension is loaded. This occurs whenever an extension is added to the "enabled" set of extensions. This includes:

- Extensions being loaded from `Session.loadExtension`.
- Extensions being reloaded:
  - from a crash.
  - if the extension requested it (`chrome.runtime.reload()`).

Event: 'extension-unloaded' Returns:

- `event` Event
- `extension` Extension

Emitted after an extension is unloaded. This occurs when `Session.removeExtension` is called.

Event: 'extension-ready' Returns:

- `event` Event
- `extension` Extension

Emitted after an extension is loaded and all necessary browser state is initialized to support the start of the extension's background page.

Event: 'preconnect' Returns:

- `event` Event
- `preconnectUrl` string - The URL being requested for preconnection by the renderer.
- `allowCredentials` boolean - True if the renderer is requesting that the connection include credentials (see the spec for more details.)

Emitted when a render process requests preconnection to a URL, generally due to a resource hint.

Event: 'spellcheck-dictionary-initialized' Returns:

- `event` Event
- `languageCode` string - The language code of the dictionary file

Emitted when a hunspell dictionary file has been successfully initialized. This occurs after the file has been downloaded.

Event: 'spellcheck-dictionary-download-begin' Returns:

- `event` Event
- `languageCode` string - The language code of the dictionary file

Emitted when a hunspell dictionary file starts downloading

Event: 'spellcheck-dictionary-download-success' Returns:

- `event` Event
- `languageCode` string - The language code of the dictionary file

Emitted when a hunspell dictionary file has been successfully downloaded

Event: 'spellcheck-dictionary-download-failure' Returns:

- `event` Event
- `languageCode` string - The language code of the dictionary file

Emitted when a hunspell dictionary file download fails. For details on the failure you should collect a netlog and inspect the download request.

Event: 'select-hid-device' Returns:

- `event` Event
- `details` Object
  - `deviceList` HIDDevice[]
  - `frame` WebFrameMain
- `callback` Function
  - `deviceId` string | null (optional)

Emitted when a HID device needs to be selected when a call to `navigator.hid.requestDevice` is made. `callback` should be called with `deviceId` to be selected; passing no arguments to `callback` will cancel the request. Additionally, permissioning on `navigator.hid` can be further managed by using ses.setPermissionCheckHandler(handler) and ses.setDevicePermissionHandler(handler)'.

```
const { app, BrowserWindow } = require('electron')

let win = null

app.whenReady().then(() => {
  win = new BrowserWindow()

  win.webContents.session.setPermissionCheckHandler((webContents, permission, requestingOrig
    if (permission === 'hid') {
      // Add logic here to determine if permission should be given to allow HID selection
      return true
    }
    return false
  })

  // Optionally, retrieve previously persisted devices from a persistent store
  const grantedDevices = fetchGrantedDevices()

  win.webContents.session.setDevicePermissionHandler((details) => {
    if (new URL(details.origin).hostname === 'some-host' && details.deviceType === 'hid') {
      if (details.device.vendorId === 123 && details.device.productId === 345) {
        // Always allow this type of device (this allows skipping the call to `navigator.hi
        return true
      }
```

```
    // Search through the list of devices that have previously been granted permission
    return grantedDevices.some((grantedDevice) => {
      return grantedDevice.vendorId === details.device.vendorId &&
            grantedDevice.productId === details.device.productId &&
            grantedDevice.serialNumber && grantedDevice.serialNumber === details.device.se
    })
  }
  return false
})

win.webContents.session.on('select-hid-device', (event, details, callback) => {
  event.preventDefault()
  const selectedDevice = details.deviceList.find((device) => {
    return device.vendorId === '9025' && device.productId === '67'
  })
  callback(selectedPort?.deviceId)
})
})
```

Event: 'hid-device-added' Returns:

- `event` Event
- `details` Object
  - `device` HIDDevice[]
  - `frame` WebFrameMain

Emitted when a new HID device becomes available. For example, when a new USB device is plugged in.

This event will only be emitted after `navigator.hid.requestDevice` has been called and `select-hid-device` has fired.

Event: 'hid-device-removed' Returns:

- `event` Event
- `details` Object
  - `device` HIDDevice[]
  - `frame` WebFrameMain

Emitted when a HID device has been removed. For example, this event will fire when a USB device is unplugged.

