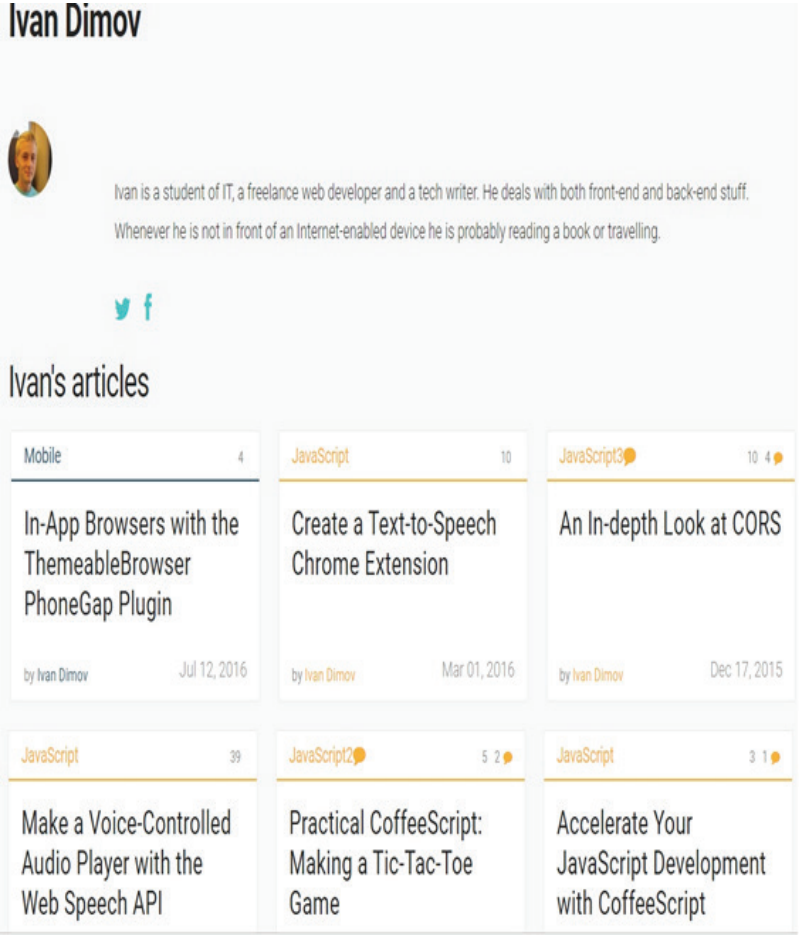
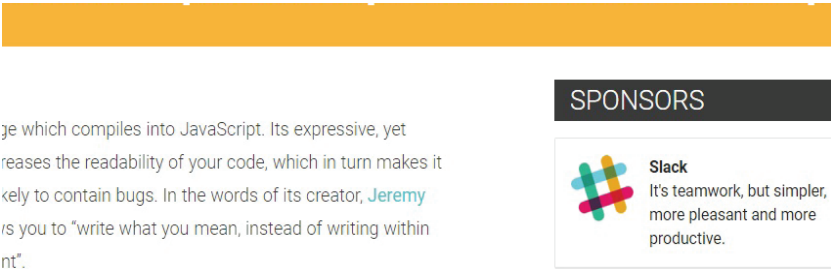
Ivan Dimov
Articles in SitePoint

Articles by SitePoint
copyright©2019 Ivan Dimov
© Издател – Фондация „Буквите“

ISBN 978-619-154-328-1


Ivan Dimov
Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.
Ivan's articles
Mobile 4
In-App Browsers with the ThemeableBrowser PhoneGap Plugin
by Ivan Dimov Jul 12, 2016
JavaScript 10
Create a Text-to-Speech Chrome Extension
by Ivan Dimov Mar 01, 2016
JavaScript3 10 4
An In-depth Look at CORS
by Ivan Dimov Dec 17, 2015
JavaScript 39
Make a Voice-Controlled Audio Player with the Web Speech API
JavaScript2 5 2
Practical CoffeeScript: Making a Tic-Tac-Toe Game
JavaScript 3 1
Accelerate Your JavaScript Development with CoffeeScript

Accelerate Your JavaScript Development with Coffee-Script



CoffeeScript is a tiny language which compiles into JavaScript. Its expressive, yet terse syntax dramatically increases the readability of your code, which in turn makes it easier to maintain and less likely to contain bugs. In the words of its creator, Jeremy Ashkenas, CoffeeScript allows you to "write what you mean, instead of writing within the limits of historical accident".

An additional benefit to writing CoffeeScript is that the JavaScript it compiles to will run in older versions of Internet Explorer. CoffeeScript also lets you forget about common JS pitfalls, such as trailing commas and automatic semicolon insertion.

And it's gaining in popularity! Spurred on by its adoption in the Rails community (Rails 3.1+ ships with built-in CoffeeScript support), CoffeeScript recently entered the Tiobe index of the top 100 programming languages, where it was raked in 64th place. This was ahead of Dart (66th place) and TypeScript (not listed), both of which also compile to JavaScript.

So are you ready to give CoffeeScript a try? In this article I will demonstrate how to install it, as well as its basic concepts.

Installation

You can install CoffeeScript globally using the Node Package Manager (npm) by typing the following command in to your terminal:

```
npm install coffee-script -g
```

You should install it globally so you can later access it in terminal with the command coffee.

If you need a primer on using npm, then please refer to this recently published SitePoint article.

Compilation

CoffeeScript files have the .coffee file extension. These files are either manually compiled, or you set a watcher that will compile your script each time it is saved with different contents.

To compile manually, go to the directory where the script is:

```
cd E:\apps\something\logic
```

And run the following command:

```
coffee  -c app.coffee
```

This will create an app.js file in the same directory which you can then include in your project.

However, you most likely want app.js to be refreshed each time you save the file. Therefore you compile it and add a watcher by typing:

```
coffee -cw app.coffee
```

Please note that in the latest version of CoffeeScript (1.9.1) there is a bug that causes the watcher not to work. All of the following examples were tested using CoffeeScript v 1.9.0.

CoffeeScript Basics

In CoffeeScript you do not have to declare variables as you do in JavaScript, although often you would need to set an initial value. We also do not have to type semi-colons (;) at the end of a line.

This means that you write:

```
hasBody = true
```

instead of :

```
var hasBody = true;
```

You can also call functions without using parentheses, but that is desirable only for outermost function calls. Therefore, you can do the following:

```
$(„.messages“) .show ‚slow‘
```

instead of:

```
$(„.messages“).show(‚slow‘);
```

Indentation matters a lot in CoffeeScript. You should indent with two spaces or a tab:

```
if hasBody
  alert „Hello Body“
else
  alert „No Body“
```

Booleans and Conditionals

In CoffeeScript the keywords on, yes and true are all equivalent and refer to the Boolean true, whereas off, no and false are also equivalent and refer to the Boolean false.

You can use is and isnt to check for equality or lack of equality (=== and !==) . You can use then to make single-line conditional statements. You can use and and or to refer to && and || respectively. All of which means that you can compare a value to two other values without repeating yourself.

An example of these concepts:

```
x = 53
y = 40
z = 33
b = true
if b is on and x is 53 and z < y > 11 then alert „ALRIGHT!“
```

The final statement compiles to:

```
if (b === true && x === 53 && (z < y && y > 11)) {
  alert(‚ALRIGHT!‘);
}
```

Iteration, Filters and Ranges

The for .. in syntax in CoffeeScript is used for iterating over an array, while a for .. of loop is used for iterating over the properties of an object.

```
arr = [1, 2, 3]

for val in arr
  console.log(val);

spartacus =
  type: “cat”
  legs: 4
  fur: yes

for key, value of spartacus
  console.log “#{key}: #{value}”
```

Notice the string interpolation in the final statement. This is achieved using a #{variableName} syntax.

This will output:

```
1
2
3
type: cat
legs: 4
fur: true
```

You can combine this with CoffeeScript's when keyword to filter items in an array:

```
spartacus =
  type: „cat“
  legs: 4
  fur: yes

shrimpy =
  type: “fish”
  legs: 0
  fur: no

pets = [spartacus, shrimpy]
myPet = pet for pet in pets when pet.type is “cat”
console.log myPet
```

Outputs:

```
Object {type: „cat“, legs: 4, fur: true}
```

Notice that you can specify the statement to be executed in the loop before you write the loop. This is useful when writing one-liner loops.

This could also be written as:

```
for pet in pets when pet.type is „cat“
  myPet = pet
  console.log myPet
```

An extremely useful feature of CoffeeScript is the ability to create numeric ranges. These can be inclusive and exclusive:

```
someValues = [0..10]
sameValues = [0...11]
```

When compiled to JavaScript the arrays look like this:

```
someValues = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
sameValues = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

If you create a range with more than 20 elements, CoffeeScript gets a little clever, as you can see from the generated JavaScript. Notice that execution takes place in an anonymous function to ward against scope leakage and variable conflict.

```
var someValues, _i, _results;
someValues = (function() {
  _results = [];
  for (_i = 0; _i <= 21; _i++){ _results.push(_i); }
  return _results;
}).apply(this);
```

Functions and the "this" Keyword

To create functions you use -> followed by the definition. If you have to add parameters you add them inside parenthesis () before the ->

You can add default values by setting the parameter to equal something.

Examples of functions with CoffeeScript:

```
myCoffee = ->
  console.log „C‘est un cafe“

makeCoffee = (brand = “Hogwarts”) ->
  console.log “Making a coffee #{brand}-style”
myCoffee()
makeCoffee()
```

Logs to the console:

```
C‘est un cafe
Making a coffee Hogwarts-style
```

You can use @ instead of the this keyword:

```
$(„a“).click ->
  $(@).hide ‚slow‘
```

This compiles to:

```
$(„a“).click(function() {
  return $(this).hide(‚slow‘);
});
```

ES6 Style Classes and OOP

CoffeeScript facilitates object-oriented programming and inheritance as well. You can define classes like this:

```
class Animal
  constructor: (@name, @breed) ->
    @.introduce = ->
        console.log „Hi, I am #{@.name}, one hell of a
#{@.breed}“

husky = new Animal(“Johnny”, “Siberian Husky”)
husky.introduce()
```

Outputs in console:

```
Hi, I am Johnny, one hell of a Siberian Husky
```

typing @ before the parameter's name in the constructor, causes the parameter to be immediately set in the constructor. Alternatively, you could just write @.name = namein the constructor function.

You can extend classes as well:

```
class Dog extends Animal


Dog.prototype.bark = ->
    console.log “#{@name} barks!”

newHusky = new Dog(“Lassy”, “Labrador Husky”)
newHusky.introduce()
newHusky.bark()
```

This code outputs:

```
Hi, I am Lassy, one hell of a Labrador Husky
Lassy barks!
```

Your functions can accept an unlimited number of arguments if you add ...(ellipsis) after the parameter. In this case, all values after and including that parameter are added in the form of an array.

Here is how you could achieve that:

```
showAwards = (awards...) ->
  console.log („Awards collected : #{awards.join ,, ,}“)
showAwards “Award 1”, “Award 2”, “Award 3”
```

The code above outputs:

