



# Electron

## tutorialspoint

SIMPLY EASY LEARNING

[www.tutorialspoint.com](http://www.tutorialspoint.com)



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

## About the Tutorial

---

Electron is an open source library developed by GitHub for building cross-platform desktop applications with HTML, CSS, and JavaScript. Electron accomplishes this by combining Chromium and Node.js into a single runtime and apps can be packaged for Mac, Windows, and Linux.

## Audience

---

This tutorial is designed for those learners who aspire to build cross-platform Desktop apps for Linux, Windows and MacOS.

## Prerequisites

---

Before proceeding with this tutorial, you should have a basic understanding of Javascript(ES6) and HTML. You also need to know about a few native Node.js APIs such as *file handling, processes*, etc.

## Copyright & Disclaimer

---

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com](mailto:contact@tutorialspoint.com)

## Table of Contents

---

About the Tutorial.....	i
Audience .....	i
Prerequisites .....	i
Copyright & Disclaimer.....	i
Table of Contents .....	ii
1. ELECTRON – OVERVIEW .....	1
2. ELECTRON – INSTALLATION .....	2
3. ELECTRON – HOW ELECTRON WORKS .....	4
4. ELECTRON – HELLO WORLD .....	5
5. ELECTRON – BUILDING UIS .....	7
6. ELECTRON – FILE HANDLING .....	10
7. ELECTRON – NATIVE NODE LIBRARIES .....	14
OS Module .....	14
Net Module .....	15
8. ELECTRON – INTERPROCESS COMMUNICATION.....	18
9. ELECTRON – SYSTEM DIALOGS .....	21
10. ELECTRON – MENUS.....	24
11. ELECTRON – SYSTEM TRAY .....	29
12. ELECTRON – NOTIFICATIONS.....	32
13. ELECTRON – WEBVIEW.....	35

14. ELECTRON – AUDIO AND VIDEO CAPTURING.....	38
15. ELECTRON – DEFINING SHORTCUTS .....	42
16. ELECTRON – ENVIRONMENT VARIABLES .....	45
<b>Production Variables</b> .....	45
<b>Development Variables</b> .....	45
17. ELECTRON – DEBUGGING .....	47
<b>Debugging the Main Process</b> .....	47
18. ELECTRON – PACKAGING APPS.....	50
<b>Supported Platforms</b> .....	50
<b>Installation</b> .....	50
<b>Packaging Apps</b> .....	50
19. ELECTRON – RESOURCES.....	52

# 1. Electron – Overview

## Why Electron?

Electron enables you to create desktop applications with pure JavaScript by providing a runtime with rich native (operating system) APIs.

This does not mean Electron is a JavaScript binding to graphical user interface (GUI) libraries. Instead, Electron uses web pages as its GUI, so you can also see it as a minimal Chromium browser, controlled by JavaScript. So all the electron apps are technically web pages running in a browser that can leverage your OS APIs.

## Who Uses Electron?

Github developed Electron for creating the text editor Atom. They were both open sourced in 2014. Electron is used by many companies like Microsoft, Github, Slack, etc.

Electron has been used to create a number of apps. Following are a few notable apps:

- Slack desktop
- Wordpress desktop app
- Visual Studio Code
- Caret Markdown Editor
- Nylas Email App
- GitKraken git client

## 2. Electron – Installation

To get started with developing using the Electron, you need to have Node and npm(node package manager) installed. If you do not already have these, head over to [Node setup](#) to install node on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version  
npm --version
```

The above command will generate the following output:

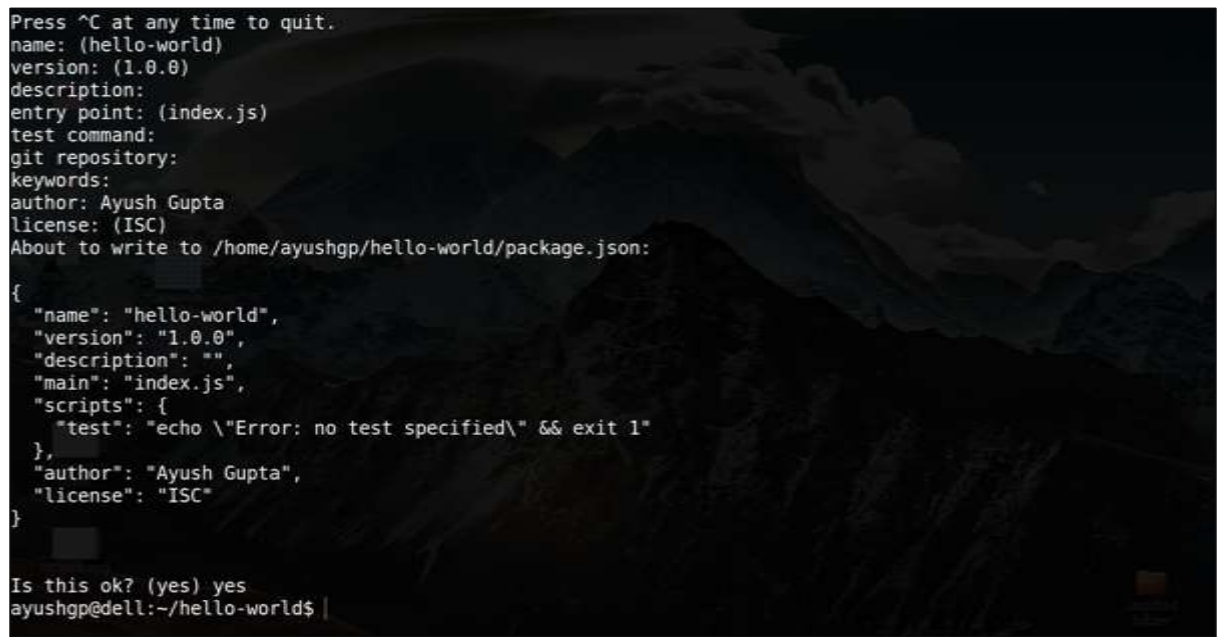
```
v6.9.1  
3.10.8
```

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

- Fire up your terminal/cmd, create a new folder named hello-world and open that folder using the cd command.
- Now to create the package.json file using npm, use the following command.

```
npm init
```

- It will ask you for the following information:



```
Press ^C at any time to quit.  
name: (hello-world)  
version: (1.0.0)  
description:  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author: Ayush Gupta  
license: (ISC)  
About to write to /home/ayushgp/hello-world/package.json:  
{  
  "name": "hello-world",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "Ayush Gupta",  
  "license": "ISC"  
}  
Is this ok? (yes) yes  
ayushgp@dell:~/hello-world$
```

Just keep pressing Enter, and enter your name at the “author name” field.

Create a new folder and open it using the `cd` command. Now run the following command to install Electron globally.

```
$ npm install -g electron-prebuilt
```

Once it executes, you can check if Electron is installed the right way by running the following command:

```
$ electron --version
```

You should get the output:

```
v1.4.13
```

Now that we have set up Electron, let us move on to creating our first app using it.

### 3. Electron – How Electron Works

Electron takes a main file defined in your *package.json* file and executes it. This main file creates application windows which contain rendered web pages and interaction with the native GUI (graphical user interface) of your Operating System.

As you start an application using Electron, a **main process** is created. This main process is responsible for interacting with the native GUI of the Operating System. It creates the GUI of your application.

Just starting the main process does not give the users of your application any application window. These are created by the main process in the main file by using the *BrowserWindow* module. Each browser window then runs its own **renderer process**. The renderer process takes an HTML file which references the usual CSS files, JavaScript files, images, etc. and renders it in the window.

The main process can access the native GUI through modules available directly in Electron. The desktop application can access all Node modules like the file system module for handling files, request to make HTTP calls, etc.

#### Difference between Main and Renderer processes

The main process creates web pages by creating the *BrowserWindow* instances. Each *BrowserWindow* instance runs the web page in its own renderer process. When a *BrowserWindow* instance is destroyed, the corresponding renderer process is also terminated.

The main process manages all web pages and their corresponding renderer processes. Each renderer process is isolated and only cares about the web page running in it.



## 4. Electron – Hello World

We have created a **package.json** file for our project. Now we will create our first desktop app using Electron.

Create a new file called *main.js*. Enter the following code in it:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

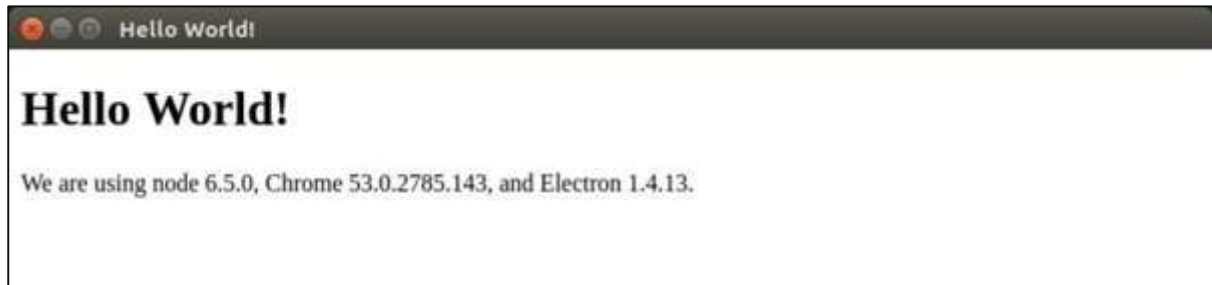
Create another file, this time an HTML file called *index.html*. Enter the following code in it.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using node <script>document.write(process.versions.node)</script>,
    Chrome <script>document.write(process.versions.chrome)</script>,
    and Electron <script>document.write(process.versions.electron)</script>.
  </body>
</html>
```

Run this app using the following command:

```
$ electron ./main.js
```

A new window will open up. It will look like the following:



## How Does This App Work?

We created a main file and an HTML file. The main file uses two modules – *app* and *BrowserWindow*. The *app* module is used to control your application's event lifecycle while the *BrowserWindow* module is used to create and control browser windows.