This event will only be emitted after `navigator.hid.requestDevice` has been called and `select-hid-device` has fired.

Event: 'select-serial-port' Returns:

- `event` Event

- `portList` SerialPort[]
- `webContents` WebContents
- `callback` Function
  - `portId` string

Emitted when a serial port needs to be selected when a call to `navigator.serial.requestPort` is made. `callback` should be called with `portId` to be selected, passing an empty string to `callback` will cancel the request. Additionally, permissioning on `navigator.serial` can be managed by using ses.setPermissionCheckHandler(handler) with the `serial` permission.

```
const { app, BrowserWindow } = require('electron')

let win = null

app.whenReady().then(() => {
  win = new BrowserWindow({
    width: 800,
    height: 600
  })

  win.webContents.session.setPermissionCheckHandler((webContents, permission, requestingOrig
    if (permission === 'serial') {
      // Add logic here to determine if permission should be given to allow serial selectio
      return true
    }
    return false
  })

  // Optionally, retrieve previously persisted devices from a persistent store
  const grantedDevices = fetchGrantedDevices()

  win.webContents.session.setDevicePermissionHandler((details) => {
    if (new URL(details.origin).hostname === 'some-host' && details.deviceType === 'serial'
      if (details.device.vendorId === 123 && details.device.productId === 345) {
        // Always allow this type of device (this allows skipping the call to `navigator.se
        return true
      }

      // Search through the list of devices that have previously been granted permission
      return grantedDevices.some((grantedDevice) => {
        return grantedDevice.vendorId === details.device.vendorId &&
               grantedDevice.productId === details.device.productId &&
               grantedDevice.serialNumber && grantedDevice.serialNumber === details.device.se
      })
    }
    return false
```

```
  })

  win.webContents.session.on('select-serial-port', (event, portList, webContents, callback)
    event.preventDefault()
    const selectedPort = portList.find((device) => {
      return device.vendorId === '9025' && device.productId === '67'
    })
    if (!selectedPort) {
      callback('')
    } else {
      callback(selectedPort.portId)
    }
  })
})
```

Event: 'serial-port-added' Returns:

- `event` Event
- `port` SerialPort
- `webContents` WebContents

Emitted after `navigator.serial.requestPort` has been called and `select-serial-port` has fired if a new serial port becomes available. For example, this event will fire when a new USB device is plugged in.

Event: 'serial-port-removed' Returns:

- `event` Event
- `port` SerialPort
- `webContents` WebContents

Emitted after `navigator.serial.requestPort` has been called and `select-serial-port` has fired if a serial port has been removed. For example, this event will fire when a USB device is unplugged.

Instance Methods

The following methods are available on instances of `Session`:

`ses.getCacheSize()` Returns `Promise<Integer>` - the session's current cache size, in bytes.

`ses.clearCache()` Returns `Promise<void>` - resolves when the cache clear operation is complete.

Clears the session's HTTP cache.

`ses.clearStorageData([options])`

- `options` Object (optional)
  - `origin` string (optional) - Should follow `window.location.origin`'s representation `scheme://host:port`.
  - `storages` string[] (optional) - The types of storages to clear, can contain: `appcache`, `cookies`, `filesystem`, `indexdb`, `localstorage`, `shadercache`, `websql`, `serviceworkers`, `cachestorage`. If not specified, clear all storage types.
  - `quotas` string[] (optional) - The types of quotas to clear, can contain: `temporary`, `persistent`, `syncable`. If not specified, clear all quotas.

Returns `Promise<void>` - resolves when the storage data has been cleared.

`ses.flushStorageData()` Writes any unwritten DOMStorage data to disk.