```
Awards collected : Award 1, Award 2, Award 3
```

The final thing I want to mention here is that when you are inside any function CoffeeScript automatically returns the result from the last statement executed. Therefore, all your functions have an implicit return value (just as you saw in the anonymous function that handles anchor clicks above).

Conclusion

In this article I have demonstrated many of the features that make CoffeeScript such a pleasure to work with. I will build on this knowledge in a future article when I will use CoffeeScript to create the well-known game TicTacToe. Until then, let me know what you think: Are you using CoffeeScript already? Has this article made you curious to try it out? Or does CoffeeScript offer nothing more than an unnecessary layer of complexity?



Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article Source: https://www.sitepoint.com/accelerate-javascript-development-coffeescript/

Practical CoffeeScript: Making a Tic-Tac-Toe Game



Practical CoffeeScript: Making a Tic-Tac-Toe Game

CoffeeScript is a tiny little language that compiles to JavaScript. There is no interpretation at runtime since you write CoffeeScript, compile it to JavaScript and use the resulting JavaScript files for your app. You can use any JavaScript library (e.g. jQuery) from within CoffeeScript, just by using its features with the appropriate CoffeeScript syntax. CoffeeScript can be used both for writing JavaScript on the front-end and JavaScript on the back-end.

So Why CoffeeScript?

SPONSORS

Бъдете бизнесът, кой
хората намират, дока
търсят в Google.

Google Ads

CoffeeScript is a tiny little language that compiles to JavaScript. There is no interpretation at runtime since you write CoffeeScript, compile it to JavaScript and use the resulting JavaScript files for your app. You can use any JavaScript library (e.g. jQuery) from within CoffeeScript, just by using its features with the appropriate CoffeeScript syntax. CoffeeScript can be used both for writing JavaScript on the front-end and JavaScript on the back-end.

So Why CoffeeScript?

Less Code

According to the Little Book on CoffeeScript, CoffeeScript's syntax reduces the amount of characters you need to type to get your JS working by around 33% to 50%. I will be presenting a simple Tic-Tac-Toe game created using CoffeeScript (you probably already guessed this from the title) which in its raw CoffeeScript format contains 4963 characters, whereas the compiled JavaScript code contains 7669 characters. That is a difference of 2706 characters or 36%!

Faster Development Time

Because you write shorter, less error-prone (e.g. variables are auto-scoped, meaning you can't accidentally overwrite globals by omitting var) you can finish your

projects quicker. CoffeeScript's terse syntax also makes for more readable code, and ultimately code which is easier to maintain.

Getting Started

In this article, we will be building a simple Tic-tac-toe game with CoffeeScript and jQuery. If you want to read up on the syntax before examining a practical case, I suggest my Accelerate Your JavaScript Development with CoffeeScript article here at SitePoint. This also details how to install CoffeeScript via npm (the Node Package manager).

As ever, all of the code from this tutorial is available on GitHub and a demo is available on CodePen or at the end of the tutorial.

The most common CoffeeScript commands you will be using are:

coffee -c fileName will compile the CoffeeScript file to a file with the same name but with a .js extension (CoffeeScript files typically have .coffee extension).

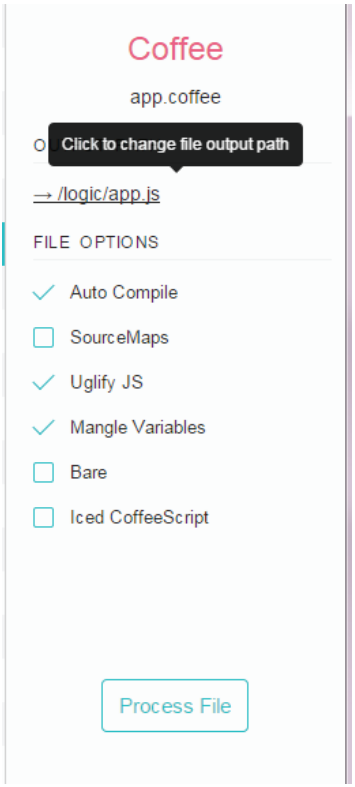
coffee -cw fileName will watch for changes in a file (whenever you save the file) and compile it.

coffee -cw folderName/ will watch for changes to all .coffee files in the folder and compile them in the same directory when there are any changes.

Finally, it is handy to compile CoffeeScript from a folder with .coffee files to a folder containing only .js files.

coffee -o js/ -cw /coffee will watch for changes in all .coffee files located in the coffee folder and place the output (JavaScript) in the js folder.

If you are not into terminals, you can use a tool with a GUI to handle your CoffeeScript files. For instance, you can try Prepros on a free unlimited trial (although you have to buy it if you like it). The image below shows some of the options it provides:



You can see that Prepros does all the work for you—it sets up watchers so your .coffee files will be compiled to JS, it allows you to use Uglify JS which will minify/compress your code, it can automatically mangle variables and it supports Iced CoffeeScript. Prepros can also be used for CSS preprocessors such as Less and Sass and template engines like Jade.

The Game

Let's start with the markup:

```
<div class="wrapper">
  <header class="text-center">
    <h1>Tic Tac Toe</h1>
  </header>

  <div id="board"></div>
  <div class="alerts welcome"></div>
  <div class="notifications"></div>

  <form action="" method="POST">
    ...
  </form>
</div>
<script src="jquery.min.js"></script>
<script src="logic/app.js"></script>
```

The game's interface consists of the following:

- A header which briefly describes the game
- A div element with the id of board which is where the 3×3 squares will be located
- A div element with a class of alerts which is where the game status will be shown
- A div element with a class of notifications which will show who is playing X and O, along with the general player statistics.
- A form which will be displayed only when the game loads and will prompt the players to enter their names.

In accordance with best practice, both jQuery and the script that makes our app tick are loaded before the closing body tag.

The Styling

Using CSS, we can make the nine squares involved appear in a 3×3 grid by floating each square and clearing every 4th one.

```
.square:nth-of-type(3n + 1) {
  clear: both;
}
```

We can also add a different color to the squares depending on whether they have the class x or o (which is added using JavaScript).

```
.square.x {
  color: crimson;
}

.square.o {
  color: #3997ff;
}
```

CoffeeScript in Action

For reference, you can find the main CoffeeScript file here.

You can see our Tic-Tac-Toe app starts with $ ->, this is equivalent to the shorthand for jQuery's function that executes code when the DOM is ready: $(function() { ... });.

CoffeeScript does not rely on semicolons and braces but on indentation. -> tells CoffeeScript that you are defining a function so you can start the body of the function on the next line and indent the body with two spaces.

Next, we create an object called Tic which itself contains an object called data. You can see that braces

or commas are not obligatory when creating objects, as long as you indent the properties correctly.

```
$ ->
  Tic =
    data:
      turns: 0
      x: {}
      o: {}
      gameOver: false
```

The turns property will hold the total number of turns taken in the game. We can check whether it holds an even or uneven number and in that way determine whether it is the turn of X or O.

The x and o properties are objects and will contain data relating to the number of X's or O's on the three axes that are important for the game: horizontal, vertical and diagonal. They will be updated on every move through the checkEnd method to represent the distribution of X and O on the board. The checkEnd method will then call checkWin to determine if there is a winner.

After that we have a method inside the Tic object that will get everything up and running:

```
initialize: ->
  @data.gameOver = false
  @.setPlayerNames()
  @.retrieveStats()
  @.assignRoles()
  @.prepareBoard()
  @.updateNotifications()
  @.addListeners()
```

Notice the use of @ which compiles to the JavaScript keyword this. As illustrated in the first property of initialize, you can skip the dot after the @ keyword when setting or calling a property or method.

By giving the methods sensible names, we have a fair idea of what they are doing:

- setPlayerNames stores the values entered by users in the inputs into the dataobject.
- retrieveStats retrieves the player's statistics from localStorage and sets them up in the data object.
- assignRoles determines who is playing X and who is playing O.
- prepareBoard hides the form, removes any notifications, empties the board and fills it with nine empty squares.

- updateNotifications updates the UI with information about who is playing X and who is playing O, as well as the player's statistics.
- addListeners attaches the event listeners, so that we can respond to players making a move.

Diving Deeper

Let's look at a couple of these methods in more detail.

```
prepareBoard: ->
  ...
    $(„<div>“, {class: „square“}).appendTo(„#board“)
for square in [0..8]
```

Here we iterate nine times and add nine divs with a class of square to the empty board in order to populate it. This demonstrates how CoffeeScript lets you write one-line loops and declare the body of the loop before writing the condition itself.

```
updateNotifications: ->
  $(„.notifications“).empty().show()
  @.addNotification „#{@data.player1} is playing #{@
data.rolep1}“
  ...
```

CoffeeScript allows for string interpolation which increases readability and reduces complexity and code length. You can add a #{} within any string and insert any variable or return value from a function call within the braces.

```
addNotification: (msg) ->
  $(„.notifications“).append($(„<p>“, text: msg));
```

The addNotification method exemplifies how you define parameters in CoffeeScript. You write them before the arrow (->):

You can provide default values for parameters similar to PHP:

```
addNotification: (msg = „I am a message“) ->
```

When a function with a default parameter is compiled, it is converted to:

```
if (msg == null) { msg = „I am a message“; }
```

Finally, let's turn to the addListeners method:

```
addListeners: ->
  $(„.square“).click ->
    if Tic.data.gameOver is no and not $(@).text().length
        if Tic.data.turns % 2 is 0 then $(@).html(„X“).
addClass(„x moved“)
    else if Tic.data.turns % 2 isnt 0 then $(@).html(„O“).
addClass(„o moved“)
      ...
```

Here we see that CoffeeScript offers additional keywords to represent truthy and falsy values such as no, yes, off and on. Additionally, !==, ===, &&, ! can be represented using isnt, is , and and not accordingly.

You can make readable single line conditionals using if ... then ... else ...syntax.

The Mechanics of the Game

The workhorse method checkEnd checks if there is a winner every time a player makes a move. It does this by iterating over the board and counting the squares that belong to X and O. It first checks the diagonal axes, then the vertical, then the horizontal.

```
checkEnd : ->
  @.data.x = {}
  @.data.o = {}

  #diagonal check
  diagonals = [[0,4,8], [2,4,6]]
  for diagonal in diagonals
    for col in diagonal
      @.checkField(col, 'diagonal')
    @.checkWin()
    @.emptyStorageVar('diagonal')
  for row in [0..2]
    start = row * 3
    end = (row * 3) + 2
    middle = (row * 3) + 1

    #vertical check
    @checkField(start, 'start')
```

```
  @checkField(middle, 'middle')
  @checkField(end, 'end')
  @checkWin()

  #horizontal check
  for column in [start..end]
    @checkField(column, 'horizontal')
  @checkWin()
  @emptyStorageVar('horizontal')
```

As you can see, this makes use of another handy CoffeeScript feature—ranges.

```
for row in [0..2]
```

This will loop three times, setting row equal to 0, 1 and 2 in this order. Alternatively, [0...2] (an exclusive range) would result in just two iterations, setting row equal to 0 and 1.

In the horizontal check we see again how indentation is crucial in determining what is part of the loop and what is outside of the loop—only the checkField call is inside the inner loop.

This is what checkField looks like:

```
checkField: (field, storageVar) ->
  if $(„.square“).eq(field).hasClass(„x“)
  if@.data.x[storageVar]?then@.data.x[storageVar]++
else @.data.x[storageVar] = 1
    else if $(„.square“).eq(field).hasClass(„o“)
  if@.data.o[storageVar]?then@.data.o[storageVar]++
else @.data.o[storageVar] = 1
```

This method demonstrates the use of the ? keyword, which when inserted next to a variable in a conditional, compiles to:

```
if (typeof someVariable !== „undefined“ &&
someVariable !== null) {
```

Which is obviously quite handy.

What the checkField method does is add one to the appropriate axis of the x or oproperty depending on the class name of the square which was clicked. The class name is added when a user clicks on an empty board square in the addListenersmethod.

This brings us on to the checkWin method, which is used to check if one of the players has won the game:

```
checkWin: ->
  for key,value of @.data.x
    if value >= 3
      localStorage.x++
      @showAlert „#{@.getPlayerName(„X“)} wins“
      @data.gameOver = true
      @addToScore(„X“)
  for key,value of @.data.o
    if value >= 3
      localStorage.o++
      @showAlert „#{@.getPlayerName(„O“)} wins“
      @data.gameOver = true
      @addToScore(„O“)
```

In CoffeeScript you can use for ... in array to loop over array values and for key,value of object to loop over the properties of an object. checkWin utilizes this to check all the properties inside the x and o objects. If any of them holds a number greater than or equal to three, then we have a winner and the game should end. In such a case, we call the addToScore method which persists the results of the players through localStorage.

A Word about Local Storage

LocalStorage is part of the Web Storage specification and has a pretty good browser support. It allows you to store data (similar to cookies) on the user's machine and access it whenever you want.

You can access the API in several ways, for example just as you would to the properties of a regular object:

```
//fetch item
localStorage.myProperty

// set item
localStorage.myProperty = 123
```

Local storage always saves strings so if you want to store an object or an array you would have to use JSON.stringify when storing the array/object and JSON.parsewhen retrieving it.

Our addToScore method utilizes this fact:

```
addToScore: (winningParty) ->
  ...
  if winningParty is „none“
    @.showAlert „The game was a tie“
  else
    ...
      localStorage[@data.player1] = JSON.stringify @data.p1stats
```

It also demonstrates how you can omit parentheses in CoffeeScript (JSON.stringify), although that is recommended for the outermost function calls only.

Next we have a couple of utility methods. We use emptyStorageVar to clear the contents of a particular horizontal row or diagonal. This is necessary because there are two diagonals on the board and inside our chekEnd method we use the same data property for both diagonals. Therefore, we have to clear the property before checking the second diagonal. The same goes for the horizontal rows.