We defined a *createWindow* function, where we are creating a new *BrowserWindow* and attaching a URL to this *BrowserWindow*. This is the HTML file that is rendered and shown to us when we run the app.

We have used a native Electron object *process* in our html file. This object is extended from the Node.js *process* object and includes all of **its** functionalities while adding many more.

## 5. Electron – Building UIs

The User Interface of Electron apps is built using HTML, CSS and JS. So we can leverage all the available tools for front-end web development here as well. You can use the tools such as Angular, Backbone, React, Bootstrap, and Foundation, to build the apps.

You can use Bower to manage these front-end dependencies. Install bower using:

```
$ npm install -g bower
```

Now you can get all the available JS and CSS frameworks, libraries, plugins, etc. using bower. For example, to get the latest stable version of bootstrap, enter the following command:

```
$ bower install bootstrap
```

This will download bootstrap in *bower\_components*. Now you can reference this library in your HTML. Let us create a simple page using these libraries.

Let us now install jquery using the npm command:

```
$ npm install --save jquery
```

Further, this will be *required* in our view.js file. We already have a main.js setup as follows:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

Open your **index.html** file and enter the following code in it:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
    <link rel="stylesheet"
href="./bower_components/bootstrap/dist/css/bootstrap.min.css" />
  </head>
  <body>
    <div class="container">
      <h1>This page is using Bootstrap and jQuery!</h1>
      <h3 id="click-counter"></h3>
      <button class="btn btn-success" id="countbtn">Click here</button>
      <script src="./view.js" ></script>
    </div>
  </body>
</html>
```

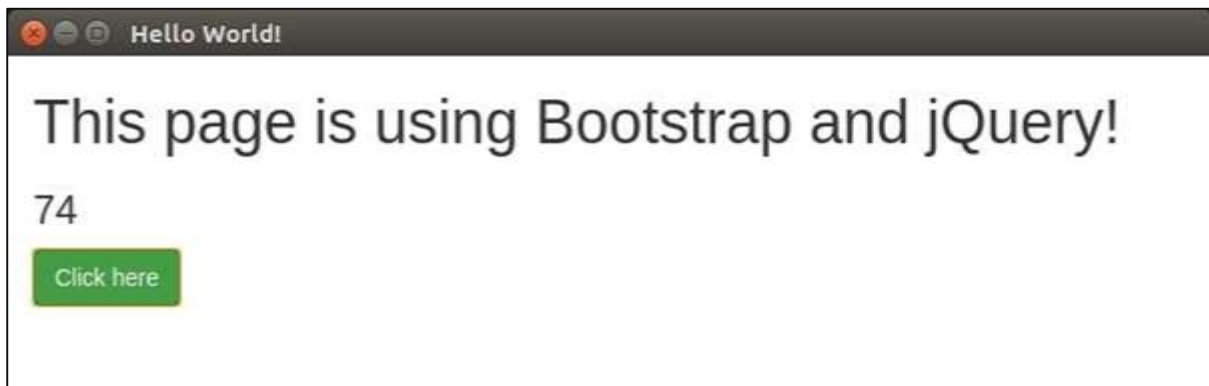
Create **view.js** and enter the click counter logic in it:

```
let $ = require('jquery') // jQuery now loaded and assigned to $
let count = 0
$('#click-counter').text(count.toString())
$('#countbtn').on('click', () => {
  count ++
  $('#click-counter').text(count)
})
```

Run the app using the following command:

```
$ electron ./main.js
```

The above command will generate the output as in the following screenshot:



You can build your native app just like you build websites. If you do not want users to be restricted to an exact window size, you can leverage the responsive design and allow users to use your app in a flexible manner.



```

    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Enter Names and Email addresses of your contacts</h1>
    <div class="form-group">
      <label for="Name">Name</label><input type="text" name="Name"
value="" id="Name" placeholder="Name" class="form-control" required>
    </div>
    <div class="form-group">
      <label for="Email">Email</label><input type="email" name="Email"
value="" id="Email" placeholder="Email" class="form-control" required>
    </div>
    <div class="form-group">
      <button class="btn btn-primary" id="add-to-list">Add to
list!</button>
    </div>
    <div id="contact-list">
      <table class="table-striped" id="contact-table">
        <tr>
          <th class="col-xs-2">S. No.</th>
          <th class="col-xs-4">Name</th>
          <th class="col-xs-6">Email</th>
        </tr>
      </table>
    </div>
    <script src="./view.js" ></script>
  </div>
</body>
</html>

```

Now we need to handle the addition event. We will do this in our **view.js** file.

We will create a function *loadAndDisplayContacts()* that will initially load contacts from the file. After creating the *loadAndDisplayContacts()* function, we will create a click handler on our **add to list** button. This will add the entry to both the file and the table.

In your view.js file, enter the following code:

```

let $ = require('jquery')
let fs = require('fs')
let filename = 'contacts'
let sno = 0

$('#add-to-list').on('click', () => {
  let name = $('#Name').val()
  let email = $('#Email').val()

  fs.appendFile('contacts', name + ',' + email + '\n')

  addEntry(name, email)
})

function addEntry(name, email) {
  if(name && email){
    sno++
    let updateString = '' + sno + '' + name + '' + email + ''
    $('#contact-table').append(updateString)
  }
}

function loadAndDisplayContacts(){
  //Check if file exists
  if(fs.existsSync(filename)){
    let data = fs.readFileSync(filename, 'utf8').split('\n')
    data.forEach((contact, index) => {
      let [ name, email ] = contact.split(',')
      addEntry(name, email)
    })
  }
  else {
    console.log("File Doesn't Exist. Creating new file.")
    fs.writeFile(filename, '', (err) => {
      if(err)
        console.log(err)
    })
  }
}

```

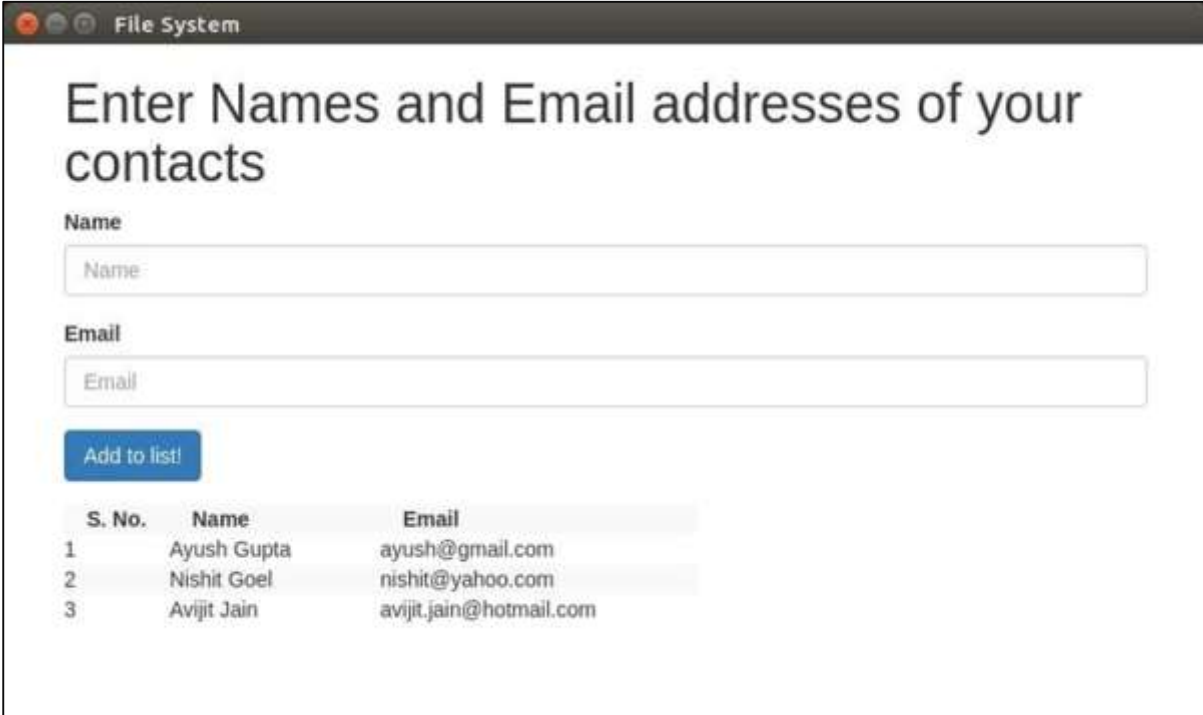