`ses.setProxy(config)`

- `config` Object
  - `mode` string (optional) - The proxy mode. Should be one of `direct`, `auto_detect`, `pac_script`, `fixed_servers` or `system`. If it's unspecified, it will be automatically determined based on other specified options.
    - `direct` In direct mode all connections are created directly, without any proxy involved.
    - `auto_detect` In auto_detect mode the proxy configuration is determined by a PAC script that can be downloaded at http://wpad/wpad.dat.
    - `pac_script` In pac_script mode the proxy configuration is determined by a PAC script that is retrieved from the URL specified in the `pacScript`. This is the default mode if `pacScript` is specified.
    - `fixed_servers` In fixed_servers mode the proxy configuration is specified in `proxyRules`. This is the default mode if `proxyRules` is specified.
    - `system` In system mode the proxy configuration is taken from the operating system. Note that the system mode is different from setting no proxy configuration. In the latter case, Electron falls back to the system settings only if no command-line options influence the proxy configuration.
  - `pacScript` string (optional) - The URL associated with the PAC file.
  - `proxyRules` string (optional) - Rules indicating which proxies to use.
  - `proxyBypassRules` string (optional) - Rules indicating which URLs should bypass the proxy settings.

Returns `Promise<void>` - Resolves when the proxy setting process is complete.

Sets the proxy settings.

When `mode` is unspecified, `pacScript` and `proxyRules` are provided together, the `proxyRules` option is ignored and `pacScript` configuration is applied.

You may need `ses.closeAllConnections` to close currently in flight connections to prevent pooled sockets using previous proxy from being reused by future requests.

The `proxyRules` has to follow the rules below:

```
proxyRules = schemeProxies[";"<schemeProxies>]
schemeProxies = [<urlScheme>"="]<proxyURIList>
urlScheme = "http" | "https" | "ftp" | "socks"
proxyURIList = <proxyURL>[","<proxyURIList>]
proxyURL = [<proxyScheme>"://"]<proxyHost>[":"<proxyPort>]
```

For example:

- `http=foopy:80;ftp=foopy2` - Use HTTP proxy `foopy:80` for `http://` URLs, and HTTP proxy `foopy2:80` for `ftp://` URLs.
- `foopy:80` - Use HTTP proxy `foopy:80` for all URLs.
- `foopy:80,bar,direct://` - Use HTTP proxy `foopy:80` for all URLs, failing over to `bar` if `foopy:80` is unavailable, and after that using no proxy.
- `socks4://foopy` - Use SOCKS v4 proxy `foopy:1080` for all URLs.
- `http=foopy,socks5://bar.com` - Use HTTP proxy `foopy` for http URLs, and fail over to the SOCKS5 proxy `bar.com` if `foopy` is unavailable.
- `http=foopy,direct://` - Use HTTP proxy `foopy` for http URLs, and use no proxy if `foopy` is unavailable.
- `http=foopy;socks=foopy2` - Use HTTP proxy `foopy` for http URLs, and use `socks4://foopy2` for all other URLs.

The `proxyBypassRules` is a comma separated list of rules described below:

- `[ URL_SCHEME "://" ] HOSTNAME_PATTERN [ ":" <port> ]`

  Match all hostnames that match the pattern HOSTNAME_PATTERN.

  Examples: “foobar.com”, “*foobar.com”, ”*.foobar.com”, “*foobar.com:99”, ”https://x*..y.com:99”

- `"." HOSTNAME_SUFFIX_PATTERN [ ":" PORT ]`

  Match a particular domain suffix.

  Examples: “.google.com”, “.com”, “http://.google.com”

- `[ SCHEME "://" ] IP_LITERAL [ ":" PORT ]`

  Match URLs which are IP address literals.

  Examples: “127.0.1”, “[0:0::1]”, “[::1]”, “http://[::1]:99”

- `IP_LITERAL "/" PREFIX_LENGTH_IN_BITS`

Match any URL that is to an IP literal that falls between the given range. IP range is specified using CIDR notation.

Examples: “192.168.1.1/16”, “fefe:13::abc/33”.

- `<local>`

  Match local addresses. The meaning of `<local>` is whether the host matches one of: “127.0.0.1”, “::1”, “localhost”.

`ses.resolveProxy(url)`

- `url` URL

Returns `Promise<string>` - Resolves with the proxy information for `url`.

`ses.forceReloadProxyConfig()` Returns `Promise<void>` - Resolves when the all internal states of proxy service is reset and the latest proxy configuration is reapplied if it’s already available. The pac script will be fetched from `pacScript` again if the proxy mode is `pac_script`.

`ses.setDownloadPath(path)`

- `path` string - The download location.

Sets download saving directory. By default, the download directory will be the `Downloads` under the respective app folder.