```
emptyStorageVar: (storageVar) ->
    @.data.x[storageVar] = null
    @.data.o[storageVar] = null
```

Getting the Player Names

When the form with the names of the players is submitted at the beginning of a game, we can prevent its default action and handle the submission using JavaScript. We check if there is an empty name or if both names are the same and display a friendly alert if so. Otherwise, we start the game by calling Tic.initialize().

```
$(„form“).on „submit“, (evt) ->
```

```
  evt.preventDefault()
  $inputs = $(„input[type='text']“)
  namesNotEntered = $inputs.filter(->
    return @.value.trim() isnt „“
  ).length isnt 2
    namesIndentical = $inputs[0].value is $inputs[1].value
     if namesNotEntered then Tic.showAlert(„Player names cannot be empty“)
   else if namesIndentical then Tic.showAlert(„Player names cannot be identical“)
  else Tic.initialize()
```

The final line uses event delegation to have any element with the class play-againrespond to a click. Event delegation is necessary, as this element is only added to a page once a game has finished. It is not present when the DOM is first rendered.

```
$(„body“).on(„click“, „.play-again“, -> Tic.initialize())
```

Putting it all Together

And that's it. In less than 150 lines of CoffeeScript we have a working game. Don't forget, you can download the code from this tutorial from GitHub.

Conclusion

I hope that this tutorial has solidified your knowledge of CoffeeScript and has shown you how jQuery and CoffeeScript can work together. There are many things that you can do to improve the game. For example you could add an option to make the board different than its standard 3×3 dimensions. You could implement some simple AI so that players can play against the machine, or you could implement bombs in the game, e.g. by adding a random X or O on a random game move while the players are battling for glory.



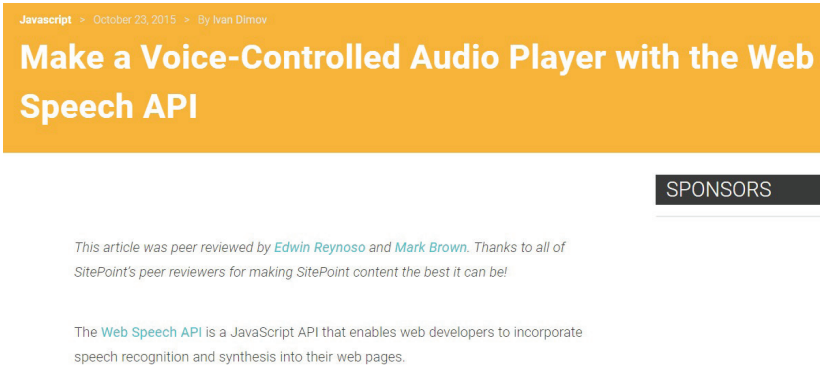
Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article source: https://www.sitepoint.com/practical-coffeescript-tic-tac-toe/

Make a Voice-Controlled Audio Player with the Web Speech API



Javascript > October 23, 2015 > By Ivan Dimov

Make a Voice-Controlled Audio Player with the Web Speech API

SPONSORS

This article was peer reviewed by Edwin Reynoso and Mark Brown. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

The Web Speech API is a JavaScript API that enables web developers to incorporate speech recognition and synthesis into their web pages.

This article was peer reviewed by Edwin Reynoso and Mark Brown. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

The Web Speech API is a JavaScript API that enables web developers to incorporate speech recognition and synthesis into their web pages.

There are many reasons to do this. For example, to enhance the experience of people with disabilities (particularly users with sight problems, or users with limited ability to move their hands), or to allows users to interact with a web app while performing a different task (such as driving).

If you have never heard of the Web Speech API, or you would like a quick primer, then it might be a good idea to read Aurelio De Rosa's articles Introducing the Web Speech API, Speech Sythesis API and the Talking Form .

Browser Support

Browsers vendors have only recently started implementing both the Speech Recognition API and the Speech Synthesis API. As you can see, support for these is still far from perfect, so if you are following along with this tutorial, please use an appropriate browser.

In addition, the speech recognition API currently requires an Internet connection, as the speech gets passed

through the wire and the results are returned to the browser. If the connection uses HTTP, the user has to permit a site to use their microphone on every request. If the connection uses HTTPS, then this is only necessary once.

Speech Recognition Libraries

Libraries can help us manage complexity and can ensure we stay forward compatible. For example when another browser starts supporting the Speech Recognition API, we would not have to worry about adding vendor prefixes.

One such library is Annyang, which is incredibly easy to work with. *Tell me more*.

Voice-controlled Audio Player

In this article, we will be building a voice-controlled audio player. We will be using both the Speech Synthesis API (to inform users which song is beginning, or that a command was not recognized) and the Speech Recognition API (to convert voice commands to strings which will trigger different app logic).

The great thing about an audio player that uses the Web Speech API is that users will be able to surf to other pages in their browser or minimize the browser and do something else while still being able to switch between

songs. If we have a lot of songs in the playlist, we could even request a particular song without searching for it manually (if we know its name or singer, of course).

We will not be relying on a third-party library for the speech recognition as we want to show how to work with the API without adding extra dependencies in our projects. The voice-controlled audio player will only be supporting browsers that support the interimResults attribute. The latest version of Chrome should be a safe bet.

As ever, you can find the complete code on GitHub, and a demo on CodePen.

Getting Started — a Playlist

Let's start with a static playlist. It consists of an object with different songs in an array. Each song is a new object containing the path to the file, the singer's name and the name of the song:

```
var data = {
  „songs“: [
    {
      „fileName“: „https://www.ruse-problem.org/songs/RunningWaters.mp3“,
```

```
    „singer“ : „Jason Shaw“,
    „songName“ : „Running Waters“
  },
  ...
```

We should be able to add a new objects to the songs array and have the new song automatically included into our audio player.

The Audio Player

Now we come to the player itself. This will be an object containing the following things:

- some setup data
- methods pertaining to the UI (e.g. populating the list of songs)
- methods pertaining to the Speech API (e.g. recognizing and processing commands)
- methods pertaining to the manipulation of audio (e.g. play, pause, stop, prev, next)

Setup Data

This is relatively straight forward.

```
var audioPlayer = {
  audioData: {
    currentSong: -1,
    songs: []
  },
```

The currentSong property refers to the index of the song that the user is currently on. This is useful, for example, when we have to play the next/previous song, or stop/pause the song.

The songs array contains all the songs that the user has listened to. This means that the next time the user listens to the same song, we can load it from the array and not have to download it.

You can see the full code here.

UI Methods

The UI will consist of a list of available commands, a list of available tracks and a context box to inform the user of both the current operation and the previous command. I won't go into the UI methods in detail,

rather offer a brief overview. You can find the code for these methods here.

load

This iterates over our previously declared playlist and appends the name of the song, as well as the name of the artist to a list of available tracks.

changeCurrentSongEffect

This indicates which song is currently playing (by marking it green and adding a pair of headphones next to it) as well as those which have finished playing.

playSong

This indicates to the user that a song is playing, or when it has ended. It does this via the changeStatus-Code method, which adds this information to the box and to inform the user of this change via the Speech API.

changeStatusCode

As mentioned above, this updates the status message in the context box (e.g. to indicate that a new song is playing) and utilizes the speak method to announce this change to the user.

changeLastCommand

A small helper which updates the last command box.

toggleSpinner

A small helper to hide or show the spinner icon (which indicates to the user that his voice command is currently processing).

Player Methods

The player will be responsible for what you might expect, namely: starting, stopping and pausing playback, as well as moving backwards and forwards through the tracks. Again, I don't want to go into the methods in detail, but would rather point you towards our GitHub repo.

Play

This checks if the user has listened to a song yet. If not, it starts the song, otherwise it just calls the playSong method we discussed previously on the currently cached song. This is located in audioData.songs and corresponds to the currentSongindex.

pauseSong

This pauses or completely stops (returns playback time to the song's beginning) a song, depending on what is passed as the second parameter. It also updates the status code to notify the user that the song has either been stopped or paused.

stop

This either pauses or stops the song based on its first and only parameter:

prev

This checks whether the previous song is cached and if so, it pauses the current song, decrements currentSong and plays the current song again. If the new song is not in the array, it does the same but it first loads the song from the file name/path corresponding to the decremented currentSong index.

next

If the user has listened to a song before, this method tries to pause it. If there is a next song in our data object (i.e. our playlist) it loads it and plays it. If there is no next song it just changes the status code and informs the user that they have reached the final song.

searchSpecificSong

This takes a keyword as an argument and performs a linear search across song names and artists, before playing the first match.

Speech API Methods

The Speech API is surprisingly easy to implement. In fact, it only takes two lines of code to get a web app talking to users:

```
var utterance = new SpeechSynthesisUtterance(‚Hello‘);
window.speechSynthesis.speak(utterance);
```

What we are doing here is creating an utterance object which contains the text we wish to be spoken. The speechSynthesis interface (which is available on the window object) is responsible for processing this utterance object and controlling the playback of the resulting speech.

Go ahead and try it out in your browser. It’s that easy!

speak

We can see this in action in our speak method, which reads aloud the message passed as an argument:

```
speak: function(text, scope) {
  var message = new SpeechSynthesisUtterance(text.
replace(„-“, „ „));
 message.rate = 1;
 window.speechSynthesis.speak(message);
 if (scope) {
  message.onend = function() {
   scope.play();
  }
 }
}
```

If there is a second argument (scope), we call the play method on scope (which would be an Audio object) after the message has finished playing.

processCommands

This method is not as exciting. It receives a command as a parameter and calls the appropriate method to respond to it. It checks if the user wants to play a specific song with a regular expression, otherwise, it enters a switch statement to test different commands. If none corresponds to the command received, it informs the user that the command was not understood.

You can find the code for it here.

Tying Things Together

By now we have a data object representing our playlist, as well as an audioPlayerobject representing the player itself. Now we need to write some code to recognize and deal with user input. Please note that this will only work in webkit browsers.

The code to have users talk to your app is equally as simple as before:

```
var recognition = new webkitSpeechRecognition();
recognition.onresult = function(event) {
  console.log(event)
}
recognition.start();
```

This will invite the user to allow a page access to their microphone. If you allow access you can start talking and when you stop the onresult event will be fired, making the results of the speech capture available as a JavaScript object.

Reference: The HTML5 Speech Recognition API

We can implement this in our app as follows:

```
if (window[‚webkitSpeechRecognition‘]) {
        var speechRecognizer = new webkitSpeechRecognition();

  // Recognition will not end when user stops speaking
  speechRecognizer.continuous = true;

  // Process the request while the user is speaking
  speechRecognizer.interimResults = true;

  // Account for accent
  speechRecognizer.lang = “en-US”;

  speechRecognizer.onresult = function (evt) { ... }
  speechRecognizer.onend = function () { ... }
  speechRecognizer.start();
} else {
  alert(“Your browser does not support the Web Speech API”);
}
```

As you can see we test for the presence of webkit-SpeechRecognition on the window object. If it is there, then we’re good to go, otherwise we inform the user that the browser doesn’t support it. If all’s good, we then set

a couple of options. Of these lang is an interesting one which can improve the results of the recognition, based on where you hail from.

We then declare handlers for the onresult and the onend events, before kicking things off with the start method.

Handling a result

There are a few things we want to do when the speech recognizer gets a result, at least in the context of the current implementation of speech recognition and our needs. Each time there is a result, we want to save it in an array and set a timeout to wait for three seconds, so the browser can collect any further results. After the thee seconds are up, we want to use the gathered results and loop over them in reverse order (newer results have better chance of being accurate) and check whether the recognized transcript contains one of our available commands. If it does, we execute the command and restart the speech recognition. We do this because waiting for a final result can take up to a minute, making our audio player seem quite unresponsive and pointless, as it would be faster to just click on a button.