```
}  
}  
  
loadAndDisplayContacts()
```

Now run the application, using the following command:

```
$ electron ./main.js
```

Once you add some contacts to it, the application will look like:



Enter Names and Email addresses of your contacts

Name

Email

S. No.	Name	Email
1	Ayush Gupta	ayush@gmail.com
2	Nishit Goel	nishit@yahoo.com
3	Avijit Jain	avijit.jain@hotmail.com

For more **fs module API calls**, please refer to [Node File System tutorial](#).

Now we can handle files using Electron. We will look at how to call the save and open dialog boxes(native) for files in the dialogs chapter.

## 7. Electron – Native Node Libraries

We used a node module, `fs`, in the previous chapter. We will now look at some other node modules that we can use with Electron.

### OS Module

Using the OS module, we can get a lot of information about the system our application is running on. Following are a few methods that help while the app is being created. These methods help us customize the apps according to the OS that they are running on.

Function	Description
<code>os.userInfo([options])</code>	The <b><code>os.userInfo()</code></b> method returns information about the currently effective user. This information can be used to personalize the application for the user even without explicitly asking for information.
<code>os.platform()</code>	The <b><code>os.platform()</code></b> method returns a string identifying the operating system platform. This can be used to customize the app according to the user OS.
<code>os.homedir()</code>	The <b><code>os.homedir()</code></b> method returns the home directory of the current user as a string. Generally, configs of all users reside in the home directory of the user. So this can be used for the same purpose for our app.
<code>os.arch()</code>	The <b><code>os.arch()</code></b> method returns a string identifying the operating system CPU architecture. This can be used when running on exotic architectures to adapt your application for that system.
<code>os.EOL</code>	A string constant defining the operating system-specific end-of-line marker. This should be used whenever ending lines in files on the host OS.

Using the same **`main.js`** file and the following HTML file, we can print these properties on the screen:

```
<html>
  <head>
    <title>OS Module</title>
  </head>
  <body>
    <script>
      let os = require('os')
```

```

        document.write('User Info: ' + JSON.stringify(os.userInfo()) + '<br>' +
            'Platform: ' + os.platform() + '<br>' +
            'User home directory: ' + os.homedir() + '<br>' +
            'OS Architecture: ' + os.arch() + '<br>')
    </script>
</body>
</html>

```

Now run the app using the following command:

```
$ electron ./main.js
```

The above command will generate the following output:

```

User Info:
{"uid":1000,"gid":1000,"username":"ayushgp","homedir":"/home/ayushgp","shell":"/usr/bin/zsh"}
Platform: linux
User home directory: /home/ayushgp
OS Architecture: x64

```

## Net Module

The net module is used for network related work in the app. We can create both servers and socket connections using this module. Generally, the use of wrapper module from npm is recommended over the use of the net module for networking related tasks.

The following tables lists down the most useful methods from the module:

Function	Description
net.createServer([options][, connectionListener])	Creates a new TCP server. The <b>connectionListener</b> argument is automatically set as a listener for the <b>'connection'</b> event.
net.createConnection(options[, connectionListener])	A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.
net.Server.listen(port[, host][, backlog][, callback])	Begin accepting connections on the specified port and host. If the host is omitted, the server will accept connections directed to any IPv4 address.
net.Server.close([callback])	Finally closed when all connections are ended and the server emits a 'close' event.

<code>net.Socket.connect(port[, host][, connectListener])</code>	Opens the connection for a given socket. If port and host are given, then the socket will be opened as a TCP socket.
--	--

The net module comes with a few other methods too. To get a more comprehensive list, see [this](#).

Now, let us create an electron app that uses the net module to create connections to the server. We will need to create a new file, **server.js**:

```
var net = require('net');
var server = net.createServer(function(connection) {
  console.log('Client Connected');

  connection.on('end', function() {
    console.log('client disconnected');
  });
  connection.write('Hello World!\r\n');
  connection.pipe(connection);
});

server.listen(8080, function() {
  console.log('Server running on http://localhost:8080');
});
```

Using the same main.js file, replace the HTML file with the following:

```
<html>
  <head>
    <title>net Module</title>
  </head>
  <body>
    <script>
      var net = require('net');
      var client = net.connect({port: 8080}, function() {
        console.log('Connection established!');
      });
      client.on('data', function(data) {
        document.write(data.toString());
      });
    </script>
  </body>
</html>
```

```
        client.end();
    });
    client.on('end', function() {
        console.log('Disconnected :(');
    });
</script>
</body>
</html>
```

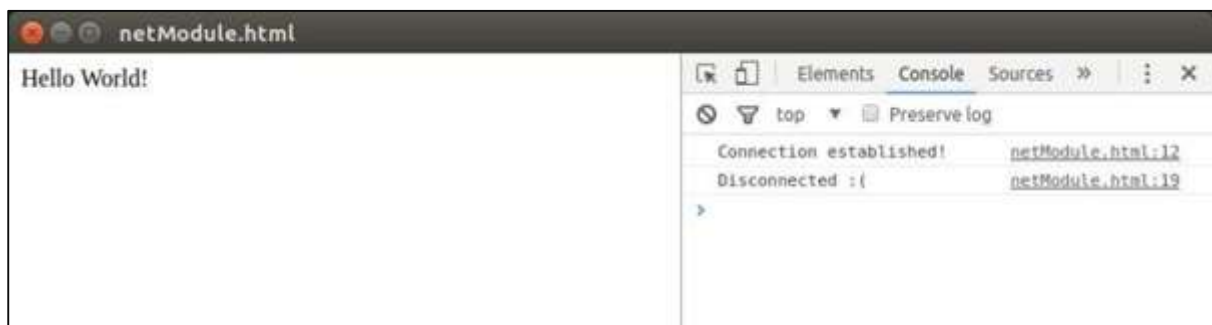
Run the server using the following command:

```
$ node server.js
```

Run the application using the following command:

```
$ electron ./main.js
```

The above command will generate the following output:



Observe that we connect to the server automatically and automatically get disconnected too.

We also have a few other node modules that we can be used directly on the front-end using Electron. The usage of these modules depends on the scenario you use them in.

## 8. Electron – Interprocess Communication

Electron provides us with 2 IPC (Inter Process Communication) modules called **ipcMain** and **ipcRenderer**.

The **ipcMain** module is used to communicate asynchronously from the main process to renderer processes. When used in the main process, the module handles asynchronous and synchronous messages sent from a renderer process (web page). The messages sent from a renderer will be emitted to this module.

The **ipcRenderer** module is used to communicate asynchronously from a renderer process to the main process. It provides a few methods so you can send synchronous and asynchronous messages from the renderer process (web page) to the main process. You can also receive replies from the main process.

We will create a main process and a renderer process that will send each other messages using the above modules.

Create a new file called **main\_process.js** with the following contents:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

// Event handler for asynchronous incoming messages
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg)

  // Event emitter for sending asynchronous messages
  event.sender.send('asynchronous-reply', 'async pong')
```

```

}))

// Event handler for synchronous incoming messages
ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg)

  // Synchronous event emission
  event.returnValue = 'sync pong'
})

app.on('ready', createWindow)

```

Now create a new **index.html** file and add the following code in it.

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <script>
      const {ipcRenderer} = require('electron')

      // Synchronous message emitter and handler
      console.log(ipcRenderer.sendSync('synchronous-message', 'sync ping'))

      // Async message handler
      ipcRenderer.on('asynchronous-reply', (event, arg) => {
        console.log(arg)
      })

      // Async message sender
      ipcRenderer.send('asynchronous-message', 'async ping')
    </script>
  </body>
</html>

```

Run the app using the following command:

```
$ electron ./main_process.js
```

The above command will generate the following output:

```
// On your app console  
Sync Pong  
Async Pong  
  
// On your terminal where you ran the app  
Sync Ping  
Async Ping
```

It is recommended not to perform computation of heavy/ blocking tasks on the renderer process. Always use IPC to delegate these tasks to the main process. This helps in maintaining the pace of your application.



## 9. Electron – System Dialogs

It is very important for any app to be a user-friendly one. As a result you should not create dialog boxes using *alert()* calls. Electron provides a pretty good interface to accomplish the task of creating dialog boxes. Let us have a look at it.

Electron provides a **dialog** module that we can use for displaying native system dialogs for opening and saving files, alerting, etc.

Let us directly jump into an example and create an app to display simple textfiles.

Create a new main.js file and enter the following code in it:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

ipcMain.on('openFile', (event, path) => {
  const {dialog} = require('electron')
  const fs = require('fs')
  dialog.showOpenDialog(function (fileNames) {
    // fileNames is an array that contains all the selected
    if(fileNames === undefined){
      console.log("No file selected");
    }else{
      readFile(fileNames[0]);
    }
  });
});
```

```

function readFile(filepath){
  fs.readFile(filepath, 'utf-8', (err, data) => {
    if(err){
      alert("An error occurred reading the file :" + err.message)
      return
    }
    // handle the file content
    event.sender.send('fileData', data)
  })
}

app.on('ready', createWindow)

```

This code will pop open the open dialog box whenever our main process receives a 'openFile' message from a renderer process. This message will redirect the file content back to the renderer process. Now, we will have to print the content.