`ses.enableNetworkEmulation(options)`

- `options` Object
  - `offline` boolean (optional) - Whether to emulate network outage. Defaults to false.
  - `latency` Double (optional) - RTT in ms. Defaults to 0 which will disable latency throttling.
  - `downloadThroughput` Double (optional) - Download rate in Bps. Defaults to 0 which will disable download throttling.
  - `uploadThroughput` Double (optional) - Upload rate in Bps. Defaults to 0 which will disable upload throttling.

Emulates network with the given configuration for the `session`.

```
// To emulate a GPRS connection with 50kbps throughput and 500 ms latency.
window.webContents.session.enableNetworkEmulation({
  latency: 500,
  downloadThroughput: 6400,
  uploadThroughput: 6400
})
```

```
// To emulate a network outage.
window.webContents.session.enableNetworkEmulation({ offline: true })
```

`ses.preconnect(options)`

- `options` Object
  - `url` string - URL for preconnect. Only the origin is relevant for opening the socket.
  - `numSockets` number (optional) - number of sockets to preconnect. Must be between 1 and 6. Defaults to 1.

Preconnects the given number of sockets to an origin.

`ses.closeAllConnections()` Returns `Promise<void>` - Resolves when all connections are closed.

Note: It will terminate / fail all requests currently in flight.

`ses.disableNetworkEmulation()` Disables any network emulation already active for the `session`. Resets to the original network configuration.

`ses.setCertificateVerifyProc(proc)`

- `proc` Function | null
  - `request` Object
    - `hostname` string
    - `certificate` Certificate
    - `validatedCertificate` Certificate
    - `isIssuedByKnownRoot` boolean - `true` if Chromium recognises the root CA as a standard root. If it isn't then it's probably the case that this certificate was generated by a MITM proxy whose root has been installed locally (for example, by a corporate proxy). This should not be trusted if the `verificationResult` is not `OK`.
    - `verificationResult` string - `OK` if the certificate is trusted, otherwise an error like `CERT_REVOKED`.
    - `errorCode` Integer - Error code.
  - `callback` Function
    - `verificationResult` Integer - Value can be one of certificate error codes from here. Apart from the certificate error codes, the following special codes can be used.
      - `0` - Indicates success and disables Certificate Transparency verification.
      - `-2` - Indicates failure.
      - `-3` - Uses the verification result from chromium.

Sets the certificate verify proc for `session`, the `proc` will be called with `proc(request, callback)` whenever a server certificate verification is requested. Calling `callback(0)` accepts the certificate, calling `callback(-2)` rejects it.

Calling `setCertificateVerifyProc(null)` will revert back to default certificate verify proc.

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow()

win.webContents.session.setCertificateVerifyProc((request, callback) => {
  const { hostname } = request
  if (hostname === 'github.com') {
    callback(0)
  } else {
    callback(-2)
  }
})
```

NOTE: The result of this procedure is cached by the network service.

`ses.setPermissionRequestHandler(handler)`

- `handler` Function | null
  - `webContents` WebContents - WebContents requesting the permission. Please note that if the request comes from a subframe you should use `requestingUrl` to check the request origin.
  - `permission` string - The type of requested permission.
    - `clipboard-read` - Request access to read from the clipboard.
    - `media` - Request access to media devices such as camera, microphone and speakers.
    - `display-capture` - Request access to capture the screen.
    - `mediaKeySystem` - Request access to DRM protected content.
    - `geolocation` - Request access to user's current location.
    - `notifications` - Request notification creation and the ability to display them in the user's system tray.
    - `midi` - Request MIDI access in the `webmidi` API.
    - `midiSysex` - Request the use of system exclusive messages in the `webmidi` API.
    - `pointerLock` - Request to directly interpret mouse movements as an input method. Click here to know more.
    - `fullscreen` - Request for the app to enter fullscreen mode.
    - `openExternal` - Request to open links in external applications.
    - `unknown` - An unrecognized permission request
  - `callback` Function
    - `permissionGranted` boolean - Allow or deny the permission.

  - `details` Object - Some properties are only available on certain permission types.
    * `externalURL` string (optional) - The url of the `openExternal` request.
    * `securityOrigin` string (optional) - The security origin of the `media` request.
    * `mediaTypes` string[] (optional) - The types of media access being requested, elements can be `video` or `audio`
    * `requestingUrl` string - The last URL the requesting frame loaded
    * `isMainFrame` boolean - Whether the frame making the request is the main frame

Sets the handler which can be used to respond to permission requests for the `session`. Calling `callback(true)` will allow the permission and `callback(false)` will reject it. To clear the handler, call `setPermissionRequestHandler(null)`. Please note that you must also implement `setPermissionCheckHandler` to get complete permission handling. Most web APIs do a permission check and then make a permission request if the check is denied.