```
speechRecognizer.onresult = function (evt) {
  audioPlayer.toggleSpinner(true);
  results.push(evt.results);
  if (!timeoutSet) {
    setTimeout(function() {
      timeoutSet = false;
      results.reverse();
      try {
        results.forEach(function (val, i) {
          var el = val[0][0].transcript.toLowerCase();
            if (currentCommands.indexOf(el.split(„ „)[0])
!== -1) {
            speechRecognizer.abort();
            audioPlayer.processCommands(el);
            audioPlayer.toggleSpinner();
            results = [];
            throw new BreakLoopException;
          }
          if (i === 0) {
            audioPlayer.processCommands(el);
            speechRecognizer.abort();
            audioPlayer.toggleSpinner();
            results = [];
          }
        });
```

```
    }
    catch(e) {return e;}
  }, 3000)
 }
 timeoutSet = true;
}
```

As we are not using a library we have to write more code to set up our speech recognizer, looping over each result and checking if its transcript matches a given keyword.

Lastly, we restart speech recognition as soon as it ends:

```
speechRecognizer.onend = function () {
 speechRecognizer.start();
}
```

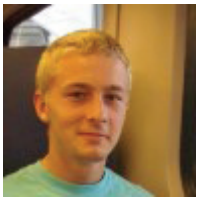
You can see the full code for this section here.
And that's it. We now have an audio player which is fully functional and voice-controlled. I urge to to download the code from Github and have a play about with it, or check out the CodePen demo. I have also made available a version which is served via HTTPS.

Conclusion

I hope this practical tutorial has served as a healthy introduction as to what is possible with the Web Speech API. I think that we will see use of this API grow, as implementations stabilize and new features are added. For example, I see a YouTube of the future which is completely voice-controlled, where we can watch the videos of different users, play specific songs and move between songs just with voice commands.

There are also many other areas where the Web Speech API could bring improvements, or open new possibilities. For example browsing email, navigating websites, or searching the web — all with your voice.

Are you using this API in your projects? I'd love to hear from you in the comments below.



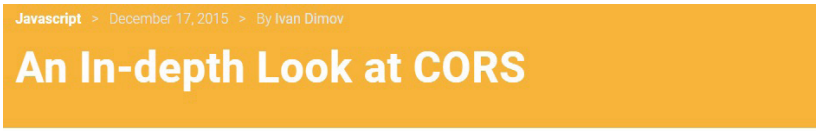
Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article source: https://www.sitepoint.com/voice-controlled-audio-player-web-speech-api/

An In-depth Look at CORS



This article was peer reviewed by Panayiotis «pvgr» Velisarakos. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

CORS is a relatively new API that came with HTML5 which allows our websites to request external and previously restricted resources. It relaxes the traditional same-origin policy by enabling us to request resources that are on a different domain than our parent page.

For example, before CORS cross-domain, Ajax requests were not possible (making an Ajax call from the page `example.com/index.html` to `anotherExample.com/index.html`).

This article was peer reviewed by Panayiotis «pvgr» Velisarakos. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

CORS is a relatively new API that came with HTML5 which allows our websites to request external and previously restricted resources. It relaxes the traditional same-origin policy by enabling us to request resources that are on a different domain than our parent page.

For example, before CORS cross-domain, Ajax requests were not possible (making an Ajax call from the page example.com/index.html to anotherExample.com/index.html).

In this article we'll see how to use CORS to further interact with other systems and websites in order to create even better Web experiences. Before exploring CORS more, let's first have a look at which browsers support it.

CORS and Browser Support

Internet Explorer 8 and 9 support CORS only through the XDomainRequest class. The main difference is that instead of doing a normal instantiation with something like var xhr = new XMLHttpRequest() you would have to use var xdr = new XDomainRequest();.

IE 11, Edge and all recent and not-really-recent versions of Firefox, Safari, Chrome, Opera fully support CORS. IE10 and Android's default browser up to 4.3 only lack support for CORS when used for images in <canvas> elements.

According to CanIuse, 92.61% of people globally have supporting browsers which indicates that we are likely not going to make a mistake if we use it.

Making a Simple Cross-Origin Ajax Request

Now that we know that the same-origin policy prohibits websites in different domains from making Ajax requests to other domains, let's see how we can bypass this in order to make a cross-origin Ajax request to another website.

If you simply try to shoot an Ajax request to a random website, it would most likely not be able to read the response unless that another website allows it.

```
<script>
   var xhr = new XMLHttpRequest();
   var url = „//example.com“;
   xhr.open(„GET“, url);
   xhr.onreadystatechange = function() {

      if (xhr.status === 200 && xhr.readyState === 4) {

            document.querySelector(“body”).innerHTML
= xhr.responseText
      }
   }
   xhr.send();
</script>
```

If you open your browser's console, you would get a message similar to: *XMLHttpRequest cannot load http://example.com. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://otherExampleSite.com' is therefore not allowed access.*

To successfully read the response, you would have to set a header called **Access-Control-Allow-Origin**. This header has to be set either in your application's back-end logic (setting the header manually before the response

is delivered to the client) or in your server's configuration (such as editing apache.conf and adding Header set Access-Control-Allow-Origin «*» to it, if you are using Apache).

Adding the header with the meta tag in your document's <head> tag like this would **not** work: <meta http-equiv=»Access-Control-Allow-Origin» content=»*»>

Here is how you can enable cross-origin requests for all origins (sites that request the resource) in PHP:

```
```

When making cross-origin requests, the destination website has to be the one who has your origin enabled and allows you to read the response from the request.

If you want to allow a specific origin you can do something like this in PHP:

```
header(„Access-Control-Allow-Origin: http://example.com“);
```

However, the Access-Control-Allow-Origin header itself does not allow multiple hosts inserted in the

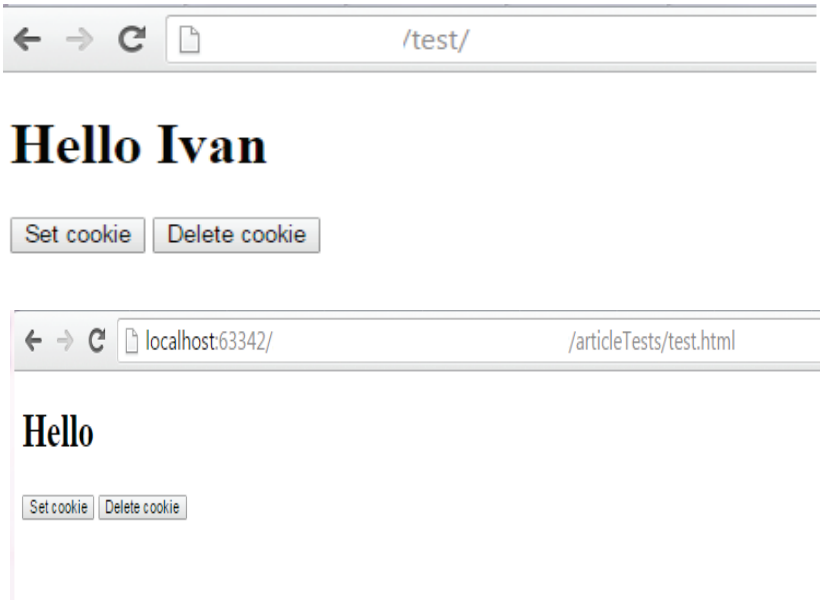
header, no matter the delimiter. This means that if you want to allow a cross-origin request from various different domains, you have to dynamically generate your header.

For example, in PHP you can check the origin of the website requesting your resource and if it matches a particular whitelist, add a header that allows that specific origin to make a cross-origin request. Here is a tiny example with a hardcoded whitelist:

Some safety is maintained in the cross-origin request and the credentials (such as cookies) are not leaked during the request-response exchange. Furthermore, if the remote server does not specifically allow the user credentials for its website to be included in a cross-origin request from another website and that website does not explicitly declare it wants the user credentials to be passed to the remote server, then the site making the request will most likely get a response that is not personalized. This happens because the user session cookies would not go to the request and the response will not contain data relevant to a particular logged-in user which reduces *CSRF* and other exploits.

To make things simpler, let's say that we have two websites. The first one sets a cookie and whenever the user enters, it shows the cookie's value which is supposed

to be his name. The other website makes a cross-origin Ajax request and adds the response to its DOM.



Getting the Page as the User Sees It with CORS

If we want to include the user credentials with the remote request, we have to do two changes, the first in the code of the website making the request and the second in the website receiving the request. In the website making the request we have to set the withCredentials property of the Ajax request to true:

```
var xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

The remote server itself, besides allowing our origin, has to set an Access-Control-Allow-Credentials header and set its value to true. Using the numbers 1 or 0 would not work.

If we simply set withCredentials to true but the server has not set the above mentioned header, we won't get the response, even if our origin is allowed. We will get a message similar to: *XMLHttpRequest cannot load http://example.com/index.php. Credentials flag is 'true', but the 'Access-Control-Allow-Credentials' header is ''. It must be 'true' to allow credentials. Origin 'http://localhost:63342' is therefore not allowed access.*

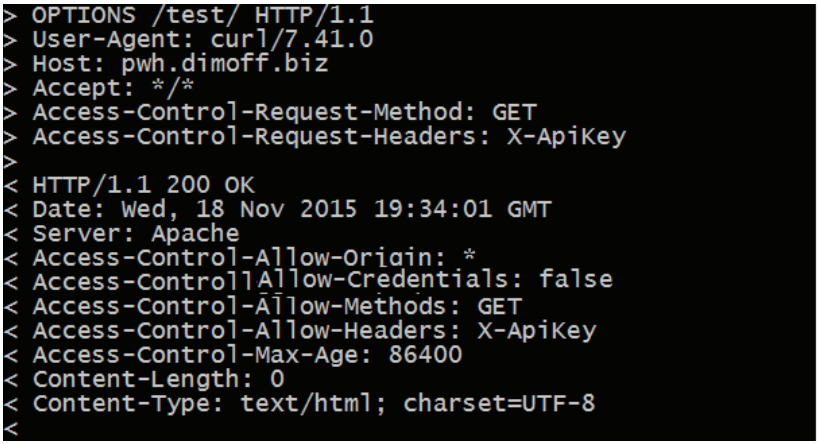
If both changes are made, we will get a personalized response. In our case, the user's name which we stored in a cookie will be in the response that the remote server returns to our website.



