HTML5 Developer's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Tuesday, August 5, 2014

Table of Contents

About This Guide

This guide explains how to develop optimal user interfaces for applications created with the QNX SDK for Apps and Media.

This table may help you find what you need in this guide:

To find out about:	Go to:
The QNX SDK for HTML5 and where to download it	*SDK Overview* (p. 9)
The WebKit-based browser engine	*Browser Engine* (p. 11)
HTML5 elements (audio, video, etc.) you can use in your apps	*HTML5 elements* (p. 15)
Using "sandboxing" so that applications can run in complete isolation from each other	*Web Sandbox Model* (p. 19)
The process for creating an HTML5 app	*Developing HTML5 Apps* (p. 21)
The HTML5 development environment and tools	*The HTML5 development environment* (p. 22)
Creating a basic application	*Creating an HTML5 app* (p. 24)
Creating plugins to extend your app's functionality	*Creating Your Own Cordova Plugin* (p. 25)
Best practices for optimal performance	*Enhancing Performance* (p. 33)
Weblauncher APIs (`webview.create`, `webview.status`, etc.)	*WebLauncher's JavaScript APIs* (p. 35)
Using Web Inspector to debug your apps	*Debugging Web Apps* (p. 50)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

To obtain technical support for this product, visit the *BlackBerry Support Forum*.

Chapter 1
SDK Overview

The QNX SDK for HTML5 provides a framework for developing and executing web-compatible applications, specifically using such technologies as HTML5, CSS3, JNext. Using Apache Cordova, you can create and package HTML5 applications for a target device.

To install the latest HTML5 development environment, download the `html5sdk-`*`datestamp`*`.zip` archive from the *QNX Download Center* and follow the instructions given in the accompanying installation note and in the README.

Apache Cordova

Apache Cordova is a framework for application development that allows you to use common web technologies, primarily HTML5, JavaScript, and CSS, to create applications for mobile devices. Using Cordova, you can:

- Package an HTML5 app to deploy it to the target hardware.
- Create JavaScript APIs to access native services on the platform.
- Use *webserver emulation* so that your app can run without actually running a webserver on the target hardware.
- Provide security features to your application.

Since the user interface for your application doesn't have to be created using HTML5, you can use other HMI technologies, such as Qt (a lighter-weight comprehensive framework) and OpenGL (for 2D and 3D graphics on embedded systems).

A non-HTML5 environment may be appropriate if you prefer a different HMI tool, if you have legacy assets in other tooling frameworks, or if a system is being built without the need for extensibility.

Chapter 2
Browser Engine

The browser engine has rich support for features such as canvas, WebSocket, session storage, offline apps, worker threads, Document Object Model (DOM) improvements, audio and video tags, and WebGL.

Included with your SDK is a reference browser application implemented with HTML5, CSS3, and JavaScript. The browser includes tool bars, status bars, URL address entry, buttons, and so on. A browser without these components is called a "chromeless browser". If required, you can customize or replace this browser application. Because the browser application is implemented in HTML5, the code is readily viewable via Web Inspector in WebKit browsers, such as Google Chrome.

Based on *WebKit*, the browser engine provides support for HTML5 and related standards and technologies (including CSS3) and for JavaScript and associated standards, such as AJAX, JavaScript Object Notation (JSON), and XML. The QNX browser engine has optimized WebKit in a number of ways:

- improved user interaction (complex touch event handling, smooth zooming/scrolling, fat-finger touch target detection, etc.), performance, and battery life for mobile devices
- enhanced user operations such as fast scrolling and zoom (e.g., zooming in on a webpage) to reduce RAM utilization
- enhanced JavaScript execution to improve performance and reduce CPU utilization and unnecessary battery drain
- reduced power consumption (e.g., by throttling background threads)
- added support for multimodal input (e.g., trackpad, keyboard, and virtual keyboard)
- improved overall speed (e.g., by selective image down-sampling)

The browser engine provides a set of core classes that you can use to display web content in a window. By default, the browser engine implements the most basic functionality of a browser, such as the ability to follow links and to download and display content. You can use the engine's functionality at the most basic level to display web content in your app or you can use APIs to create your own full-featured, customized, web-based app.

