SAMPLE CHAPTER

Cross-Platform Desktop Applications

Using Electron and NW.js

Paul B. Jensen

FOREWORD BY Cheng Zhao

MANNING

Cross-Platform Desktop Applications
Using Electron and NW.js

by Paul Jensen

Chapter 11

brief contents

11 Using a webcam in your application

This chapter covers

- Accessing the webcam on your computer
- Creating still images from live video
- Saving the still images to your computer

Not many years ago, webcams were external devices that you bought to plug into your computer and used to chat with friends and family. Today, almost all laptops come with webcams and microphones built in, making it easy for people to travel and communicate with each other, as long as they have a good internet connection. Back then, the only way you could access a webcam feed was via a desktop app, or by using Adobe Flash. There wasn't an easy way to do it over a web browser.

But that changed. With the introduction of the HTML5 Media Capture API, webcams could be accessible to web pages (with good security procedures in place), and it is this capability that we'll explore in this chapter. We'll look at ways to access and use these APIs to build a photo booth app.

11.1 Photo snapping with the HTML5 Media Capture API

When using Electron or NW.js to build your desktop app, you get the benefit of Google Chrome's extensive support for HTML5 APIs, one of which is the Media

Capture API. The HTML5 Media Capture API allows you to access the microphone and video camera that are embedded in your computer, and the app you'll build will make use of this.

Selfies are powerful—look at Snapchat's IPO valuation ($22 billion as of February 2017). Build an app for selfies, people take selfies, other people view selfies, selfies spawn more selfies, network effects kick in, and suddenly you're a multibillion-dollar startup. Who knew that there was so much money in selfies?

That's why you'll build an app for selfies called Facebomb. Facebomb boils down to *open app, take photo, save photo to your computer.* Simple, usable, and straight to the point. Life is short, so rather that make you build the app from scratch, I've given you an assembled app so you can investigate the particularly interesting bits.

There are two code repositories for the app: one that uses Electron as the desktop app framework, and another that uses NW.js. You'll find them under the names Facebomb-NW.js and Facebomb-Electron in the book's GitHub repository at http://mng.bz/dST8 and http://mng.bz/TX1k.

You can download whichever version of the app you're interested in inspecting, and run the installation instructions for it from the README.md file. Then, you can run the app and see it in action.

11.1.1 Inspecting the NW.js version of the app

A lot of the code is pretty standard boilerplate code for an NW.js app. We'll narrow focus to the index.html and app.js files, which contain code unique to the app, starting with the index.html file.

Listing 11.1 The index.html file for the Facebomb NW.js app

```
<html>
  <head>
    <title>Facebomb</title>
    <link href="app.css" rel="stylesheet" />
    <link rel="stylesheet" href="css/font-awesome.min.css">
    <script src="app.js"></script>
  </head>
  <body>
    <input type="file" nwsaveas="myfacebomb.png" id="saveFile">
    <canvas width="800" height="600"></canvas>
    <video autoplay></video>
    <div id="takePhoto" onclick="takePhoto()">
      <i class="fa fa-camera" aria-hidden="true"></i>
    </div>
  </body>
</html>
```

Triggers the Save File dialog in NW.js

Captures an image from the video

Video feed is streamed into this element

Button that's clicked to trigger taking a photo

The HTML file contains the following:

- An `input` element that's used for the file save. Inside it is a custom NW.js attribute called `nwsaveas` that contains the default filename to save the file as.

- The `canvas` element is used to store the picture data of the photo snapshot you take from the video feed.
- The `video` element will display the video feed from the webcam, which is the source for the photo.
- The `div` element with the id `takePhoto` is the round button in the bottom right of the app window that you'll use to take the photo and save it as a file on the computer. Inside it is a Font Awesome icon for the camera. The advantages of using the camera icon in place of text are that icons use less screen space than words and can be easier to visually process as a result, and if the icon is universally recognizable, you don't need to consider implementing internationalization. Not everyone speaks English—in fact, English is the third-most-commonly spoken language after Mandarin Chinese and Spanish.

Most of this code is compatible with running inside a web browser. The notable element unique to NW.js is the `nwsaveas` attribute (which brings up the Save As dialog for the file) on the `input` element. To read more about this custom attribute, see the docs at http://mng.bz/nU1c.