However, allowing the credentials to be passed to a cross-origin request is quite dangerous, since it opens up the possibility for various attacks such as CSRF (Cross-Site Request Forgery), XSS (Cross-Site Scripting) and an attacker could take advantage of the user's logged in status to take actions in the remote server without the user knowing (such as withdrawing money if the remote server is banking website).

Preflights

When requests start getting more complicated, we may want to know if a particular request method (such as get, put, post, patchor delete) or a particular custom header is allowed and accepted by the server. In this case, you may want to use preflights where you first send a request with the options method and declare what method and headers your request will have. Then, if the server returns CORS headers and we see that our origin, headers, and request methods are allowed, we can make the actual request (Origin is a header that is passed by our browsers with every cross-origin request we make. And no, we cannot change the Origin's value when making a request in a typical browser).



As we can see in the image above, the server returns several headers which we can use in determining whether to make the actual request. It returns to us that all origins are allowed (Access-Control-Allow-Origin: *, that we cannot make the request while passing the user credentials (Access-Control-Allow-Credentials), that we can only make get requests (Access-Control-Allow-Methods) and that we may use the *X-ApiKey* custom header (Access-Control-Allow-Headers). Lastly, the Access-Control-Max-Age headers shows the value in seconds, pinpointing how long (from the time of the request) we can make requests without relying on another preflight.

On the other hand, in our front-end logic we pass the Access-Control-Request-Method and we pass the Access-Control-Request-Headers to indicate what kind of request method and what kind of headers we intend to add to our real request. In Vanilla JavaScript, you can attach a header when making Ajax calls using the xhr.setRequestHeader('headerString', 'headerValueString');.

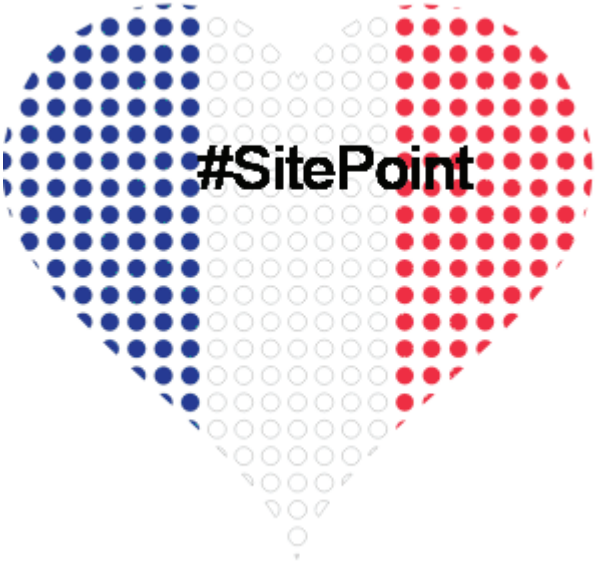
CORS for Canvas Images

If we want to load external images and edit them in canvas or just save their base64 encoded value in localStorage as a cache mechanism, the remote server has to enable CORS. There are various ways this can be done. One way is to edit your web server's configuration to add the Access-Control-Allow-Origin header on every request for specific image types, such an example is shown in the Mozilla docs. If we have a script which dynamically generates images by changing the Content-Typeand outputs an image such as *http://example.com/image/image.php?image=1* we can simply set that header along with outputting the image. Without CORS, if we try to access a remote image, load it in canvas, edit it and save it with toDataURL or just try to add the modified image to the DOM with toDataURL, we will get the following security exception (and we will not be able to save or show it): *Image from origin 'http://example.com' has been blocked from loading by Cross-Origin Resource Sharing policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:63342' is therefore not allowed access*. If the server where the image is returns the image along with a Access-Control-Allow-Origin: * header, then we can do the following:

```
var img = new Image,
   canvas = document.createElement(„canvas“),
   ctx = canvas.getContext(„2d“),
          src  =  „http://example.com/test/image/image.
php?image=1“;
   img.setAttribute(‚crossOrigin‘, ‚anonymous‘);
   img.onload = function() {
      canvas.width = img.width;
      canvas.height = img.height;
      ctx.drawImage( img, 0, 0 );
      ctx.font = „30px Arial“;
      ctx.fillStyle = „#000“;
       ctx.fillText(„#SitePoint“,canvas.width / 3,canvas.
height / 3);
      img.src = canvas.toDataURL();
  document.querySelector(„body“).appendChild(img);
      localStorage.setItem( „savedImageData“, canvas.
toDataURL(„image/png“) );
   }
   img.src = src;
```

This will load an external image, add a *#SitePoint* text in it and both display it to the user and save it in localStorage. Notice that we set a crossOrigin attribute of the external image – img.setAttribute(‹crossOrigin›,

‹anonymous›);. This attribute is mandatory and if we do not add it to the external image we will still get another security exception.



The Crossorigin Attribute

When we make requests for external images, audio, video, stylesheets and scripts using the appropriate HTML(5) tag we are not making a CORS request. This means that no Origin header is sent to the page serving the external resource. Without CORS, we would not be able to edit an external image in canvas, view exceptions and error logging from external scripts that

our website loads, or use the CSS Object Model when working with external stylesheets and so on. There are certain cases when we want to use those features and this is where the crossorigin attribute that we mentioned above comes in handy.

The crossorigin attribute can be set to elements such as <link>,<img> and <script>. When we add the attribute to such an element, we make sure that a CORS request will be made with the Origin header set properly. If the external resource allows your origin through the Access-Control-Allow-Origin header the limitations to non-CORS requests will not apply.

The crossorigin attribute has two possible values:

1. anonymous– setting the crossorigin attribute to this value will make a CORS request without passing the user's credentials to the external resource (similar to making an Ajax CORS request without adding the withCredentials attribute).
2. use-credentials – setting the crossorigin attribute to this value will make a CORS request to the external resource along with any user credentials that might exist for that resource. For this to work, the server must not only set an Access-Control-Allow-Origin header that allows your Origin but it must also set Access-Control-Allow-Credentials to true.

User credentials include cookies, HTTP Basic Auth credentials, certificates and other user data that are sent when the user requests a specific website.

Conclusions

CORS enables developers to further interact with other systems and websites in order to create even better Web experiences. It can be used along with traditional requests made by popular HTML tags as well as with Web 2.0 technologies like Ajax.

Have you been using CORS in your projects? Did you have difficulties with it? We would like to know what your impressions of it are so far.

References and Further Reading:

- Caniuse.com
- HTML5 Rocks – A resource for open web HTML5 developers
- Mozilla Developer Network – CORS enabled image
- Mozilla Developer Network – CORS settings attributes
- Mozilla Developer Network – HTTP access control (CORS)

- Mozilla Developer Network – XDomainRequest
- W3.org – Cross-Origin Resource Sharing
- Mozilla Developer Network – The HTML < link > element



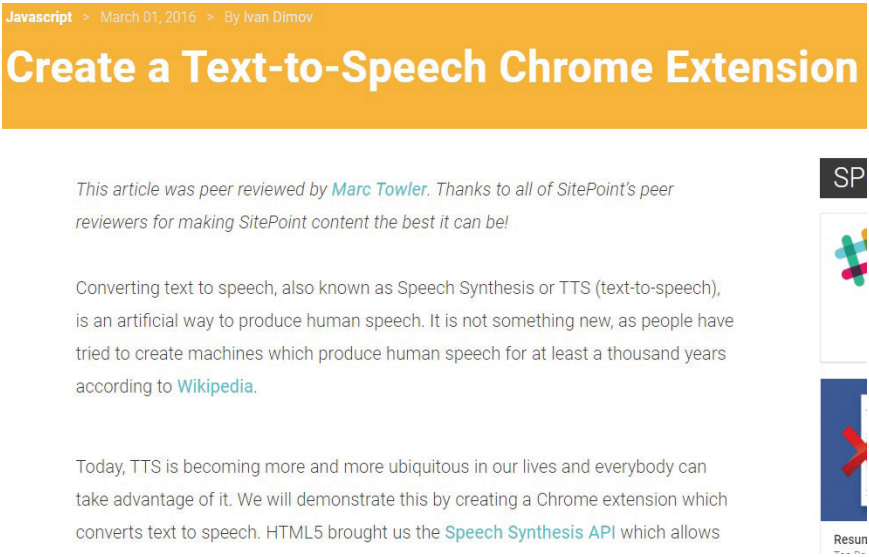
Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article source: https://www.sitepoint.com/an-in-depth-look-at-cors/

Create a Text-to-Speech Chrome Extension



Javascript > March 01, 2016 > By Ivan Dimov

Create a Text-to-Speech Chrome Extension

This article was peer reviewed by Marc Towler. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

Converting text to speech, also known as Speech Synthesis or TTS (text-to-speech), is an artificial way to produce human speech. It is not something new, as people have tried to create machines which produce human speech for at least a thousand years according to Wikipedia.

Today, TTS is becoming more and more ubiquitous in our lives and everybody can take advantage of it. We will demonstrate this by creating a Chrome extension which converts text to speech. HTML5 brought us the Speech Synthesis API which allows

This article was peer reviewed by Marc Towler. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

Converting text to speech, also known as Speech Synthesis or TTS (text-to-speech), is an artificial way to produce human speech. It is not something new, as people have tried to create machines which produce human speech for at least a thousand years according to Wikipedia.

Today, TTS is becoming more and more ubiquitous in our lives and everybody can take advantage of it. We will demonstrate this by creating a Chrome extension which converts text to speech. HTML5 brought us the Speech Synthesis API which allows any Web application to convert some arbitrary text string to speech and play it to its users without a cost.

An extension for Chrome typically consists of the following things:

1. A manifest (a required file with metadata)
2. Images (such as the extension's icon)
3. HTML files (such as a popup that will be shown when the users click on the extension's icon)
4. JavaScript files (such as content and/or background scripts which will be explained in a bit)
5. Any other asset that may be used by your application (such as a stylesheet)

About the Page to Speech Extension

Because of Chrome's popularity and the rise of TTS, we will be creating a Chrome extension which converts text to speech. The extension is going to wait until the user clicks on its icon or presses a special hotkey (shift + Y) and then it will try to either find what the user has highlighted on the page that they are currently viewing or it will try to find what is copied to their clipboard. If anything is there, it will convert it to speech by first trying to use the HTML5 Speech Synthesis API and if that is not available – by calling a third-party API.

Chrome Extension Basics

Each Chrome extension needs to have a file called manifest.json. The manifest is a JSON-formatted file which contains data crucial to the application ranging from things such as the name, description, icons, and the author of the extension to data defining the needs of the extension – which websites should the extension be able to run on (those would be the permissions which the user must grant) or what files to run when the user is browsing a particular website.

```
{
 „manifest_version“: 2,

 “name”: “Page to Speech”,
  “description”: “This extension will produce English speech to whatever text you highlight on a webpage. Highlight text and click the extension’s icon”,
 “author”: “Ivan Dimov”,
 “version”: “1.0”,
 “icons”: {
  “16”: “icon16.png”,
  “48”: “icon48.png”,
  “128”: “icon128.png”
 },
```

Our manifest starts by documenting the name, the description, the author, the version and the icons of the extension. You can provide numerous icons that respond to different sizes in the icons object.

```
„background“: {
  „scripts“: [„background.min.js“]
},
„content_scripts“: [
  {
    „matches“: [„http://*/*“, „https://*/*“],
    „js“: [ „polyfill.min.js“, „ext.min.js“],
    „run_at“: „document_end“
  }],
```

Then, we have a background script called background.min.js (notice that we use minified files) defined in th e background object. Background scripts are scripts which are long-running and will continue to run until the user's browser is closed or the extension is disabled.