```
const { session } = require('electron')
session.fromPartition('some-partition').setPermissionRequestHandler((webContents, permissio
  if (webContents.getURL() === 'some-host' && permission === 'notifications') {
    return callback(false) // denied.
  }

  callback(true)
})
```

`ses.setPermissionCheckHandler(handler)`

- `handler` Function<boolean> | null
  - `webContents` (WebContents | null) - WebContents checking the permission. Please note that if the request comes from a subframe you should use `requestingUrl` to check the request origin. All cross origin sub frames making permission checks will pass a `null` webContents to this handler, while certain other permission checks such as `notifications` checks will always pass `null`. You should use `embeddingOrigin` and `requestingOrigin` to determine what origin the owning frame and the requesting frame are on respectively.
  - `permission` string - Type of permission check. Valid values are `midiSysex`, `notifications`, `geolocation`, `media`,`mediaKeySystem`,`midi`, `pointerLock`, `fullscreen`, `openExternal`, `hid`, or `serial`.
  - `requestingOrigin` string - The origin URL of the permission check
  - `details` Object - Some properties are only available on certain permission types.
    * `embeddingOrigin` string (optional) - The origin of the frame

embedding the frame that made the permission check. Only set for cross-origin sub frames making permission checks.
    * `securityOrigin` string (optional) - The security origin of the `media` check.
    * `mediaType` string (optional) - The type of media access being requested, can be `video`, `audio` or `unknown`
    * `requestingUrl` string (optional) - The last URL the requesting frame loaded. This is not provided for cross-origin sub frames making permission checks.
    * `isMainFrame` boolean - Whether the frame making the request is the main frame

Sets the handler which can be used to respond to permission checks for the `session`. Returning `true` will allow the permission and `false` will reject it. Please note that you must also implement `setPermissionRequestHandler` to get complete permission handling. Most web APIs do a permission check and then make a permission request if the check is denied. To clear the handler, call `setPermissionCheckHandler(null)`.

```
const { session } = require('electron')
const url = require('url')
session.fromPartition('some-partition').setPermissionCheckHandler((webContents, permission,
  if (new URL(requestingOrigin).hostname === 'some-host' && permission === 'notifications')
    return true // granted
  }

  return false // denied
})
```

`ses.setDevicePermissionHandler(handler)`

- `handler` Function<boolean> | null
  - `details` Object
    * `deviceType` string - The type of device that permission is being requested on, can be `hid` or `serial`.
    * `origin` string - The origin URL of the device permission check.
    * `device` HIDDevice | SerialPort- the device that permission is being requested for.
    * `frame` WebFrameMain - WebFrameMain checking the device permission.

Sets the handler which can be used to respond to device permission checks for the `session`. Returning `true` will allow the device to be permitted and `false` will reject it. To clear the handler, call `setDevicePermissionHandler(null)`. This handler can be used to provide default permissioning to devices without first calling for permission to devices (eg via `navigator.hid.requestDevice`). If this handler is not defined, the default device permissions as granted through

device selection (eg via `navigator.hid.requestDevice`) will be used. Additionally, the default behavior of Electron is to store granted device permision through the lifetime of the corresponding WebContents. If longer term storage is needed, a developer can store granted device permissions (eg when handling the `select-hid-device` event) and then read from that storage with `setDevicePermissionHandler`.