That covers the index.html file. The app.js file is around 39 lines of code, so we'll look at it in chunks. We'll start with the dependencies and the `bindSavingPhoto` function.

Listing 11.2 The initial code in the app.js file for the Facebomb NW.js app

```
'use strict';

const fs = require('fs');
let photoData;
let saveFile;
let video;

function bindSavingPhoto () {
  saveFile.addEventListener('change', function () {
    const filePath = this.value;
    fs.writeFile(filePath, photoData, 'base64', (err) => {
      if (err) {
        alert('There was a problem saving the photo:', err.message);
      }
      photoData = null;
    });
  });
}
```

Function binds on the input element in the HTML

File path for photo is set by value in input element

Attempts to save file to disk as Base64-encoded image

If error saving the file, displays alert dialog with error message

photoData variable that held photo data reset to null

Here, you require some dependencies, define a few empty variables, and then define a function that's used for binding on when a photo is saved. Inside that function, you add an event listener on the `input` element for when its value changes. When it changes, it's because the Save As dialog has been triggered. When an action is taken to save a photo under a given file name or to cancel it, you attempt to save the photo data to the computer as a Base64-encoded image file. If the file write is successful,

nothing else happens. But if there's an error, you report it to the user in an alert dialog. Finally, you reset the `photoData` variable that was holding the photo snapshot.

Next, we'll look at the `initialize` function in the app.js file.

Listing 11.3 The `initialize` function in the app.js file for the Facebomb NW.js app

```
function initialize () {
  saveFile = window.document.querySelector('#saveFile');
  video = window.document.querySelector('video');

  let errorCallback = (error) => {
    console.log(
      'There was an error connecting to the video stream:', error
    );
  };

  window.navigator.webkitGetUserMedia(
    {video: true},
(localMediaStream) => {
    video.src = window.URL.createObjectURL(localMediaStream);
    video.onloadedmetadata = bindSavingPhoto;
  }, errorCallback);
}
```

initialize function called when app window finishes loading

Creates error-Callback function to handle error on creating video stream

Attaches video stream to video element

Media Capture API request to access video stream from user's computer

Binds on saving photo

If you can't access video stream, then calls the error callback

This bit of code does the key actions of requesting the video stream from the user's media capture device (be it a webcam built into their computer or an external video device) and inserting that video stream into the `video` element in the app window. It also attaches the `bindSavingPhoto` element to the video's `loadedmetadata` event. This event is triggered when the video stream starts to be fed into the `video` element (it usually takes a second or two before the video stream kicks in).

Once you've got the `initialize` function defined, you define the `takePhoto` function that's triggered when the `takePhoto div` element is clicked in the app window. The code for this is shown in the following listing.

Listing 11.4 The `takePhoto` function in the Facebomb NW.js app's app.js file

```
function takePhoto () {
  let canvas = window.document.querySelector('canvas');
  canvas.getContext('2d').drawImage(video, 0, 0, 800, 600);
  photoData = canvas.toDataURL('image/png')
    .replace(/^data:image\/(png|jpg|jpeg);base64,/, '');
  saveFile.click();
}

window.onload = initialize;
```

takePhoto function defined for div element button that's clicked to take photo

canvas element captures image snapshot from video element

photoData variable turns canvas element into Base64-encoded set of data

Binds initialize function to execute when app window has loaded

Triggers Save As dialog programmatically to save photo to computer

Here, the `canvas` element is used to capture an image snapshot from the `video` element. You tell it to use a 2D context and to then draw an image from the video element that begins at 0 pixels left and 0 pixels top, and then goes 800 pixels wide and 600 pixels high. These dimensions mean that you capture the full picture of the video.

You then take the image that has been recorded in the `canvas` element and convert the data format to one for a PNG image. To make the data suitable for saving as a file to the computer, you have to remove a bit of the data that's used to make the image render as an inline image in a web browser. The string `replace` method uses a regular expression to find that bit of data and strip it out.

You programmatically trigger clicking the `input` element that displays the Save As dialog to the user. This means that when the `#takePhoto div` element is clicked in the app window, you'll create an image snapshot from the `video` element at that point in time and then trigger the Save As dialog so that the user can save the image to their computer.

With that function defined, the final bit of code left is to bind the `initialize` function on when the app window has loaded. You do it this way because you want to make sure the app window has finished loading the HTML—otherwise, it will attempt to bind on DOM elements that haven't yet been rendered in the app window, which would cause an error.