Afterwards, we have a content_scripts array which instructs Chrome to load two JavaScript files on every website request due to the wildcards «http://*/*» and «https://*/*». Content scripts, unlike background scripts, have access to the DOM of the actual website which the user is visiting. Content scripts can both read and make modifications to the DOM of any webpage that they are embedded in. Therefore, our polyfill.min.js and ext.min.js would be able to read and modify all data on every webpage.

```
„browser_action“: {
  „default_icon“: „speech.png“
},
 „permissions“: [
   „activeTab“,
   „clipboardRead“
  ]
}
```

Not so fast! We have another array called permissions with which we request to have access only to the webpage that is currently opened by the user (the active tab). We also ask for another permission called clipboardRead which would allow us to read the clipboard of the user (so we can convert its contents to speech).

Coding the Page to Speech Chrome Extension

Firstly, we create our one and only background script which hooks up an event listener that will be fired when a user clicks on the extension's icon. When this happens we call the sendMessage function which sends a message to our content script (the content script can read the DOM and find out what the user has highlighted or/and what the user has placed on their clipboard) with the help of the chrome.tabs.sendMessage(tabId, mes

sage, callback) method. We send the message to the currently opened tab – as this is what interests us and it is what we have access to – with the help of the chrome.tabs.query method whose arguments involve a callback that will be called with an argument that contains the tabs that match the query.

```
chrome.browserAction.onClicked.addListener(function (tab) {
  //fired when the user clicks on the ext‘s icon
  sendMessage();
});
function sendMessage() {
   chrome.tabs.query({active: true, currentWindow: true}, function(tabs){
        chrome.tabs.sendMessage(tabs[0].id, {action: „pageToSpeech“}, function(response) {});
  });
}
```

Now, the lengthier part is our content script. We create an object which will hold some of the data involved with the extension and then define our initialization method.

```
initialize: function() {
    if (!pageToSpeech.hasText()) { return;}
    if (!pageToSpeech.trySpeechSynthesizer()) {
        pageToSpeech.trySpeechApi();
    }
},
```

The method checks if the user has not highlighted text or if they do not have anything on their clipboard and just returns in such a case. Otherwise, it tries to produce speech with the HTML5 Speech Synthesis API. If that fails too, it finally tries to use a third-party API.

The method that checks for text does several things. It tries to get an object with the highlighted text with the help of the built-in getSelection() method and convert it to a text string with toString(). Then, if no text is highlighted it attempts to find the text on the user's clipboard. It does this by adding an input element to the page, focusing it, firing up a paste event with the help of execCommand('paste') and then saving the pasted text inside that input in a property. Then it empties the input. In either case, it returns whatever it has found.

```
 hasText: function() {
    this.data.highlightedText = window.getSelection().
toString();
   if (!this.data.highlightedText) {
     var input = document.createElement(„input“);
     input.setAttribute(„type“, „text“);
     input.id = „sandbox“;
         document.getElementsByTagName(„body“)[0].
appendChild(input);
  varsandbox=document.getElementById(„sandbox“);
     sandbox.value = „“;
     sandbox.style.opacity = 0;
     sandbox.focus();
     if (document.execCommand(‚paste‘)) {
       this.data.highlightedText = sandbox.value;
     }
     sandbox.value = „“;
   }
   return this.data.highlightedText;

 },
```

To enable the user to run the text to speech conversion with a hotkey (hardcoded to shift + Y) we initialize an array and set an event listener for the onkey-

down and onkeyup events. In the listeners, we store an index corresponding to the keyCode of the pressed key whose value results from the comparison of the type of event e.type to keydown and is a boolean. Therefore, whenever a key is down, the corresponding key index's value will be set to true and whenever a key is released – the index's value will be changed to false. Thus, if both indices 16 and 84 are holding a truthy value – we know that the user is using our hotkeys so we initialize the text to speech conversion.

```
addHotkeys: function() {
  var activeKeys = [];
  onkeydown = onkeyup = function(evt) {
    var e = evt || event;
    activeKeys[e.keyCode] = e.type == ‚keydown‘;
    if (activeKeys[16] && activeKeys[84]) {
      pageToSpeech.initialize();
    }
  };
}
```

To convert the text to speech we rely on the trySpeechSynthesizer() method. If HTML5 Speech Synthesis exists in the user's browser (window.speechSynthesis) we know that the user is able to use

it and so we check if a speech is currently running (we know if it is running through the pageToSpeech.data.speechInProgress boolean). We stop the current speech if it is in progress (as the trySpeechSynthesizer will start a new speech and we do not want two simultaneous sounds). Then, we set speechInProgress to true and whenever the speech finishes set the property to a falsy value again.

Now, I do not want to go into details as to why we are using speechUtteranceChunker but it is a bug fix related to Chrome stopping the speech synthesis while still in progress after 200-300 hundred words are uttered. Basically, it splits our text string into many smaller chunks (of 120 words in our case) and calls the Speech Synthesis API with one chunk after another.

```
trySpeechSynthesizer: function() {
  if (window.speechSynthesis ) {
     //new speech is about to get started
     if (this.data.speechInProgress) {
       polyfills.speechUtteranceChunker.cancel = true;
     }
   this.data.speechInProgress = true;
       var msg = new SpeechSynthesisUtterance(this.
data.highlightedText);
     //speechSynthesis.speak(msg);
```

```
  //Chrome Implementation BUG: http://stackoverflow.com/questions/21947730/chrome-speech-synthesis-with-longer-texts
      polyfills.speechUtteranceChunker(msg, {
        chunkLength: 120
      },function() {
        //speech has finished
        pageToSpeech.data.speechInProgress = false;
      });
```

Finally, if the HTML5 Speech Synthesis API is not available, we try an API. We have the same property which is used to know if it is necessary to stop an already running audio. Then, we directly create a new Audio object and pass it the URL to the desired API endpoint since the API we have chosen for demonstration directly streams the audio. We just pass it our API key and the text to convert. We also check if the audio fires an error. In such a case, we just display an alert to the user saying that at this time we cannot help (this particular API, Voice RSS, allows 300 requests on the free tier with which we tested the code).

```
trySpeechApi: function() {
  if (this.data.speechInProgress) {
    this.data.fallbackAudio.pause();
  }
  this.data.speechInProgress = true;
    this.data.fallbackAudio = new Audio(„http://api.voicerss.org/?key=your_api_key&src=“ + this.data.highlightedText);
    this.data.fallbackAudio.addEventListener(„error“, function(evt) {
    alert(„Sorry, we cannot produce speech right now. Try upgrading your Chrome browser!“);
  })
  this.data.fallbackAudio.play();
  this.data.fallbackAudio.onended = function() {
    pageToSpeech.data.speechInProgress = false;
  }

},
```

In the end, outside of any local scope we call the addHotkeys method which will start waiting for the user to press the right hotkey and we set up a listener which will wait until a message is received from the background script. If the right message is received (*speakHighlight*) or the hotkey is pressed, we will initialize our text to speech conversion object.

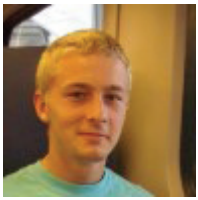
```
chrome.extension.onMessage.
addListener(function(msg, sender, sendResponse) {
  if (msg.action == ‚pageToSpeech‘) {
    pageToSpeech.initialize();
  }
});
pageToSpeech.addHotkeys();
```

Conclusion

And voilà, we have a nice Chrome extension that converts text to speech. The concepts here can be used to create Chrome extensions for different purposes. Have you built any interesting Chrome extensions or do you want to built one? Let me know in the comments!

If you liked the idea and want to develop it further, you can find the complete code in our GitHub repository. The production version of the extension can be found in the Chrome Web Store if you want to test it out.

References:
https://en.wikipedia.org/wiki/Speech_synthesis#History
http://stackoverflow.com/questions/21947730/chrome-speech-synthesis-with-longer-texts



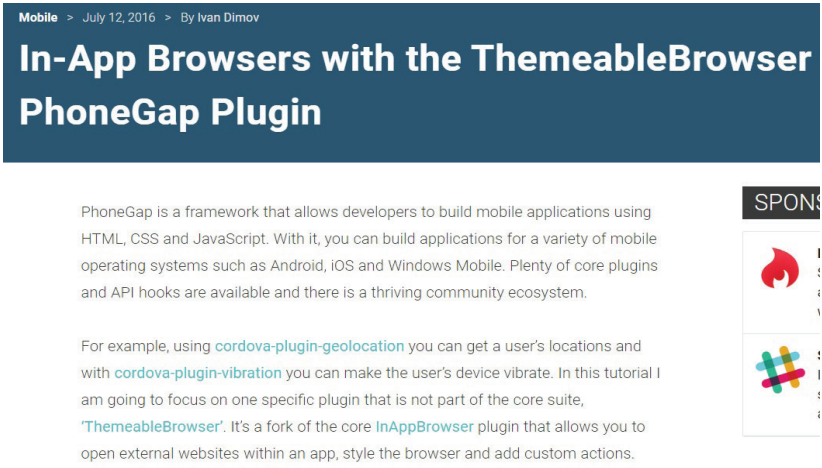
Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article source: https://www.sitepoint.com/create-text-to-speech-chrome-extension/

In-App Browsers with the ThemeableBrowser PhoneGap Plugin



Mobile > July 12, 2016 > By Ivan Dimov

In-App Browsers with the ThemeableBrowser PhoneGap Plugin

PhoneGap is a framework that allows developers to build mobile applications using HTML, CSS and JavaScript. With it, you can build applications for a variety of mobile operating systems such as Android, iOS and Windows Mobile. Plenty of core plugins and API hooks are available and there is a thriving community ecosystem.

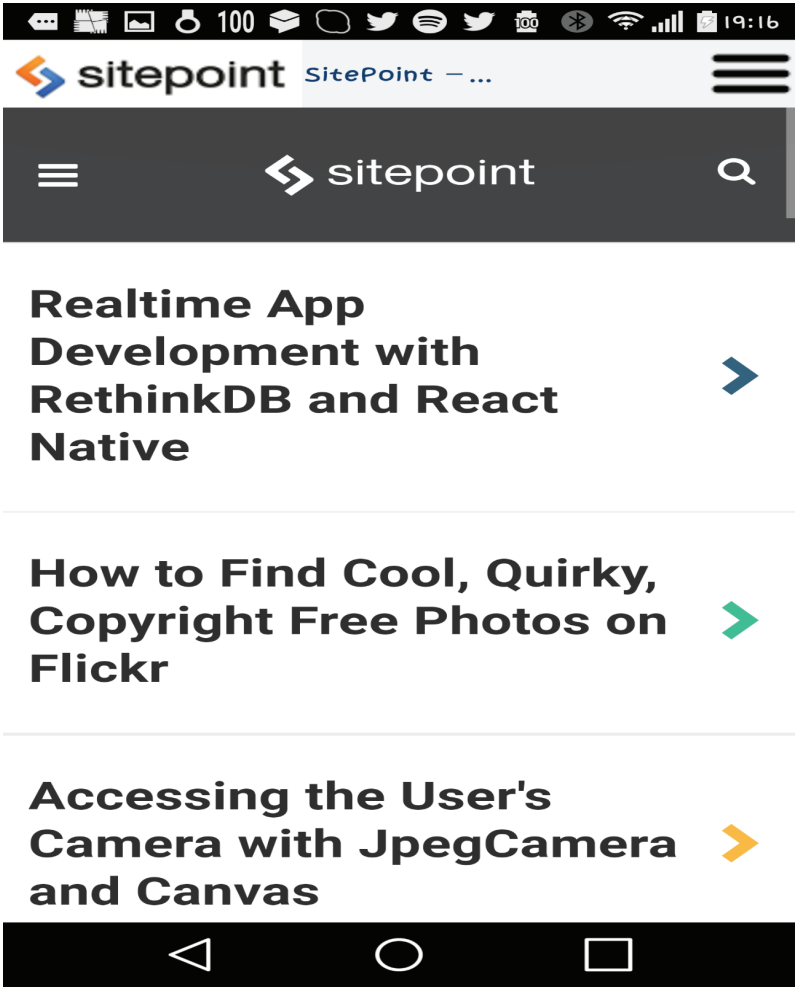
For example, using cordova-plugin-geolocation you can get a user's locations and with cordova-plugin-vibration you can make the user's device vibrate. In this tutorial I am going to focus on one specific plugin that is not part of the core suite, 'ThemeableBrowser'. It's a fork of the core InAppBrowser plugin that allows you to open external websites within an app, style the browser and add custom actions.