Downloading and browsing content from the web can be a fairly daunting task for a browser, given the wide variety of content and encoding types used on the Internet. The browser engine handles these different content types transparently by creating and managing the objects necessary to render the incoming content. The engine provides view classes used to display content. Each view class (called a *WebView*) contains frames (called *WebFrames*); each frame implements its own scroll bar. You

don't need to implement custom views or custom frames in order to display content in your app.

By running multiple WebViews in a single engine instance, overall memory footprint can be reduced. However, since all apps share the same engine, they're not isolated from each other. Bad behavior in one app can impact all other apps that share the same engine instance. This mode would typically be used for a set of apps that are tested together and deployed as a bundle (e.g., core apps shipped from the manufacturer). Or, a single app can be run in its own private engine instance. This provides isolation at the expense of increased memory footprint.

Web browser app

The web browser app supports URL entry, tabs, back, forward, history, settings, and bookmarks. By default, the browser supports 800×480 and 720p resolutions in landscape mode. To support other resolutions, you can modify the browser application source.

HTML5 apps

The HTML5 application framework provides the necessary additions to the browser engine to allow it to support full-fledged apps. This environment allows developers to create and deploy apps built from web technologies (HTML5, CSS3, and JavaScript) with plugins that provide access to the underlying device hardware and native services, just like native C/C++ apps.

Plugins

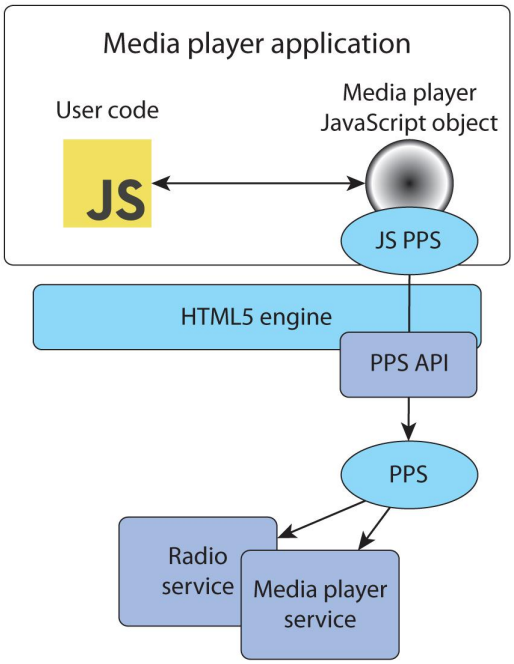
The browser engine includes plugins based on the Netscape Plugin API (NPAPI) through a dynamically linked library. The plugins provide access to PPS (Persistent Publish/Subscribe), SQL, and Screen services. You can add additional plugins as required.

The PPS plugin provides the HTML5 domain with access to the full PPS API.

The SQLite 3 plugin provides SQLite database access, including a complete API for opening, querying, and modifying the database.

JavaScript plugins

JavaScript Cordova plugins use the browser engine plugins to provide HTML5 apps with access to middleware-layer services, including radio, phone, media engine, ASR, and navigation. For example, the audioplayer plugin uses the PPS plugin to provide access to functions such as start, pause, and play. The reference media player application depends on this plugin.



Web Inspector tool

Included as part of WebKit, *Web Inspector* is a useful debugging and profiling development tool for web content. You can use this tool to troubleshoot and optimize your web content for your apps. The tool includes a variety of features and capabilities, such as inspection, profiling, console integration, and more. For details, see "*Debugging Web Apps* (p. 47)" in this guide.