With all that code defined in the app.js file, there's a bit of configuration in the package.json file that ensures that the app window is set to 800 pixels wide and 600 pixels high and ensures that the app window cannot be resized or set into full-screen mode. The next listing shows the code for the package.json file.

Listing 11.5 The package.json file for the Facebomb NW.js app

```
{
  "name": "facebomb",
  "version": "1.0.0",
  "main": "index.html",
  "window": {
    "toolbar": false,
    "width": 800,
    "height": 600,
    "resizable": false,
    "fullscreen": false
  },
  "dependencies": {
    "nw": "^0.15.2"
  },
  "scripts": {
    "start": "node_modules/.bin/nw ."
  }
}
```

You also have an app.css file with some styling.

Listing 11.6 The app.css file for the Facebomb NW.js app

```
body {
  margin: 0;
  padding: 0;
  background: black;
  color: white;
  font-family: 'Helvetica', 'Arial', 'Sans';
  width: 800px;
  height: 600px;
}

#saveFile, canvas {
  display: none;
}

video {
  z-index: 1;
  position: absolute;
  width: 800px;
  height: 600px;
}

#takePhoto {
  z-index: 2;
  position: absolute;
  bottom: 5%;
  right: 5%;
  text-align: center;
  border: solid 2px white;
  box-shadow: 0px 0px 7px rgba(255,255,255,0.5);
  margin: 5px;
  border-radius: 3em;
  padding: 1em;
  background-color: rgba(0,0,0,0.2);
}

#takePhoto:hover {
  background: #FF5C5C;
  cursor: pointer;
}
```

Now, you can look at what the app would look like when it's run. Figure 11.1 shows an example of the app running on Windows 10.

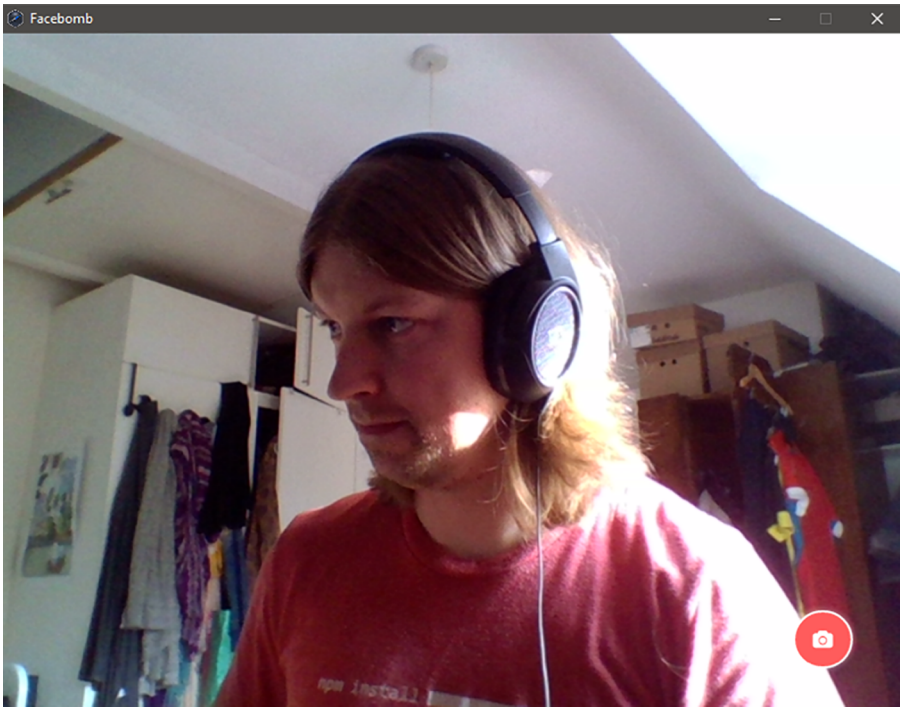

Figure 11.1 Facebomb in action (that's me by the way—I could use a shave)

Why isn't the app asking for permission to use the camera?

The HTML5 Media Capture API has a security policy of asking users if they want a web page to be allowed to access their camera or microphone before the web app can use them. This is to prevent malicious use of the camera or microphone to take photos or record audio.