SPONS

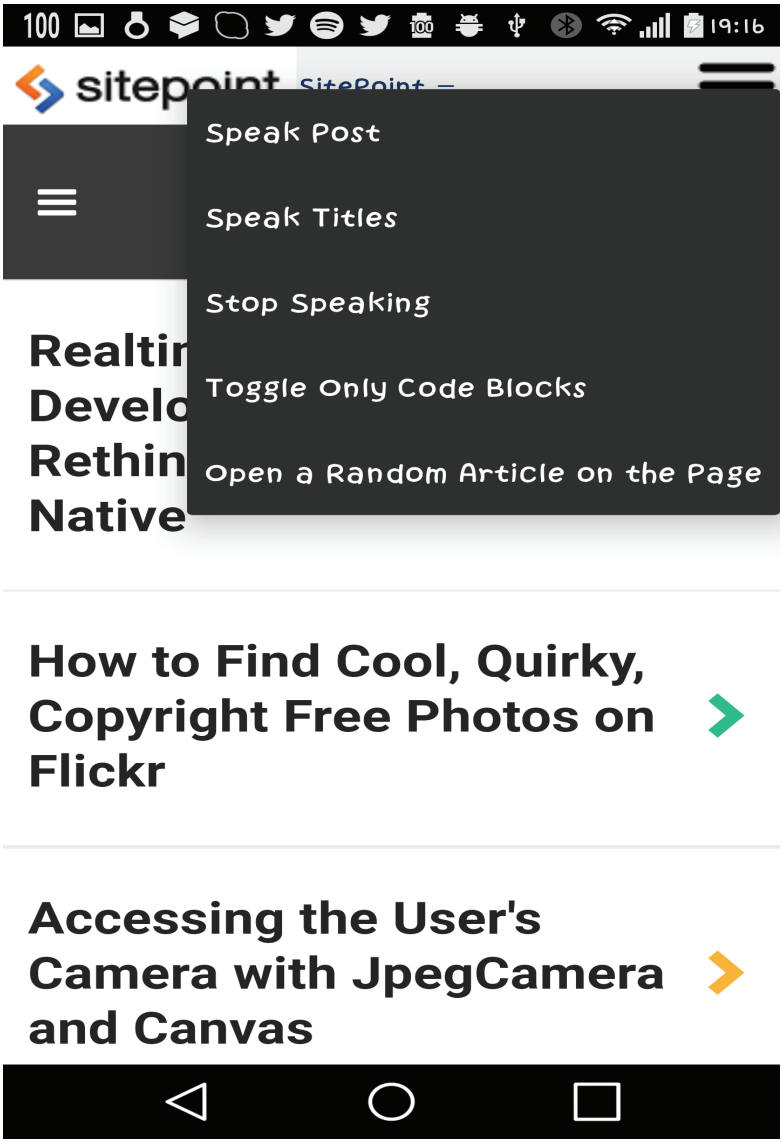
PhoneGap is a framework that allows developers to build mobile applications using HTML, CSS and JavaScript. With it, you can build applications for a variety of mobile operating systems such as Android, iOS and Windows Mobile. Plenty of core plugins and API hooks are available and there is a thriving community ecosystem.

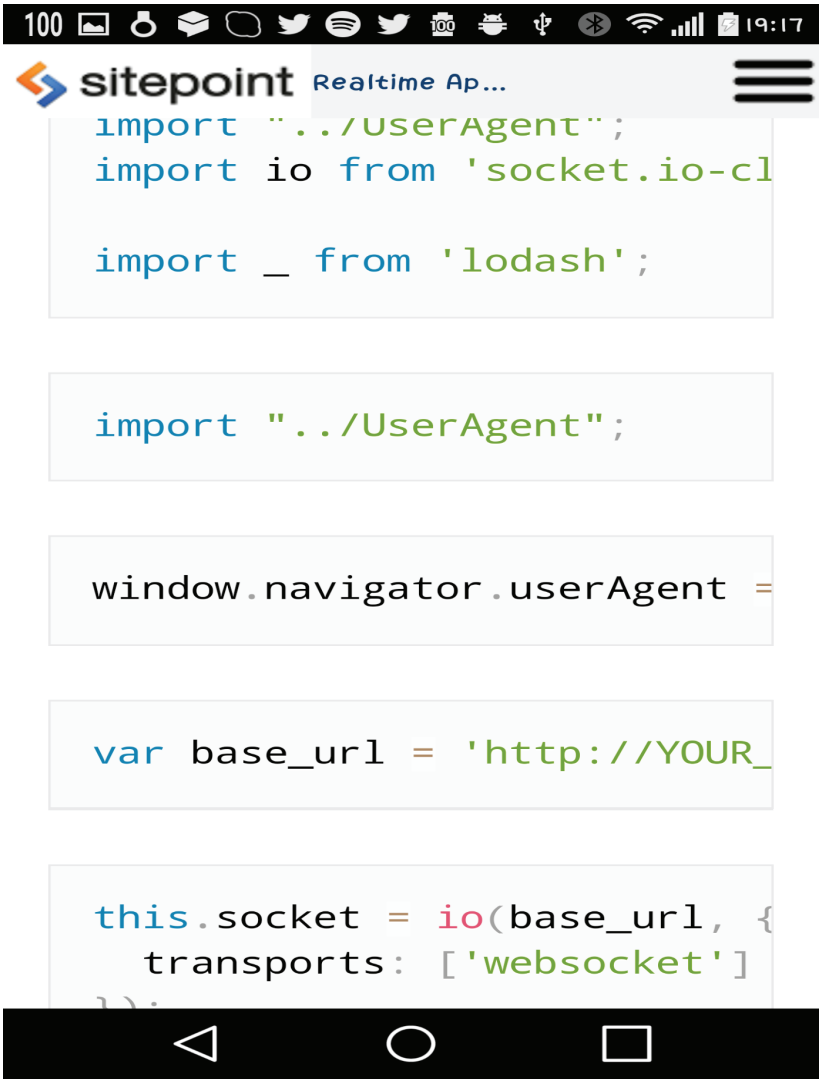
For example, using cordova-plugin-geolocation you can get a user's locations and with cordova-plugin-vibration you can make the user's device vibrate. In this tutorial I am going to focus on one specific plugin that is not part of the core suite, 'ThemeableBrowser'. It's a fork of the core InAppBrowser plugin that allows you to open external websites within an app, style the browser and add custom actions.

By the end of this tutorial, you will have created a simple mobile app which displays SitePoint's website alongside some bonus features.



Notice the hamburger icon to the right, the logo to the left and the title of the current webpage in the middle


19:16
sitepoint
Speak Post
Speak Titles
Stop Speaking
Toggle Only Code Blocks
Open a Random Article on the Page
How to Find Cool, Quirky, Copyright Free Photos on Flickr
Accessing the User's Camera with JpegCamera and Canvas



You can find the code for this tutorial on GitHub.

Creating an 'Hello World' PhoneGap App

You will need Node and npm installed. If you don't already then read SitePoint's quick tip to get you started.

Now run the following in your terminal to install PhoneGap:

```
npm install -g phonegap@latest
```

Now you can run:

```
phonegap create project-name
```

On the command line to create a new project and a *project-name* folder. Inside that folder is a *www* folder that contains your HTML-based application.

To test the application in a browser run:

```
phonegap serve
```

Or run the application on an emulator (in this case iOS) or device with:

```
phonegap run ios
```

You can build a test version of the app using the phonegap build command. If you want to make a version ready for release to the appropriate device store (in this case Android), run phonegap build android --release --buildConfig fileWithKeyInformation.json, passing the command build data (such as a keystore) in a JSON file.

You may need to download other dependencies to build mobile apps for different devices. For example, to build an app for the Google Play Store you will need to install the Android SDK, Java SE Runtime Environment and the Java Development Kit.

Installing and Managing Required Plugins

You can see the plugins an app is using by typing phonegap plugin list inside a PhoneGap project folder. You can delete unneeded plugins using phonegap plugin remove plugin-name. You can add plugins using phonegap plugin add plugin-name. It's good practice to remove the plugins you don't need as many use permissions that may be unnecessary and put off potential users.

To declare that you want to use the ThemeableBrowser plugin, run phonegap plugin add cordova-plugin-themeablebrowser inside the project folder. You can then call its methods within JavaScript code using cordova.

ThemeableBrowser. You also need to add a text-to-speech (tts) API for use within the custom browser. Do this by running phonegap plugin add https://github.com/domaemon/org.apache.cordova.plugin.tts.git and using navigator.tts within JavaScript code. You can see all the available methods of the tts in its included JavaScript file.

Launching the Custom Mobile Browser

As the code for the application contains only HTML, CSS and JavaScript you can use a file and folder structure that suits you. The 'Hello World' PhoneGap app comes with an *index.html* file (you can define the file from which your application starts in *config.xml*) where you can add your logic and load other assets that the application requires such as JavaScript files.

Sections of the code use JQuery, so add this as a dependency to *index.html*:

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js" integrity="sha256-ZosEbRLbNQzLpnKIkEdrPv7lOy9C27hHQ+Xp8a4MxAQ=" crossorigin="anonymous"></script>
```

Open *www/js/index.js* and you will notice:

```
...
```

```
bindEvents: function() {
        document.addEventListener(‚deviceready‘,  this.
onDeviceReady, false);
},
...
```

With PhoneGap, it’s best to trigger application logic after the deviceready event has fired. This event fires after all PhoneGap APIs have loaded and the device is ready to execute further code. If you run any code before it has fired and use an hardware APIs, Geolocation for example, there is a chance that the application will crash.

Call the method that will open the SitePoint website with ThemeableBrowser inside the receivedEvent function after receiving notification that the device is ready:

```
receivedEvent: function(id) {
   app.openPage(„http://sitepoint.com“);
   ...
}
```

The openPage method takes an URI, calls cordova.ThemeableBrowser.open, passing it the URI and an object with options and saves the reference to the opened webpage in a property. You need that property

if you want to navigate to another page whilst the user is browsing this one, or if you want to close the web browser.

Create the function in *www/js/index.js*:

```
...
openPage: function(url) {
  var options = app.getBrowserOptions();
    app.properties.ref = cordova.ThemeableBrowser.
open(url, ,_blank‘, options);
  app.addEventListeners();
},
...
```

And the associated getBrowserOptions function that passes the options to the function:

```
...
getBrowserOptions: function() {
  var options = {
    ...
  }
  return options;
},
...
```

In the options object, add a status bar, and a toolbar property which sets the height of the toolbar to 44 pixels and adds a white-ish background color:

```
var options = {
 statusbar: {
   color: ‚#ffffffff‘
},
 toolbar: {
    height: 44,
    color: ‚#f0f0f0ff‘
 },
}
```

Add another property which indicates that you want to show the title of the opened webpage in the middle of the toolbar, and give it a specific color:

```
var options = {
toolbar: {
   height: 44,
   color: ‚#f0f0f0ff‘
},
title: {
   color: ‚#003264ff‘,
   align: „center“,
   showPageTitle: true
}
}
```

Add a customButtons property which contains an array of custom buttons to be added to the browser. To add a new button, you pass an object with the button's features.

For this particular button pass the object an image or wwwImage property with the URL to the image of the button. wwwImage differs from the image property in that the path to the image when using wwwImage starts from the www folder whereas image expects images in the native project. You can find the images used in the example here.

Align the button to the left side of the toolbar and tell ThemeableBrowser that you want the event called SitePointSitePressed to fire when the button is clicked. You will add the event later.