```
const { app, BrowserWindow } = require('electron')

let win = null

app.whenReady().then(() => {
  win = new BrowserWindow()

  win.webContents.session.setPermissionCheckHandler((webContents, permission, requestingOrig
    if (permission === 'hid') {
      // Add logic here to determine if permission should be given to allow HID selection
      return true
    } else if (permission === 'serial') {
      // Add logic here to determine if permission should be given to allow serial port sele
    }
    return false
  })

  // Optionally, retrieve previously persisted devices from a persistent store
  const grantedDevices = fetchGrantedDevices()

  win.webContents.session.setDevicePermissionHandler((details) => {
    if (new URL(details.origin).hostname === 'some-host' && details.deviceType === 'hid') {
      if (details.device.vendorId === 123 && details.device.productId === 345) {
        // Always allow this type of device (this allows skipping the call to `navigator.hid
        return true
      }

      // Search through the list of devices that have previously been granted permission
      return grantedDevices.some((grantedDevice) => {
        return grantedDevice.vendorId === details.device.vendorId &&
               grantedDevice.productId === details.device.productId &&
               grantedDevice.serialNumber && grantedDevice.serialNumber === details.device.se
      })
    } else if (details.deviceType === 'serial') {
      if (details.device.vendorId === 123 && details.device.productId === 345) {
        // Always allow this type of device (this allows skipping the call to `navigator.hid
        return true
      }
    }
```

```
    return false
  })

  win.webContents.session.on('select-hid-device', (event, details, callback) => {
    event.preventDefault()
    const selectedDevice = details.deviceList.find((device) => {
      return device.vendorId === '9025' && device.productId === '67'
    })
    callback(selectedPort?.deviceId)
  })
})
```

`ses.clearHostResolverCache()` Returns `Promise<void>` - Resolves when the operation is complete.

Clears the host resolver cache.

`ses.allowNTLMCredentialsForDomains(domains)`

- `domains` string - A comma-separated list of servers for which integrated authentication is enabled.

Dynamically sets whether to always send credentials for HTTP NTLM or Negotiate authentication.

```
const { session } = require('electron')
// consider any url ending with `example.com`, `foobar.com`, `baz`
// for integrated authentication.
session.defaultSession.allowNTLMCredentialsForDomains('*example.com, *foobar.com, *baz')