With Electron and NW.js, because the app is running on the user's computer, the app is trusted with access to the computer's devices, so there's no permission bar appearing in the app. This means you can create apps that have direct access to the camera and microphone, but as Peter Parker's (Spider-Man's) uncle said, "With great power comes great responsibility."

With the app, you can take a photo of yourself and the file is saved to the computer. Nice and simple—but the key thing here is that it demonstrates how easy it is to build an app that takes in the camera feed and can do all kinds of things with it.

That shows how you can do it with NW.js, but what about Electron?

11.1.2 Creating Facebomb with Electron

If you want to have the cake and eat it straightaway, you can grab the Facebomb-Electron app from the book's GitHub repository. I'll walk you through the differences of Electron's approach to implementing Facebomb. First, as expected, the entry point of the app differs from NW.js—you have a main.js file that handles the responsibility of loading the app window and applying constraints to it so it can't be resized or enter full-screen mode. Other differences with Electron are in how it implements the Save As dialog, as well as the level of customization you can apply to the dialog.

You'll take a look first at the entry point of the app to see how the constraints are applied to the app window. The following listing shows the code for the main.js file.

Listing 11.7 The main.js file for the Facebomb Electron app

```
'use strict';

const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

let mainWindow = null;

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') app.quit();
});

app.on('ready', () => {
  mainWindow = new BrowserWindow({
    useContentSize: true,
    width: 800,
    height: 600,
    resizable: false,
    fullscreen: false
  });
  mainWindow.loadURL(`file://${__dirname}/index.html`);
  mainWindow.on('closed', () => { mainWindow = null; });
});
```

Requires Electron; loads app and browser window dependencies

Creates empty mainWindow variable to hold app window reference

If all windows are closed and you're not running app on Mac OS, quits app

Creates browser window with width, height, resizable, and full-screen properties

Gets main app window to load index.html file inside it

Adds event binding to reset mainWindow variable when window is closed

This is pretty much standard boilerplate for an Electron app, but the key bit of interest is the configuration object that's passed into the initialization of the `BrowserWindow` instance.

The first property passed in the configuration object is called `useContentSize`. It ensures that the `width` and `height` properties of the app window are referring to the content of the app window and not to the entire app window. If you don't pass this property (or explicitly set it to `false`), you'll see scrollbars appear in the app window. This is because Electron treats the `width` and `height` properties as referring to not

only the app window's content size, but also the title bar at the top of the app window, as well as any trim around the edges of the app window.

If you didn't pass this, you would otherwise have to tweak the `width` and `height` properties to make sure that the app window didn't have any scrollbars. This is the kind of pixel pushing that you don't want to have to deal with—plus, if your app is running across multiple OSs, you would have to tweak these numbers for each build you want to target. Not ideal. I recommend you always pass the `useContentSize` attribute if you're going to define `width` and `height` properties to your app windows. For more on this attribute and other options that can be passed to the window configuration, see http://electron.atom.io/docs/api/browser-window/.

You also pass the options for disabling the ability to resize the window or make it allow full-screen mode here. Whereas in NW.js these options are configured in the package.json file, Electron passes the configuration at the point of creating the app window. The advantage of this approach is that it's easier to give separate app windows different configurations rather than inherit the same configuration from the package.json file.

Now, take a quick look at the index.html file.

Listing 11.8 The index.html file for the Facebomb Electron app

```
<html>
  <head>
    <title>Facebomb</title>
    <link href="app.css" rel="stylesheet" />
    <link rel="stylesheet" href="css/font-awesome.min.css">
    <script src="app.js"></script>
  </head>
  <body>
    <canvas width="800" height="600"></canvas>
    <video autoplay></video>
    <div id="takePhoto" onclick="takePhoto()">
      <i class="fa fa-camera" aria-hidden="true"></i>
    </div>
  </body>
</html>
```

The index.html file that's loaded for the app window is almost identical to the one used in the NW.js variant. The only difference is that there's no `input` element in the Electron version, and that's because it's not needed. If you remember, the `input` element was used for storing the filename for the photo, as well as containing the custom attribute `nwsaveas`, which NW.js uses to bind a Save File dialog.

Electron handles dialog windows differently than NW.js, and to see how differently, you need to take a look at the app.js file. The app.js file is around 40 lines of code, so we'll scan through it bit by bit, starting with the dependencies and the alternative to the `bindSavingPhoto` function.