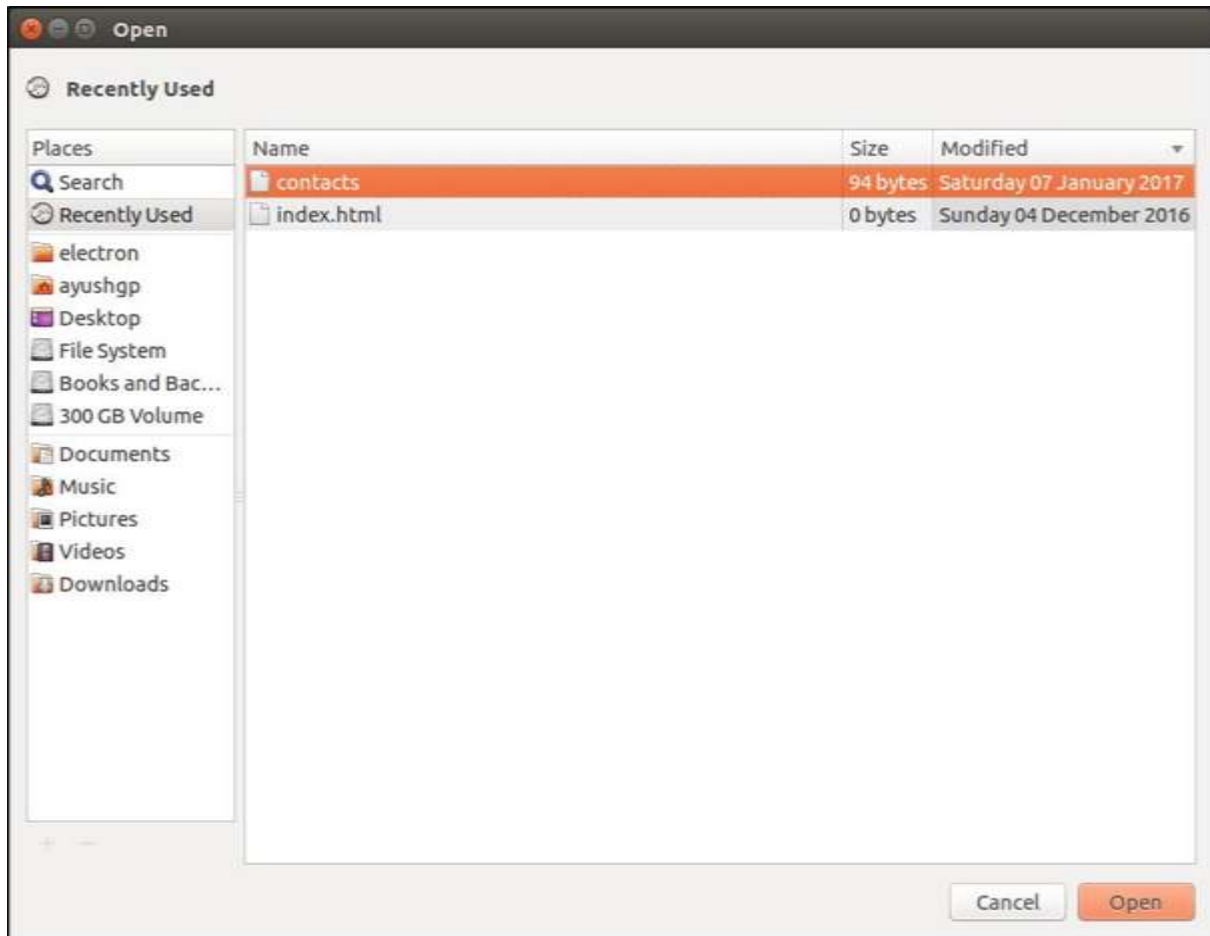
Now, create a new **index.html** file with the following content:

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>File read using system dialogs</title>
  </head>
  <body>
    <script type="text/javascript">
      const {ipcRenderer} = require('electron')
      ipcRenderer.send('openFile', () => {
        console.log("Event sent.");
      })
      ipcRenderer.on('fileData', (event, data) => {
        document.write(data)
      })
    </script>
  </body>
</html>

```

Now whenever we run our app, a native open dialog box will pop up as shown in the following screenshot:



Once we select a file to display, its contents will be displayed on the app window:



This was just one of the four dialogs that Electron provides. They all have similar usage though. Once you learn how to do it using **showOpenDialog**, then you can use any of the other dialogs.

The dialogs having the same functionality are:

- `showSaveDialog([browserWindow, ]options[, callback])`
- `showMessageDialog([browserWindow, ]options[, callback])`
- `showErrorDialog(title, content)`

## 10. Electron – Menus

The desktop apps come with two types of menus – the **application menu** (on the top bar) and a **context menu** (right-click menu). We will learn how to create both of these in this chapter.

We will be using two modules – the *Menu* and the *MenuItem* modules. Note that the *Menu* and the *MenuItem* modules are only available in the main process. For using these modules in the renderer process, you need the *remote* module. We will come across this when we create a context menu.

Now, let us create a new **main.js** file for the main process:

```
const {app, BrowserWindow, Menu, MenuItem} = require('electron')
const url = require('url')
const path = require('path')

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

const template = [
  {
    label: 'Edit',
    submenu: [
      {
        role: 'undo'
      },
      {
        role: 'redo'
      },
      {
        type: 'separator'
      }
    ]
  }
]
```

```
    },  
    {  
      role: 'cut'  
    },  
    {  
      role: 'copy'  
    },  
    {  
      role: 'paste'  
    }  
  ]  
},  
{  
  label: 'View',  
  submenu: [  
    {  
      role: 'reload'  
    },  
    {  
      role: 'toggledevtools'  
    },  
    {  
      type: 'separator'  
    },  
    {  
      role: 'resetzoom'  
    },  
    {  
      role: 'zoomin'  
    },  
    {  
      role: 'zoomout'  
    },  
    {  
      type: 'separator'  
    },  
  ],  
}
```

```
        {
          role: 'togglefullscreen'
        }
      ]
    },
    {
      role: 'window',
      submenu: [
        {
          role: 'minimize'
        },
        {
          role: 'close'
        }
      ]
    },
    {
      role: 'help',
      submenu: [
        {
          label: 'Learn More'
        }
      ]
    }
  ]

  const menu = Menu.buildFromTemplate(template)
  Menu.setApplicationMenu(menu)
```

We are building a menu from a template here. This means that we provide the menu as a JSON to the function and it will take care of the rest. Now we have to set this menu as the Application menu.

Now create an empty HTML file called index.html and run this application using:

```
$ electron ./main.js
```

On the normal position of application menus, you will see a menu based on the above template.



We created this menu from the main process. Let us now create a context menu for our app. We will do this in our HTML file:

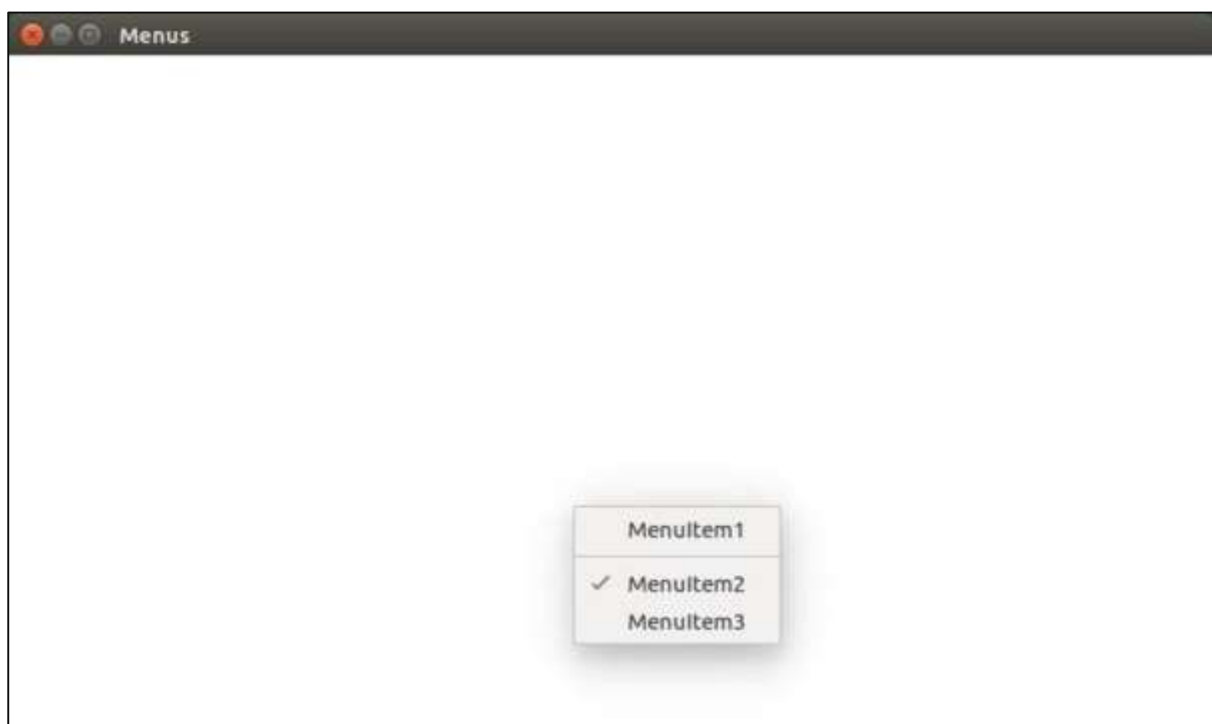
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Menus</title>
  </head>
  <body>
    <script type="text/javascript">
      const {remote} = require('electron')
      const {Menu, MenuItem} = remote

      const menu = new Menu()

      // Build menu one item at a time, unlike
      menu.append(new MenuItem({
        label: 'MenuItem1',
        click() {
          console.log('item 1 clicked')
        }
      }))
      menu.append(new MenuItem({type: 'separator'}))
      menu.append(new MenuItem({label: 'MenuItem2', type: 'checkbox', checked: true}))
      menu.append(new MenuItem({
        label: 'MenuItem3',
        click() {
          console.log('item 3 clicked')
        }
      }))
    </script>
  </body>
</html>
```

```
    }  
  }  
}))  
  
  // Prevent default action of right click in chromium. Replace with our menu.  
  window.addEventListener('contextmenu', (e) => {  
    e.preventDefault()  
    menu.popup(remote.getCurrentWindow())  
  }, false)  
</script>  
</body>  
</html>
```

We imported the Menu and MenuItem modules using the remote module; then, we created a menu and appended our menuitems to it one by one. Further, we prevented the default action of right-click in chromium and replaced it with our menu.



The creation of menus in Electron is a very simple task. Now you can attach your event handlers to these items and handle the events according to your needs.



# 11. Electron – System Tray

System tray is a menu outside of your application window. On MacOS and Ubuntu, it is located on the top right corner of your screen. On Windows it is on the bottom right corner. We can create menus for our application in system trays using Electron.

Create a new **main.js** file and add the following code to it. Have a png file ready to use for the system tray icon.

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

After having set up a basic browser window, we will create a new **index.html** file with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Menus</title>
  </head>
  <body>
    <script type="text/javascript">
      const {remote} = require('electron')
      const {Tray, Menu} = remote
```

```

    const path = require('path')

    let trayIcon = new
    Tray(path.join('', '/home/ayushgp/Desktop/images.png'))

    const trayMenuTemplate = [
      {
        label: 'Empty Application',
        enabled: false
      },
      {
        label: 'Settings',
        click: function () {
          console.log("Clicked on settings")
        }
      },
      {
        label: 'Help',
        click: function () {
          console.log("Clicked on Help")
        }
      }
    ]
    let trayMenu = Menu.buildFromTemplate(trayMenuTemplate)
    trayIcon.setContextMenu(trayMenu)
  </script>
</body>
</html>

```

We created the tray using the Tray submodule. We then created a menu using a template and further attached the menu to our tray object.

Run the application using the following command:

```
$ electron ./main.js
```

When you run the above command, check your system tray for the icon you used. I used a smiley face for my application. The above command will generate the following output:



## 12. Electron – Notifications

Electron provides native notifications API only for MacOS. So we are not going to use that, instead we'll be using a npm module called *node-notifier*. It allows us to notify users on Windows, MacOS and Linux.

Install the node-notifier module in your app folder using the following command in that folder:

```
$ npm install --save node-notifier
```

Let us now create an app that has a button which will generate a notification every time we click on this button.

Create a new **main.js** file and enter the following code in it:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

Let us now create our webpage and script that will trigger the notification. Create a new **index.html** file with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Menus</title>
```

```

</head>
<body>
  <button type="button" id="notify" name="button">Click here to trigger a
notification!</button>
  <script type="text/javascript">
    const notifier = require('node-notifier')
    const path = require('path');
    document.getElementById('notify').onclick = (event) => {
      notifier.notify({
        title: 'My awesome title',
        message: 'Hello from electron, Mr. User!',
        icon: path.join('', '/home/ayushgp/Desktop/images.png'),
        // Absolute path (doesn't work on balloons)
        sound: true, // Only Notification Center or Windows Toasters
        wait: true
        // Wait with callback, until user action is taken against notification
      }, function (err, response) {
        // Response is response from notification
      });

      notifier.on('click', function (notifierObject, options) {
        console.log("You clicked on the notification")
      });

      notifier.on('timeout', function (notifierObject, options) {
        console.log("Notification timed out!")
      });
    }
  </script>
</body>
</html>

```

The **notify** method allows us to pass it an **object** with information like the title, message, thumbnail, etc. which help us customize the notification. We can also set some event listeners on the notification.

Now, run the app using the following command:

```
$ electron ./main.js
```

When you click on the button that we created, you will see a native notification from your operating system as shown in the following screenshot:



We have also handled the events wherein, the user clicks the notification or the notification times out. These methods help us make the app more interactive if its running in the background.

## 13. Electron – Webview

The webview tag is used to embed the 'guest' content like web pages in your Electron app. This content is contained within the webview container. An embedded page within your app controls how this content will be displayed.

The webview runs in a separate process than your app. To ensure security from malicious content, the webview doesn't have same permissions as your web page. This keeps your app safe from the embedded content. All interactions between your app and the embedded page will be asynchronous.

Let us consider an example to understand the embedding of an external webpage in our Electron app. We will embed the tutorialspoint website in our app on the right side. Create a new **main.js** file with the following content:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

Now that we have set up our main process, let us create the HTML file that will embed the tutorialspoint website. Create a file called **index.html** with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Menus</title>
  </head>
```

```
<body>
  <div>
    <div>
      <h2>We have the website embedded below!</h2>
    </div>
    <webview id="foo" src="https://www.tutorialspoint.com/"
style="width:400px; height:480px;">
      <div class="indicator"></div>
    </webview>
  </div>
  <script type="text/javascript">
// Event handlers for loading events.
// Use these to handle loading screens, transitions, etc
onload = () => {
  const webview = document.getElementById('foo')
  const indicator = document.querySelector('.indicator')