// consider all urls for integrated authentication.
session.defaultSession.allowNTLMCredentialsForDomains('*')
```

`ses.setUserAgent(userAgent[, acceptLanguages])`

- `userAgent` string
- `acceptLanguages` string (optional)

Overrides the `userAgent` and `acceptLanguages` for this session.

The `acceptLanguages` must a comma separated ordered list of language codes, for example `"en-US,fr,de,ko,zh-CN,ja"`.

This doesn't affect existing `WebContents`, and each `WebContents` can use `webContents.setUserAgent` to override the session-wide user agent.

`ses.isPersistent()` Returns `boolean` - Whether or not this session is a persistent one. The default `webContents` session of a `BrowserWindow` is persistent.

When creating a session from a partition, session prefixed with `persist:` will be persistent, while others will be temporary.

`ses.getUserAgent()` Returns `string` - The user agent for this session.

`ses.setSSLConfig(config)`

- `config` Object
  - `minVersion` string (optional) - Can be `tls1`, `tls1.1`, `tls1.2` or `tls1.3`. The minimum SSL version to allow when connecting to remote servers. Defaults to `tls1`.
  - `maxVersion` string (optional) - Can be `tls1.2` or `tls1.3`. The maximum SSL version to allow when connecting to remote servers. Defaults to `tls1.3`.
  - `disabledCipherSuites` Integer[] (optional) - List of cipher suites which should be explicitly prevented from being used in addition to those disabled by the net built-in policy. Supported literal forms: 0xAABB, where AA is `cipher_suite[0]` and BB is `cipher_suite[1]`, as defined in RFC 2246, Section 7.4.1.2. Unrecognized but parsable cipher suites in this form will not return an error. Ex: To disable TLS_RSA_WITH_RC4_128_MD5, specify 0x0004, while to disable TLS_ECDH_ECDSA_WITH_RC4_128_SHA, specify 0xC002. Note that TLSv1.3 ciphers cannot be disabled using this mechanism.

Sets the SSL configuration for the session. All subsequent network requests will use the new configuration. Existing network connections (such as WebSocket connections) will not be terminated, but old sockets in the pool will not be reused for new connections.

`ses.getBlobData(identifier)`

- `identifier` string - Valid UUID.

Returns `Promise<Buffer>` - resolves with blob data.

`ses.downloadURL(url)`

- `url` string

Initiates a download of the resource at `url`. The API will generate a DownloadItem that can be accessed with the will-download event.

Note: This does not perform any security checks that relate to a page's origin, unlike `webContents.downloadURL`.

`ses.createInterruptedDownload(options)`

- `options` Object

  - `path` string - Absolute path of the download.
  - `urlChain` string[] - Complete URL chain for the download.
  - `mimeType` string (optional)
  - `offset` Integer - Start range for the download.
  - `length` Integer - Total length of the download.
  - `lastModified` string (optional) - Last-Modified header value.
  - `eTag` string (optional) - ETag header value.
  - `startTime` Double (optional) - Time when download was started in number of seconds since UNIX epoch.

Allows resuming `cancelled` or `interrupted` downloads from previous `Session`. The API will generate a DownloadItem that can be accessed with the will-download event. The DownloadItem will not have any `WebContents` associated with it and the initial state will be `interrupted`. The download will start only when the `resume` API is called on the DownloadItem.

`ses.clearAuthCache()` Returns `Promise<void>` - resolves when the session's HTTP authentication cache has been cleared.

`ses.setPreloads(preloads)`

- `preloads` string[] - An array of absolute path to preload scripts

Adds scripts that will be executed on ALL web contents that are associated with this session just before normal `preload` scripts run.

`ses.getPreloads()` Returns `string[]` an array of paths to preload scripts that have been registered.

`ses.setCodeCachePath(path)`

- `path` String - Absolute path to store the v8 generated JS code cache from the renderer.

Sets the directory to store the generated JS code cache for this session. The directory is not required to be created by the user before this call, the runtime will create if it does not exist otherwise will use the existing directory. If directory cannot be created, then code cache will not be used and all operations related to code cache will fail silently inside the runtime. By default, the directory will be `Code Cache` under the respective user data folder.

`ses.clearCodeCaches(options)`

- `options` Object
  - `urls` String[] (optional) - An array of url corresponding to the resource whose generated code cache needs to be removed. If the list is empty then all entries in the cache directory will be removed.

Returns `Promise<void>` - resolves when the code cache clear operation is complete.

`ses.setSpellCheckerEnabled(enable)`

- `enable` boolean

Sets whether to enable the builtin spell checker.

`ses.isSpellCheckerEnabled()` Returns `boolean` - Whether the builtin spell checker is enabled.

`ses.setSpellCheckerLanguages(languages)`

- `languages` string[] - An array of language codes to enable the spellchecker for.

The built in spellchecker does not automatically detect what language a user is typing in. In order for the spell checker to correctly check their words you must call this API with an array of language codes. You can get the list of supported language codes with the `ses.availableSpellCheckerLanguages` property.

Note: On macOS the OS spellchecker is used and will detect your language automatically. This API is a no-op on macOS.

`ses.getSpellCheckerLanguages()` Returns `string[]` - An array of language codes the spellchecker is enabled for. If this list is empty the spellchecker will fallback to using `en-US`. By default on launch if this setting is an empty list Electron will try to populate this setting with the current OS locale. This setting is persisted across restarts.

Note: On macOS the OS spellchecker is used and has its own list of languages. This API is a no-op on macOS.

`ses.setSpellCheckerDictionaryDownloadURL(url)`

- `url` string - A base URL for Electron to download hunspell dictionaries from.

By default Electron will download hunspell dictionaries from the Chromium CDN. If you want to override this behavior you can use this API to point the dictionary downloader at your own hosted version of the hunspell dictionaries. We publish a `hunspell_dictionaries.zip` file with each release which contains the files you need to host here.

The file server must be **case insensitive**. If you cannot do this, you must upload each file twice: once with the case it has in the ZIP file and once with the filename as all lowercase.

If the files present in `hunspell_dictionaries.zip` are available at `https://example.com/dictionaries/language-code.bdic` then you should call this api with `ses.setSpellCheckerDictionaryDownloadURL('https://example.com/dictionaries/')`. Please note the trailing slash. The URL to the dictionaries is formed as `${url}${filename}`.

Note: On macOS the OS spellchecker is used and therefore we do not download any dictionary files. This API is a no-op on macOS.

`ses.listWordsInSpellCheckerDictionary()` Returns `Promise<string[]>` - An array of all words in app's custom dictionary. Resolves when the full dictionary is loaded from disk.

`ses.addWordToSpellCheckerDictionary(word)`

- `word` string - The word you want to add to the dictionary

Returns `boolean` - Whether the word was successfully written to the custom dictionary. This API will not work on non-persistent (in-memory) sessions.

Note: On macOS and Windows 10 this word will be written to the OS custom dictionary as well

`ses.removeWordFromSpellCheckerDictionary(word)`

- `word` string - The word you want to remove from the dictionary

Returns `boolean` - Whether the word was successfully removed from the custom dictionary. This API will not work on non-persistent (in-memory) sessions.

Note: On macOS and Windows 10 this word will be removed from the OS custom dictionary as well

`ses.loadExtension(path[, options])`

- `path` string - Path to a directory containing an unpacked Chrome extension
- `options` Object (optional)
  - `allowFileAccess` boolean - Whether to allow the extension to read local files over `file://` protocol and inject content scripts into `file://` pages. This is required e.g. for loading devtools extensions on `file://` URLs. Defaults to false.

Returns `Promise<Extension>` - resolves when the extension is loaded.

This method will raise an exception if the extension could not be loaded. If there are warnings when installing the extension (e.g. if the extension requests an API that Electron does not support) then they will be logged to the console.

Note that Electron does not support the full range of Chrome extensions APIs. See Supported Extensions APIs for more details on what is supported.

Note that in previous versions of Electron, extensions that were loaded would be remembered for future runs of the application. This is no longer the case: `loadExtension` must be called on every boot of your app if you want the extension to be loaded.