Listing 11.9 The dependencies in the app.js file for the Facebomb Electron app

```
'use strict';

const electron = require('electron');
const dialog = electron.remote.dialog;
const fs = require('fs');
let photoData;
let video;

function savePhoto (filePath) {
  if (filePath) {
    fs.writeFile(filePath, photoData, 'base64', (err) => {
      if (err) {
        alert(`There was a problem saving the photo: ${err.message}`);
      }
      photoData = null;
    });
  }
}
```

Loads Electron and requires dialog module through remote API

savePhoto function receives file path from Save File dialog

Checks for file path in case user clicked Cancel on Save File dialog

In the dependencies at the top of the app.js file, you require Electron and then use the remote API to load Electron's dialog module from a renderer process (the app.js file). You then define a function called `savePhoto`. The purpose of this function is to save the photo to disk when a file path is passed to it from Electron's Save File dialog. If it manages to successfully save the file to disk, you're good, but if it encounters an error, you alert the user. You also reset the `photoData` variable afterward.

Let's look at the `initialize` function in the app.js file.

Listing 11.10 The app.js file's `initialize` function for the Facebomb Electron app

```
function initialize () {
  video = window.document.querySelector('video');
  let errorCallback = (error) => {
    console.log(`There was an error connecting to the video stream:
     ${error.message}`);
  };

  window.navigator.webkitGetUserMedia({video: true}, (localMediaStream) => {
    video.src = window.URL.createObjectURL(localMediaStream);
  }, errorCallback);
}
```

This code is almost identical to the same-named function in the NW.js variant, but with a slight difference: you don't need to define a `saveFile` variable as there is no `input` element in the HTML, and you don't need to bind on the video's `loadedmetadata` event triggering, because you pass the data and file in another location in the app's code.

Finally, let's take a look at the `takePhoto` function and the `window.onload` event binding that makes up the rest of the app.js file.

Listing 11.11 The app.js file's `takePhoto` function for the Facebomb Electron app

```
function takePhoto () {
  let canvas = window.document.querySelector('canvas');
  canvas.getContext('2d').drawImage(video, 0, 0, 800, 600);
  photoData =
     canvas.toDataURL('image/png').replace(/^data:image\/(png|jpg|jpeg);base6
     4,/, '');
  dialog.showSaveDialog({
    title: "Save the photo",
    defaultPath: 'myfacebomb.png',
    buttonLabel: 'Save photo'
  }, savePhoto);
}

window.onload = initialize;
```

Calls dialog module to create Save File dialog

Sets title of Save File dialog window

Passes default filename for the file

Sets label of success action button to "Save photo"

Passes savePhoto function as callback to dialog, which will pass final file path

In this version of the app, the `takePhoto` function does a bit more work. It directly triggers the rendering of the Save File dialog window. You set the title, default file path, and Success button's labels, and then pass the `savePhoto` function as the callback function that the dialog window will call once the user has either clicked Save Photo or Cancel on the dialog window. When the `savePhoto` function is called, it will receive the file path with the name of the file given by the user, or it will receive a null value if the user cancelled. Last but not least, you bind the `initialize` function on triggering when the window has loaded the HTML.

Here, you can see that to bring about a dialog window for saving a file, you call a function in Electron's dialog module. The `showSaveDialog` function is one of a number of functions you can call from the module. If you want to trigger other behaviors, like a dialog for opening a file or displaying a message dialog with an icon, the API methods and their arguments are available at http://electron.atom.io/docs/api/dialog/.

What does the Electron version of the app look like? It's almost identical to the NW.js version, as figure 11.2 shows.

The key takeaway here is that you've been able to build an app with embedded video and photo-saving features. Imagine the effort involved in trying to replicate the same app in native frameworks! It's fair to say that HTML5 Media Capture has taken away a lot of the pain, so the ability to build desktop apps on top of that kind of work is a massive timesaver.