  const loadstart = () => {
    indicator.innerText = 'loading...'
  }

  const loadstop = () => {
    indicator.innerText = ''
  }

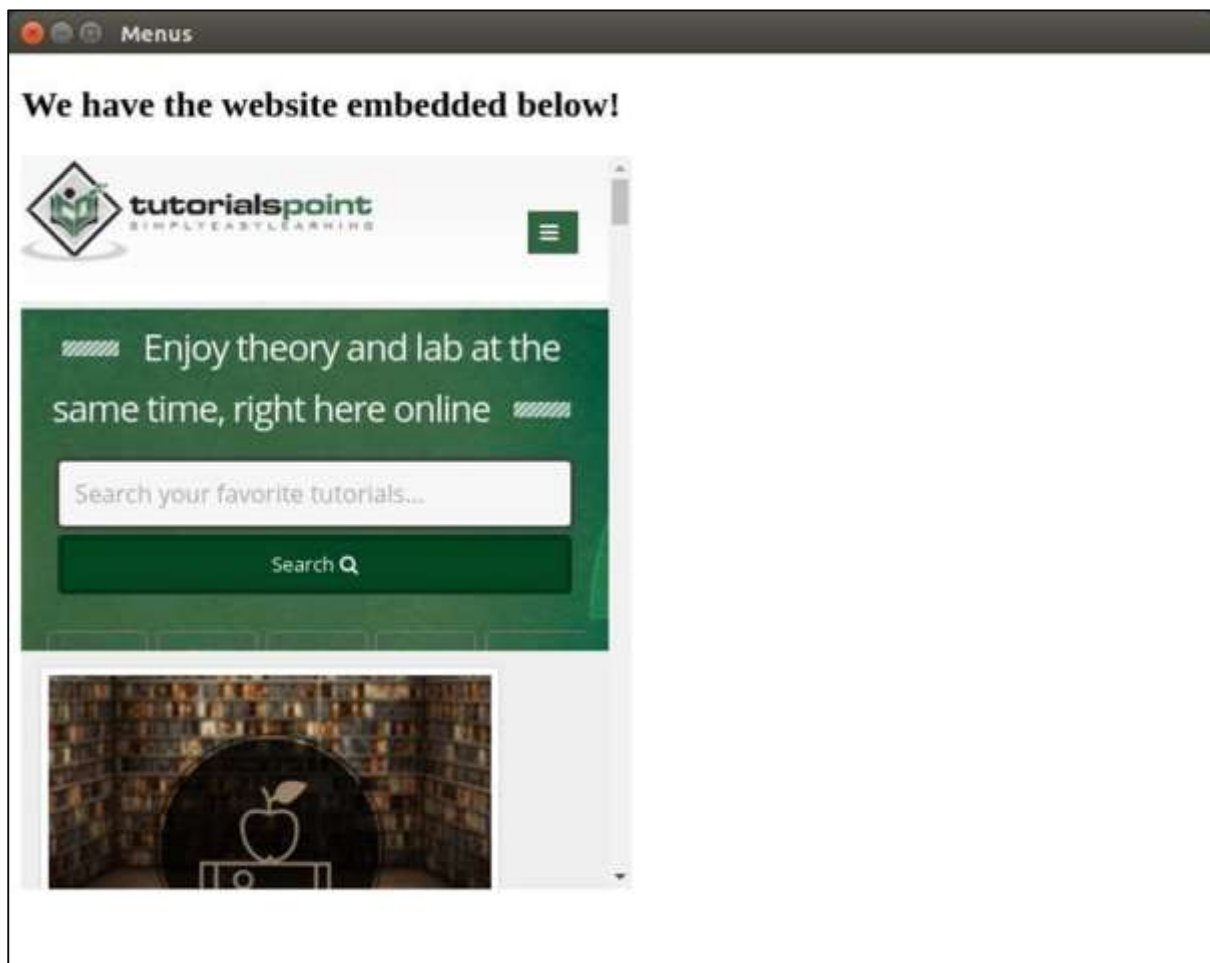
  webview.addEventListener('did-start-loading', loadstart)
  webview.addEventListener('did-stop-loading', loadstop)
}
</script>
</body>
</html>
```

Run the app using the following command:

```
$ electron ./main.js
```



The above command will generate the following output:



The webview tag can be used for other resources as well. The webview element has a list of events that it emits listed on the official docs. You can use these events to improve the functionality depending on the things that take place in the webview.

Whenever you are embedding scripts or other resources from the Internet, it is advisable to use webview. This is recommended as it comes with great security benefits and does not hinder normal behaviour.

# 14. Electron – Audio and Video Capturing

Audio and video capturing are important characteristics if you are building apps for screen sharing, voice memos, etc. They are also useful if you require an application to capture the profile picture.

We will be using the *getUserMedia* HTML5 API for capturing audio and video streams with Electron. Let us first set up our main process in the **main.js** file as follows:

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

let win

// Set the path where recordings will be saved
app.setPath("userData", __dirname + "/saved_recordings")

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

Now that we have set up our main process, let us create the HTML file that will be capturing this content. Create a file called **index.html** with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Audio and Video</title>
  </head>
  <body>
```

```
<video autoplay></video>
<script type="text/javascript">
  function errorCallback(e) {
    console.log('Error', e)
  }

  navigator.getUserMedia({video: true, audio: true}, (localMediaStream) => {
    var video = document.querySelector('video')
    video.src = window.URL.createObjectURL(localMediaStream)
    video.onloadedmetadata = (e) => {
      // Ready to go. Do some stuff.
    };
  }, errorCallback)
</script>
</body>
</html>
```

The above program will generate the following output:



You now have the stream from both your webcam and your microphone. You can send this stream over the network or save this in a format you like.

Have a look at the [MDN Documentation](#) for capturing images to get the images from your webcam and store them. This was done using the HTML5 *getUserMedia* API. You can also capture the user desktop using the *desktopCapturer* module that comes with Electron. Let us now see an example of how to get the screen stream.

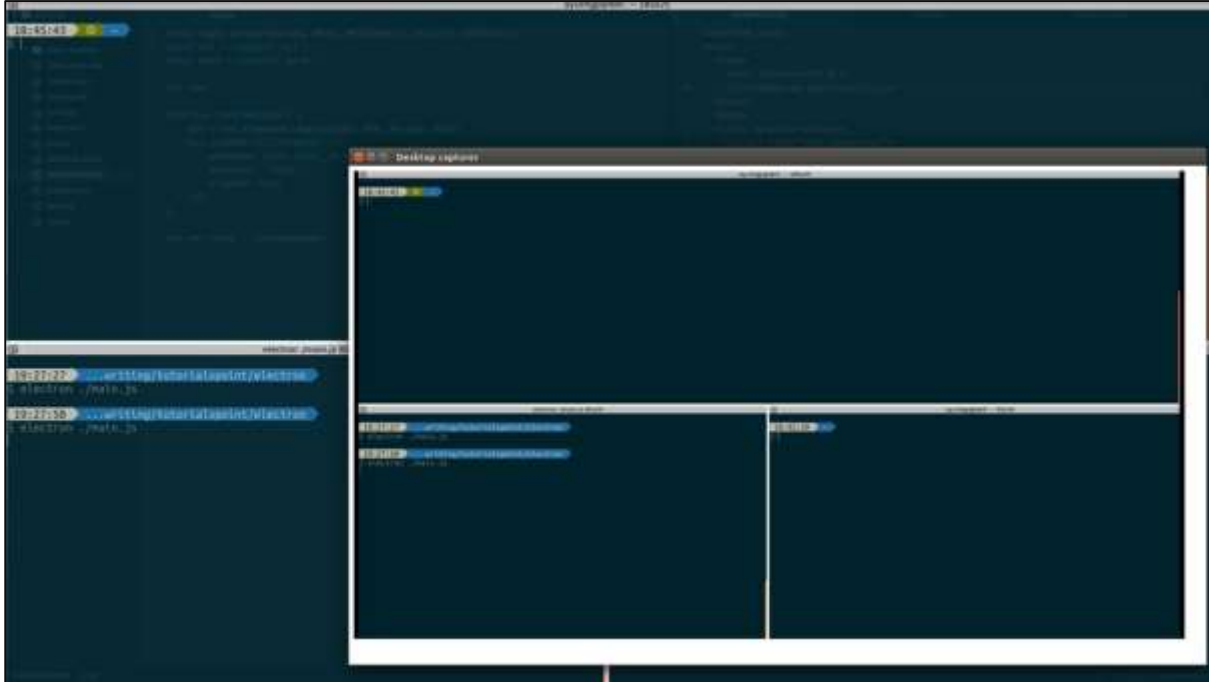
Use the same main.js file as above and edit the index.html file to have the following content:

```
desktopCapturer.getSources({types: ['window', 'screen']}), (error, sources) => {
  if (error) throw error
  for (let i = 0; i < sources.length; ++i) {
    if (sources[i].name === 'Your Window Name here!') {
      navigator.webkitGetUserMedia({
        audio: false,
        video: {
          mandatory: {
            chromeMediaSource: 'desktop',
            chromeMediaSourceId: sources[i].id,
            minWidth: 1280,
            maxWidth: 1280,
            minHeight: 720,
            maxHeight: 720
          }
        }
      }, handleStream, handleError)
      return
    }
  }
})

function handleStream (stream) {
  document.querySelector('video').src = URL.createObjectURL(stream)
}

function handleError (e) {
  console.log(e)
}
```

We have used the *desktopCapturer* module to get the information about each open window. Now you can capture the events of a specific application or of the entire screen depending on the name you pass to the above **if statement**. This will stream only that which is happening on that screen to your app.



You can refer to [this StackOverflow question](#) to understand the usage in detail.

## 15. Electron – Defining Shortcuts

We typically have memorized certain shortcuts for all the apps that we use on our PC daily. To make your applications feel intuitive and easily accessible to the user, you must allow the user to use shortcuts.

We will use the *globalShortcut* module to define shortcuts in our app. Note that **Accelerators** are Strings that can contain multiple modifiers and key codes, combined by the + character. These accelerators are used to define keyboard shortcuts throughout our application.

Let us consider an example and create a shortcut. For this, we will follow the dialog boxes example where we used the open dialog box for opening files. We will register a **CommandOrControl+O** shortcut to bring up the dialog box.