CSS3 support

The browser engine supports CSS3 properties. For a complete list of supported CSS3 properties for WebKit-based browsers, see the *CSS3 Browser Support Reference* at the following W3Schools site:

http://www.w3schools.com/cssref/css3_browsersupport.asp

HTML5 elements

The SDK lets you use HTML5 elements in your apps. For details about these elements, see the following W3Schools references:

Element	Description
HTML5 Audio	Represents a sound or audio stream.
HTML5 Canvas	Provides a container for JavaScript to draw graphics on a webpage.
HTML5 Geolocation	Scripts use this object to programmatically determine the location information associated with the hosting device.
HTML5 Web Storage (`localStorage`)	Provides functions to access a list of key/value pairs for *local storage objects* (i.e., objects that persist after a browser session has ended).
HTML5 Web Storage (`sessionStorage`)	Lets you save a large amount of key/value pairs and text for *session storage objects* (i.e., objects that are valid only for the current browser session).
HTML5 Video	Represents a video or video stream.
HTML5 Web Workers	Allows JavaScript code to be executed in a background thread.

HTML5 offline web applications

HTML5 includes several features that address the challenge of building web apps that work offline. These features include SQL, offline app-caching APIs, `online`/`offline` events, status, and the `localStorage` API.

For more information about creating web apps that work offline, see the W3C document *Offline Web Applications*.

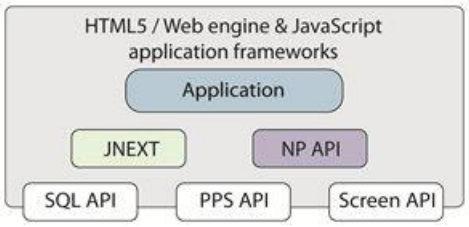
Browser API support

The browser engine supports various standard APIs. For information about these APIs, see the following W3C resources:

API Support	Description
Web SQL Database	A set of APIs to manipulate client-side databases using SQL.
WebSocket API	An API that allows webpages to use the WebSocket protocol to enable web apps to maintain bidirectional communications with a remote host.
Web Workers	An API that allows authors of web apps to spawn background workers running scripts in parallel to their main page. This process allows for thread-like operation with *message passing* as the coordination mechanism.
Geolocation API Specification	An API that provides scripted access to geographical location information associated with the hosting device.

Chapter 3
Web Sandbox Model

The BlackBerry 10 OS architecture is designed to be both simple and powerful. A powerful benefit is the use of sandboxing, so that applications can run in complete isolation from each other. The platform provides a multiprocess architecture that allows system developers to partition the UI into a set of core and sandboxed apps. With this architecture, multiple WebViews (or windows) can either share a common engine instance or run in their own engine instance. Each WebView (equivalent to a tab in a desktop browser) can be implemented with a separate (and different) JavaScript application framework (for example, jQuery Mobile or Sencha Touch).



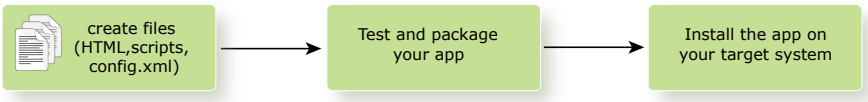
Applications running within the same instance of the HTML5 engine wouldn't be isolated from each other. Incorrect behavior in one application could impact all other applications. For example, a hung or stalled JavaScript thread in one application would hang all other HTML5 applications. This is why the BlackBerry 10 OS architecture runs each application in its own private HTML5 engine instance (its “sandbox”). This design isolates applications so that any problems they encounter don't impact other applications. The sandbox model does, however, increase the system's memory footprint.

Chapter 4
Developing HTML5 Apps

The browser engine is based on the WebKit (*www.webkit.org*) open-source web browser engine, which we have optimized for embedded environments and have extended with numerous capabilities to provide a full-featured web browser.