```
...
options.customButtons = [{
    wwwImage: ‚img/btns/sitepoint-logo.png‘,
    wwwImagePressed: ‚img/btns/sitepoint-logo.png‘,
    wwwImageDensity: 1,
    align: ‚left‘,
    event: ‚SitePointSitePressed‘
}]
...
```

Add a menu to the toolbar by creating a menu property, again passing an image. Align it to the right and add an items property which contains an array of the different menu items and the event that is going to fire when selected.

```
...
options.menu = {
    wwwImage: ‚img/btns/menu.png‘,
    imagePressed: ‚img/btns/menu-pressed.png‘,
    wwwImageDensity: 1,
    title: ‚Effects‘,
    cancel: ‚Cancel‘,
    align: ‚right‘,
    items: [{
        event: ‚speakPostPressed‘,
        label: „Speak Post“
    },
        {
            event: ‚speakTitlesPressed‘,
            label: „Speak Titles“
        }, {
            event: ‚stopSpeakingPressed‘,
            label: „Stop Speaking“
        }, {
            event: ‚viewCodeBlocks‘,
```

```
            label: ‚Toggle Only Code Blocks‘
        },
        {
            event: ‚randomArticlePressed‘,
            label: ‚Open a Random Article on the Page‘
        }
    ]
}
...
```

You now need to add event listeners and functions. To do this, set the event listeners on the reference you have of the result of cordova.ThemeableBrowser.open, i.e. the app.properties.ref set before.

These event listeners that call the run method of the object that will handle the application's logic, passing it different parameters for each event. The run method will to try to call the method that you pass as a parameter.

```
addEventListeners: function() {
 app.properties.ref.addEventListener(‚viewCodeBlock
s‘, function(e) {
        SitePointPostOptions.run(„viewCodeBlocks“);
   }).addEventListener(‚speakPostPressed‘, function(e)
{
        SitePointPostOptions.run(„speakPost“);
        }).addEventListener(cordova.ThemeableBrowser.
```

```
EVT_ERR, function(e) {
        console.error(e.message);
        }).addEventListener(cordova.ThemeableBrowser.
EVT_WRN, function(e) {
        console.log(e.message);
     }).addEventListener("stopSpeakingPressed", func-
tion(e) {
        SitePointPostOptions.run("stopSpeaking");
        }).addEventListener("speakTitlesPressed", func-
tion(e) {
        SitePointPostOptions.run("speakTitles");
    })
         .addEventListener("SitePointSitePressed", func-
tion(e) {
        SitePointPostOptions.run("logoClick");
    })
       .addEventListener("randomArticlePressed", func-
tion(e) {
        SitePointPostOptions.run("randomArticle");
    }).addEventListener("loadstop", function(evt) {
       if (SitePointPostOptions.properties && SitePoint-
PostOptions.properties.length) {
         SitePointPostOptions.properties.areCodeBlocks-
Shown = false;
        }
},
```

The next step is crucial. Add an extra event listener for the loadstop event of the property that contains a reference to the opened webpage. The loadstop event fires whenever the browser finishes loading a webpage. It could be the webpage that just opened or any other webpage that the user navigates to afterwards.

In the listener, return from the function if the user is not located on the SitePoint website. If they are, execute JavaScript code on the webpage that the user is on.

Call document.body.innerHTML to get the HTML contents of the webpage and add a callback which saves the content to a property. Using this property, you can traverse and read the contents of any webpage the user navigates to.

```
addEventListeners: function() {
 app.properties.ref.addEventListener(‚viewCodeBlock
s‘, function(e) {
        SitePointPostOptions.run(„viewCodeBlocks“);
  }).addEventListener(‚speakPostPressed‘, function(e)
{
        SitePointPostOptions.run(„speakPost“);

      }).addEventListener(cordova.ThemeableBrowser.
EVT_ERR, function(e) {
        console.error(e.message);
```

```
    }).addEventListener(cordova.ThemeableBrowser.
EVT_WRN, function(e) {
        console.log(e.message);

    }).addEventListener("stopSpeakingPressed", func-
tion(e) {
        SitePointPostOptions.run("stopSpeaking");

    }).addEventListener("speakTitlesPressed", func-
tion(e) {
        SitePointPostOptions.run("speakTitles");
    })
    .addEventListener("SitePointSitePressed", func-
tion(e) {
        SitePointPostOptions.run("logoClick");
    })
    .addEventListener("randomArticlePressed", func-
tion(e) {
        SitePointPostOptions.run("randomArticle");
    }).addEventListener("loadstop", function(evt) {
        if (SitePointPostOptions.properties && Site-
PointPostOptions.properties.length) {
            SitePointPostOptions.properties.areCode-
BlocksShown = false;
        }
```

```
    if (evt.url.indexOf("sitepoint.com") === -1) {
      return;
    }
    app.properties.ref.executeScript({
        code: "document.body.innerHTML"
      },
      function(values) {
        alert("The app's menu is now ready for use.");
        app.properties.pageContents = values;
      }
    );
  })
},
```

The method that opens a random article shows something interesting, how to close the open webpage and open a new one. This uses the close method on the property with the opened page and sets a listener for the exit event of the browser. Whenever the browser is about to exit, wait a few seconds for it to close and open a new page by calling the openPage method with the new URI. The new URI is a random href attribute of the anchors of all articles within a SitePoint page that contains lists of articles. If the page does not contain any articles, pop up an alert.

Add the logic behind the browser to a new *ww/js/menuLogic/logic.js* file:

```
SitePointPostOptions = {};
SitePointPostOptions.properties = {};
SitePointPostOptions.properties.areCodeBlocksShown = false;

SitePointPostOptions.run = function(type, options) {
   SitePointPostOptions[type].call(this, options);
}

SitePointPostOptions.randomArticle = function() {
      var articles = $(app.properties.pageContents[0]).find(“.article .article_title a”);
   if (!articles.length) {
        alert(“You are probably not on a SitePoint page with a list of articles!”);
   }
   var randomIndex = Math.floor(Math.random() * articles.length);
     var linkToFollow = articles[randomIndex].getAttribute(“href”);
      app.properties.ref.addEventListener(“exit”, function() {
```

```
        setTimeout(function() {
            app.properties.ref = null;
            app.openPage(linkToFollow);
        }, 2000)

    })
    app.properties.ref.close();
}
```

Link to this new file in *index.html*:

```
<script type='text/javascript' src='js/menuLogic/logic.
js'></script>
```

When the user clicks the logo (the custom button added on the left side), open the SitePoint website using the system's default browser. To open a webpage outside of the internal browser, give a second parameter to the cordova.ThemeableBrowser.open method with the value _system.

Still inside *www/js/menuLogic/logic.js*, add:

```
SitePointPostOptions.logoClick = function() {
    cordova.ThemeableBrowser.open(„http://sitepoint.com“, „_system“);
}
```

The next method speaks the titles of webpages by passing the text of all heading elements to the speakText helper.

```
SitePointPostOptions.speakTitles = function() {
  //TODO: Speak only the tiles of the pages
 vartitlesContents=$(app.properties.pageContents[0]).find(„h1,h2,h3,h4,h5,h6“);
  if (!titlesContents.length) {
    alert(„There is probably no title out there to speak aloud!“);
  }
  titlesContents = titlesContents.text();
  SitePointPostOptions.speakText(titlesContents);
}
```

The next method speaks the contents of posts, with an accompanying method that stops the speaking:

```
SitePointPostOptions.speakPost = function() {
   //TODO: speak post
  var postContents = $(app.properties.pageContents[0]).
find(„.ArticleCopy“).find(„p,h1,h2,h3,h4,h5,h6“);
   if (!postContents.length) {
         alert(„There is probably no post open to speak
aloud.“);
   }
   postContents = postContents.text();
   SitePointPostOptions.speakText(postContents);
}

SitePointPostOptions.stopSpeaking = function() {
  navigator.tts.stop(function() { /*success callback*/ },
function() { /*err callback*/ });
   navigator.tts.interrupt(“”, function() { /*success call-
back*/ }, function() { /*err callback*/ });

   navigator.tts.shutdown(function() {
      /*successfully shut down tts*/
   }, function() { /*err*/ })
}
```

The next method toggles code blocks on the given SitePoint post by inserting specific CSS rules to the webpage that the user is viewing:

```
SitePointPostOptions.viewCodeBlocks = function() {
  //TODO: filter only code blocks;
                if (SitePointPostOptions.properties.
areCodeBlocksShown) {
    app.properties.ref.insertCSS({
       code: „.ArticleCopy > *:not(pre) { display:block
!important;}“
    })
                        SitePointPostOptions.properties.
areCodeBlocksShown = false;
    return;
  }
  app.properties.ref.insertCSS({
      code: „.ArticleCopy > *:not(pre) { display:none
!important;}“
  })
                        SitePointPostOptions.properties.
areCodeBlocksShown = true;
}
```

What Else?

You have created a mobile app browser packed full of goodies. You could expand these to include custom branding and functionality that suits your app use case.

Have you ever implemented a custom mobile browser before or do you feel like you have something that you want to create? What is it?

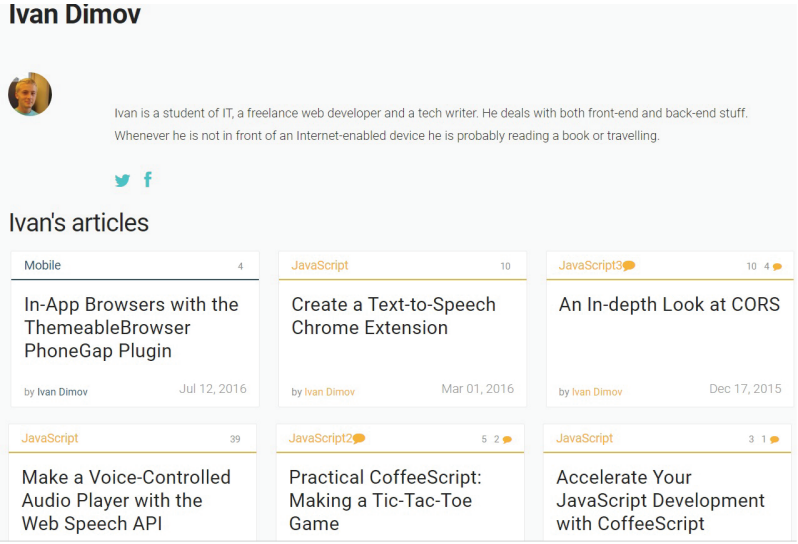


Meet the author

Ivan Dimov

Ivan is a student of IT, a freelance web developer and a tech writer. He deals with both front-end and back-end stuff. Whenever he is not in front of an Internet-enabled device he is probably reading a book or travelling.

Article source: https://www.sitepoint.com/an-in-app-browser-with-themeablebrowser-phonegap-plugin/



1.Accelerate Your JavaScript Development with CoffeeScript …….. 5

2.Practical CoffeeScript: Making a Tic-Tac-Toe Game ……..19

3. Make a Voice-Controlled Audio Player with the Web Speech API ……. 41

4.An In-depth Look at CORS ……59

5.Create a Text-to-Speech Chrome Extension …….. .77

6. In-App Browsers with the ThemeableBrowser PhoneGap Plugin ……..……93