Our **main.js** code will remain the same as before. So create a new **main.js** file and enter the following code in it:

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

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

ipcMain.on('openFile', (event, path) => {
  const {dialog} = require('electron')
  const fs = require('fs')
  dialog.showOpenDialog(function (fileNames) {
    // fileNames is an array that contains all the selected
    if(fileNames === undefined)
      console.log("No file selected")
  })
})
```

```

        else
            readFile(fileNames[0])
    })

    function readFile(filepath){
        fs.readFile(filepath, 'utf-8', (err, data) => {
            if(err){
                alert("An error occurred reading the file :" + err.message)
                return
            }
            // handle the file content
            event.sender.send('fileData', data)
        })
    }
})

app.on('ready', createWindow)

```

This code will pop open the open dialog box whenever our main process receives a 'openFile' message from a renderer process. Earlier this dialog box popped up whenever the app was run. Let us now limit it to open only when we press **CommandOrControl+O**.

Now create a new **index.html** file with the following content:

```

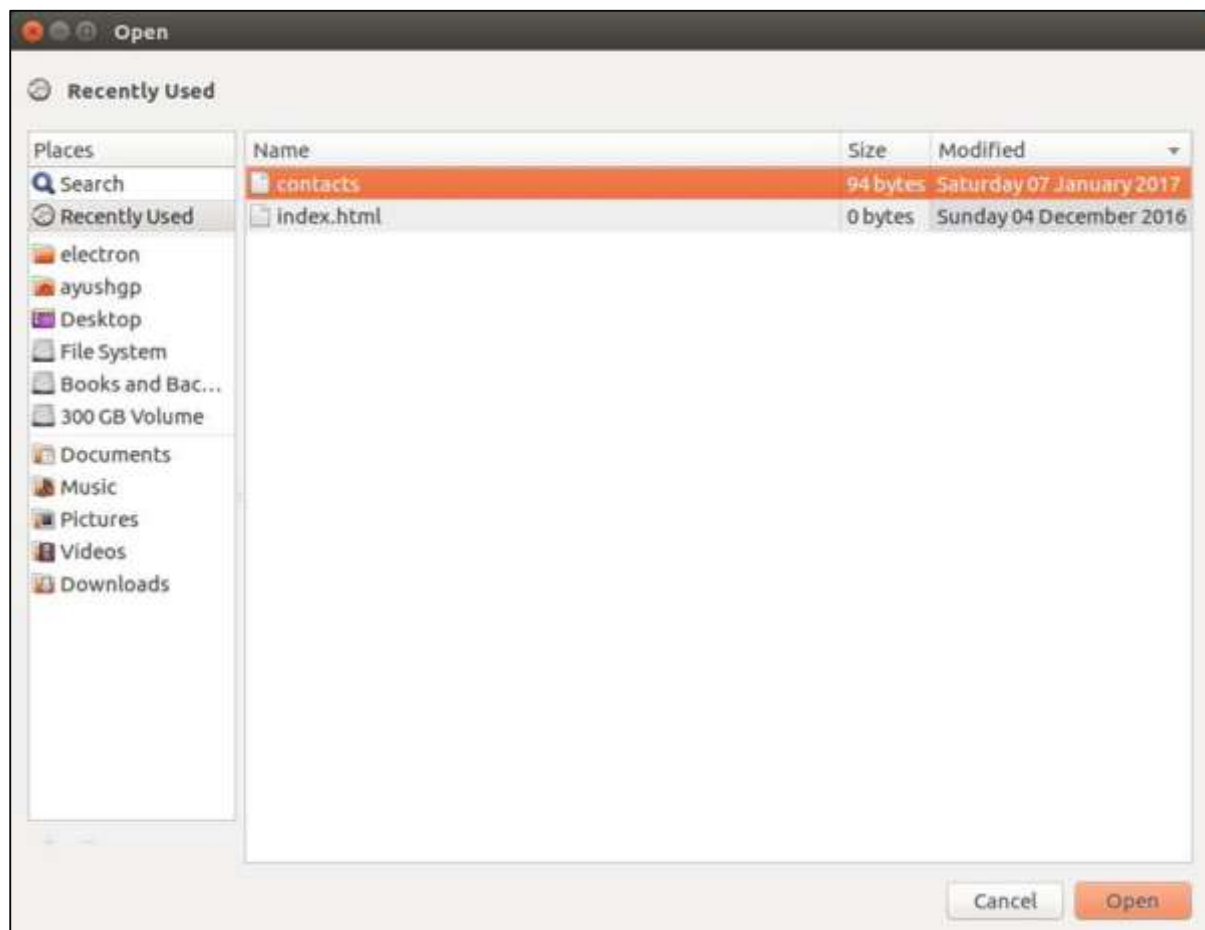
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>File read using system dialogs</title>
  </head>
  <body>
    <p>Press CTRL/CMD + O to open a file. </p>
    <script type="text/javascript">
      const {ipcRenderer, remote} = require('electron')
      const {globalShortcut} = remote
      globalShortcut.register('CommandOrControl+O', () => {
        ipcRenderer.send('openFile', () => {
          console.log("Event sent.");
        })
      })
    </script>
  </body>
</html>

```

```
ipcRenderer.on('fileData', (event, data) => {  
    document.write(data)  
  })  
})  
</script>  
</body>  
</html>
```

We registered a new shortcut and passed a callback that will be executed whenever we press this shortcut. We can deregister shortcuts as and when we do not require them.

Now once the app is opened, we will get the message to open the file using the shortcut we just defined.



These shortcuts can be made customizable by allowing the user to choose his own shortcuts for defined actions.



# 16. Electron – Environment Variables

Environment Variables control application configuration and behavior without changing code. Certain Electron behaviors are controlled by environment variables because they are initialized earlier than the command line flags and the app's code.

There are two kinds of environment variables encoded in electron – **Production variables** and **Development variables**.

## Production Variables

The following environment variables are intended for use at runtime in packaged Electron applications.

Variable	Description
GOOGLE_API_KEY	Electron includes a hardcoded API key for making requests to Google's geocoding webservice. Because this API key is included in every version of Electron, it often exceeds its usage quota.  To work around this, you can supply your own Google API key in the environment. Place the following code in your main process file, before opening any browser windows that will make geocoding requests:  <code>process.env.GOOGLE_API_KEY = 'YOUR_KEY_HERE'</code>
ELECTRON_RUN_AS_NODE	Starts the process as a normal Node.js process.
ELECTRON_FORCE_WINDOW_MENU_BAR (Linux Only)	Do not use the global menu bar on Linux.

## Development Variables

The following environment variables are intended primarily for development and debugging purposes.

Variable	Description
ELECTRON_ENABLE_LOGGING	Prints Chrome's internal logging to the console.
ELECTRON_ENABLE_STACK_DUMPING	Prints the stack trace to the console when Electron crashes.
ELECTRON_DEFAULT_ERROR_MODE	Shows the Windows's crash dialog when Electron crashes.

To set any of these environment variables as true, set it in your console. For example, if you want to enable logging, then use the following commands:

### For Windows:

```
> set ELECTRON_ENABLE_LOGGING=true
```

### For Linux:

```
$ export ELECTRON_ENABLE_LOGGING=true
```

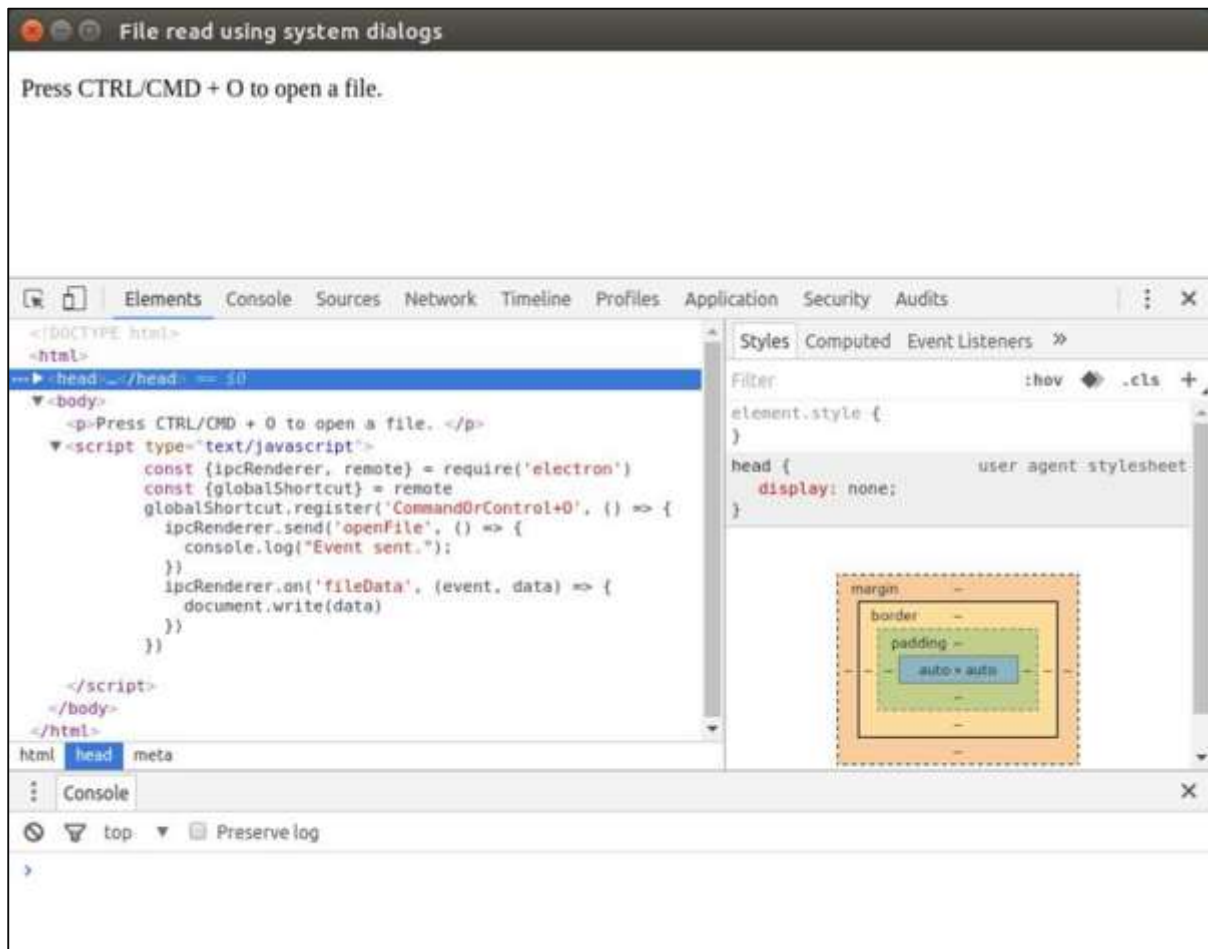
Note that you will need to set these environment variables every time you restart your computer. If you want to avoid doing so, add these lines to your **.bashrc** files.

# 17. Electron – Debugging

We have two processes that run our application – the main process and the renderer process.

Since the renderer process is the one being executed in our browser window, we can use the Chrome DevTools to debug it. To open DevTools, use the shortcut "Ctrl+Shift+I" or the <F12> key. You can check out how to use devtools [here](#).

When you open the DevTools, your app will look like as shown in the following screenshot:



## Debugging the Main Process

The DevTools in an Electron browser window can only debug JavaScript that is executed in that window (i.e., the web pages). To debug JavaScript that is executed in the main process you will need to use an external debugger and launch Electron with the `--debug` or the `--debug-brk` switch.

Electron will listen for the V8 debugger protocol messages on the specified port; an external debugger will need to connect on this port. The default port is 5858.

Run your app using the following:

```
$ electron --debug=5858 ./main.js
```

Now you will need a debugger that supports the V8 debugger protocol. You can use VSCode or node-inspector for this purpose. For example, let us follow these steps and set up VSCode for this purpose. Follow these steps to set it up:

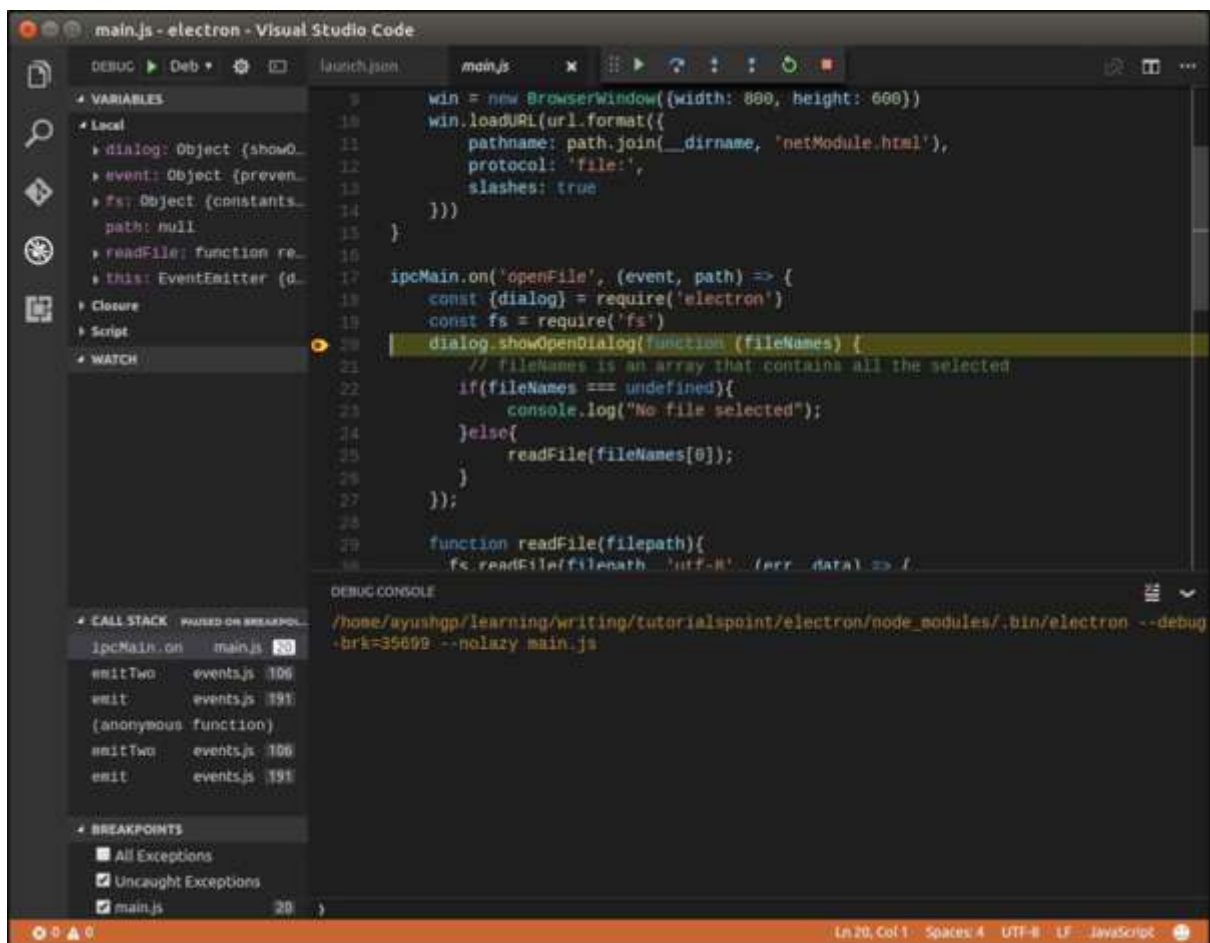
Download and install [VSCode](#). Open your Electron project in VSCode.

Add a file **.vscode/launch.json** with the following configuration:

```
{
  "version": "1.0.0",
  "configurations": [
    {
      "name": "Debug Main Process",
      "type": "node",
      "request": "launch",
      "cwd": "${workspaceRoot}",
      "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/electron",
      "program": "${workspaceRoot}/main.js"
    }
  ]
}
```

**Note:** For Windows, use **"\${workspaceRoot}/node\_modules/.bin/electron.cmd"** for **runtimeExecutable**.

Set some breakpoints in **main.js**, and start debugging in the Debug View. When you hit the breakpoints, the screen will look something like this:



The VSCode debugger is very powerful and will help you rectify errors quickly. You also have other options like **node-inspector** for debugging electron apps.

# 18. Electron – Packaging Apps

Packaging and distributing apps is an integral part of the development process of a desktop application. Since Electron is a cross-platform desktop application development framework, packaging and distribution of apps for all the platforms should also be a seamless experience.

The electron community has created a project, [electron-packager](#) that takes care of the same for us. It allows us to package and distribute our Electron app with OS-specific bundles (.app, .exe etc) via JS or CLI.

## Supported Platforms

---

Electron Packager runs on the following host platforms:

- Windows (32/64 bit)
- OS X
- Linux (x86/x86\_64)

It generates executables/bundles for the following target platforms:

- Windows (also known as win32, for both 32/64 bit)
- OS X (also known as darwin) / Mac App Store (also known as mas)
- Linux (for x86, x86\_64, and armv7l architectures)

## Installation

---

Install the electron packager using:

```
# for use in npm scripts
$ npm install electron-packager --save-dev

# for use from cli
$ npm install electron-packager -g
```

## Packaging Apps

---

In this section, we will see how to run the packager from the command line. The basic form of the command is:

```
electron-packager <sourcedir> <appname> --platform=<platform> --arch=<arch>
[optional flags...]
```

This will:

- Find or download the correct release of Electron.
- Use that version of Electron to create an app in <output-folder>/<appname>-<platform>-<arch>.

**--platform** and **--arch** can be omitted, in two cases. If you specify **--all** instead, bundles for all valid combinations of target platforms/architectures will be created. Otherwise, a single bundle for the host platform/architecture will be created.

# 19. Electron – Resources

We have used the following resources to learn more about Electron. We have referred to these while creating this tutorial.

The most important resource is the [Electron documentation](#). The Documentation has extensive coverage of almost all features and quirks of the framework. They are alone enough to make your way through building an app.

There are also some very good Electron examples presented in the [electron-sample-apps](#) repository.

## Video Resources

[Desktop apps with web languages](#)

[Rapid cross platform desktop app development using JavaScript and Electron](#)

## Blog Posts

[Building a desktop application with Electron](#)

[Build a Music Player with React & Electron](#)

[Creating Your First Desktop App With HTML, JS and Electron](#)

[Create Cross-Platform Desktop Node Apps with Electron](#)