An HTML5 app created with the *Apache Cordova* framework can be targeted to run on a variety of devices or ported to a different OS, such as iOS or Android. You can take advantage of popular mobile web frameworks, such as Sencha Touch and jQuery Mobile, to provide a wide range of useful APIs that can greatly simplify cross-browser web development.

The process for creating an HTML5 app involves three main steps:



The HTML5 app environment lets you create and deploy applications built from web technologies (HTML5, CSS3, and JavaScript) with plugins that can access the underlying device hardware and native services, just like native C/C++ applications. The HTML5 standard ensures compatibility between browsers, making the BlackBerry 10 OS environment compatible with mobile, desktop, and web environments—the same HTML code can be used in all these environments.

The HTML5 development environment

Here are the key components for developing HTML5 apps for the QNX SDK for Apps and Media:

HTML5 SDK

The HTML5 SDK contains the *Apache Cordova* framework, which you can use to create plugins for your mobile apps. When you installed the QNX SDK for Apps and Media 1.0, you should also have installed the HTML5 SDK (available from the *QNX Download Center*).

JavaScript

HTML5 applications can interact with underlying services via JavaScript plugins. JavaScript classes provide interfaces to various feature components. These consist of the PPS interfaces and the UI core APIs for home screen development. The NPAPI provides extensions to the HTML5 engine through a dynamically linked library. PPS, SQL, and the UI core APIs are implemented through an NPAPI interface. These APIs provide access to native DLLs that give applications access to services such as the composition manager and the launcher. If such services are required, you can add additional plugins to extend your applications.

Persistent Publish/Subscribe (PPS) API

The PPS service provides a simple, filesystem-based API for client applications. For details, see the *Persistent Publish/Subscribe Developer's Guide*.

Node.js

Node.js is a platform built on Chrome's JavaScript runtime for building fast, scalable network applications. Node.js is required—you should have installed it when you installed the HTML5 SDK.

jQuery

jQuery is a popular open-source JavaScript framework for developing web apps (available from *www.jquery.com*). jQuery includes a number of UI widgets and utilities for DOM manipulation. Its features include:

- Open-source framework
- jQuery UI widgets
- Event handling
- CSS3 animations and effects
- *Asynchronous JavaScript And XML* (AJAX) utilities

For more information about jQuery, see *Using jQuery Core* at the jQuery site.

Web Graphics Library (WebGL)

This graphics library is integrated into the *browser engine* (p. 11), so you don't need to download and install any binaries to use it. Based on OpenGL ES 2.0, WebGL is a cross-platform JavaScript API. As a DOM API, it runs in the HTML5 `canvas` element to render interactive 3D graphics in compatible browsers.

For debugging your apps, you may want to install Google Chrome on your development system. This WebKit-based browser works in conjunction with the Web Inspector, a useful debugging and profiling development tool for web content. For details, see "*Debugging Web Apps* (p. 47)" in this guide.

Creating an HTML5 app

An HTML5 application created with Apache Cordova is a standalone app, which means you're not required to point to a remote server to load a webpage or another app because your app lives on the device. The application is essentially an arrangement of web assets that are packaged into a container that can be viewed by a browser (the browser engine container).

To create a basic Cordova app template:

1. Navigate to the folder where you installed the HTML5 SDK.
2. Change to the `cordova/blackberry10/bin` folder.
3. Run the Cordova `create` command to create a new Cordova project:

   `create` *`project_path package_name app_ID`*

   where:

   - *project_path* points to an empty directory
   - *package_name* is a reverse domain name, such as `com.company.appname`
   - *app_ID* is unique for the target system

   In addition, the *package_name* and *app_ID* can't be more than 50 alphanumeric characters long.

   For example: `create hello com.example.hello HelloWorld`

The `create` command generates the folder structure for your project at the specified location (for the example above, the `hello` directory). If you build this app without modifications, you'll produce a small Cordova app that handles the `deviceready` event and displays the Cordova logo on the screen.

You can add your own code and additional resources to create a more robust application. All of your project resource files should be stored in the `/www` folder (or in a subfolder within `/www`).