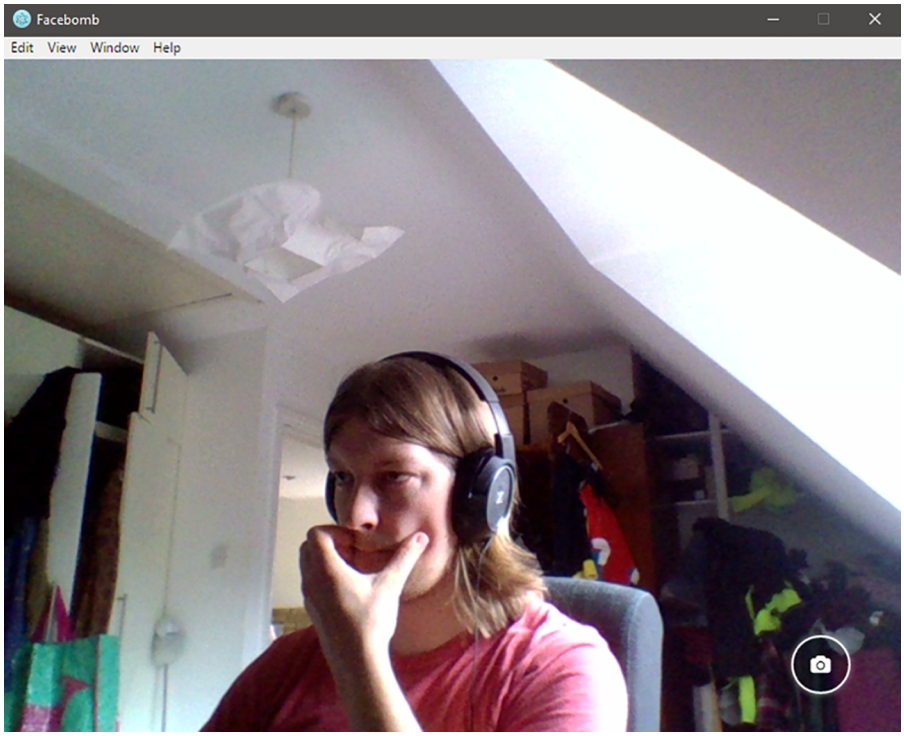

Figure 11.2 Facebomb Electron on Windows 10. Notice how the app looks exactly the same, except for the app icon in the app title.

11.2 Summary

In this chapter, you created a photo booth–like app called Facebomb and explored different implementations of it in NW.js and Electron. This discussion has introduced you to the idea that you can leverage the HTML5 Media Capture API to access video and use it in creative ways. Some of the key takeaways from the chapter include these:

- You don't need to worry about asking for permission to access the webcam or microphone when using HTML5 Media Capture APIs, because both Electron and NW.js apps run locally on the user's computer and are therefore trusted.
- You can use the `video` element to display the video feed in your app, and the HTML5 `canvas` element to record an image from it to be saved to your computer.

That was fun. In chapter 12, we'll turn our attention to ways of storing app data.

JAVASCRIPT/WEB DEVELOPMENT

Cross-Platform Desktop Applications

Paul B. Jensen

Desktop application development has traditionally required high-level programming languages and specialized frameworks. With Electron and NW.js, you can apply your existing web dev skills to create desktop applications using only HTML, CSS, and JavaScript. And those applications will work across Windows, Mac, and Linux, radically reducing development and training time.

Cross-Platform Desktop Applications guides you step by step through the development of desktop applications using Electron and NW.js. This example-filled guide shows you how to create your own file explorer, and then steps through some of the APIs provided by the frameworks to work with the camera, access the clipboard, make a game with keyboard controls, and build a Twitter desktop notification tool. You'll then learn how to test your applications, and debug and package them as binaries for various OSs.

What's Inside

- Create a selfie app with the desktop camera
- Learn how to test Electron apps with Devtron
- Learn how to use Node.js with your application

Written for developers familiar with HTML, CSS, and JavaScript.

Paul Jensen works at Starcount and lives in London, UK.

"You will be shocked by how easy it is to write a desktop app!"
—From the Foreword by Cheng Zhao, Creator of Electron

"Write-once/run-anywhere just became a real thing."
—Stephen Byrne, Dell

"The definitive guide to two paradigm-shifting JavaScript frameworks. Indispensable."
—Clive Harber, Distorted Thinking

"Packed full of examples that will help you write cross-platform desktop apps using JavaScript."
—Jeff Smith, Ascension

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/cross-platform-desktop-applications

ISBN-13: 978-1-61729-284-2
ISBN-10: 1-61729-284-2
54999
9 781617 292842

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]