```
const { app, session } = require('electron')
const path = require('path')

app.on('ready', async () => {
  await session.defaultSession.loadExtension(
    path.join(__dirname, 'react-devtools'),
    // allowFileAccess is required to load the devtools extension on file:// URLs.
    { allowFileAccess: true }
  )
  // Note that in order to use the React DevTools extension, you'll need to
  // download and unzip a copy of the extension.
})
```

This API does not support loading packed (.crx) extensions.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

Note: Loading extensions into in-memory (non-persistent) sessions is not supported and will throw an error.

`ses.removeExtension(extensionId)`

- `extensionId` string - ID of extension to remove

Unloads an extension.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

`ses.getExtension(extensionId)`

- `extensionId` string - ID of extension to query

Returns `Extension` | `null` - The loaded extension with the given ID.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

`ses.getAllExtensions()` Returns `Extension[]` - A list of all loaded extensions.

Note: This API cannot be called before the `ready` event of the `app` module is emitted.

ses.getStoragePath() A `string | null` indicating the absolute file system path where data for this session is persisted on disk. For in memory sessions this returns `null`.

Instance Properties

The following properties are available on instances of `Session`:

ses.availableSpellCheckerLanguages *Readonly* A `string[]` array which consists of all the known available spell checker languages. Providing a language code to the `setSpellCheckerLanguages` API that isn't in this array will result in an error.

ses.spellCheckerEnabled A `boolean` indicating whether builtin spell checker is enabled.

ses.storagePath *Readonly* A `string | null` indicating the absolute file system path where data for this session is persisted on disk. For in memory sessions this returns `null`.

ses.cookies *Readonly* A `Cookies` object for this session.

ses.serviceWorkers *Readonly* A `ServiceWorkers` object for this session.

ses.webRequest *Readonly* A `WebRequest` object for this session.

ses.protocol *Readonly* A `Protocol` object for this session.

```
const { app, session } = require('electron')
const path = require('path')

app.whenReady().then(() => {
  const protocol = session.fromPartition('some-partition').protocol
  if (!protocol.registerFileProtocol('atom', (request, callback) => {
    const url = request.url.substr(7)
    callback({ path: path.normalize(`${__dirname}/${url}`) })
  })) {
    console.error('Failed to register protocol')
  }
})
```

ses.netLog *Readonly* A `NetLog` object for this session.

```
const { app, session } = require('electron')

app.whenReady().then(async () => {
```

```
  const netLog = session.fromPartition('some-partition').netLog
  netLog.startLogging('/path/to/net-log')
  // After some network events
  const path = await netLog.stopLogging()
  console.log('Net-logs written to', path)
})
```