For information on packaging, installing, and launching your HTML5 app, see the *Application and Window Management* guide.

For instructions on creating a simple Cordova plugin that uses PPS, see "*Creating Your Own Cordova Plugin* (p. 25)" in this guide.

Chapter 5
Creating Your Own Cordova Plugin

Although HTML5 offers a wide range of functionality, that functionality is limited to what's provided by the SDK. An application is implemented as a webpage (named `index.html` by default) that references whatever CSS, JavaScript, images, media files, or other resources are necessary for it to run. The app executes as a WebView within the native application framework. For the web app to interact with various device features the way native apps do, it must also reference a `cordova.js` file, which provides API bindings. If you want to access other features not provided by the platform SDK, you have to write a *plugin*. A plugin is a bridge between the WebView the app is running in and the native layer of the platform. Plugins provide a mechanism to call into native APIs that aren't provided with the SDK.

This chapter assumes that you know how to create Cordova plugins for other platforms. For information on creating Cordova plugins, see the *Apache Cordova Documentation*.

Many of the services available with BlackBerry 10 OS can be accessed through a PPS interface. In the sections that follow, we'll walk through the creation of a simple plugin (`com.qnx.demo`) that creates a PPS object and writes to it. You can use the same principles to manipulate other PPS objects.

Structure of a plugin

You can place the artifacts of the plugin in any directory structure, as long as you specify the file locations in the `plugin.xml` file. Here's a typical structure (names in bold represent directories):

- **`plugin_name`**
  - `plugin.xml`
  - **`www`**
    - `client.js`
  - **`src`**
    - **`blackberry10`**
      - `index.js`
      - `plugin_name.js`
      - other JavaScript or native files, as required (`*.js`, `*.cpp`, `*.hpp`)

The JavaScript part of a plugin must contain, at a minimum, the following resources:

`plugin.xml`

The `plugin.xml` file is an XML document in the `plugins` namespace, `http://apache.org/cordova/ns/plugins/1.0`. The `plugin.xml` file contains a top-level `plugin` element that defines the plugin itself and child elements that define the file structure of the plugin.

You *must* name this file `plugin.xml`.

`client.js`

Considered the client side, this file exports APIs that a Cordova application can call. The APIs in `client.js` make calls to `index.js`. The APIs in `client.js` also connect callback functions to the events that fire the callbacks.

You *must* name this file `client.js`.

`index.js`

Cordova loads `index.js` and makes it accessible through the *cordova.exec()* bridge. The `client.js` file makes calls to the APIs in the `index.js` file, which in turn makes calls to JNEXT to communicate with the native side.

If your plugin needs to include events, make sure to define these events in `index.js`. Events are defined inside the *_actionMap* variable.

You *must* name this file `index.js`.

In our example, we need a file to deal with the PPS activities of the plugin. We'll call this file `demo.js` to reflect the name of the plugin. You can name this file whatever you want, but using the plugin name is standard practice.

Depending on what your plugin needs to do, you might need to create other JavaScript files. In addition, if the native functionality you need isn't available through an existing interface (such as PPS or another plugin), you'll need to write the C/C++ code to provide JavaScript access to that functionality.

The *cordova.exec()* function

You can structure your plugin's JavaScript according to your preference. However, you must use the *cordova.exec()* function to communicate between the Cordova JavaScript and the native environment. *The cordova.exec()* function is defined in the `cordova.js` file and has the following signature:

```
exec (<successFunction>, <failFunction> , <service> , <action> , [< args >]);
```

The parameters are:

<successFunction>

Success function callback. Assuming your *exec()* call completes successfully, this function is invoked (optionally with any parameters you pass back to it).

<failFunction>

Error function callback. If the operation doesn't complete successfully, this function is invoked (optionally with an error parameter).

<service>

The name of the service to call into on the native side. This is mapped to a native class.

<action>

The name of the action to call into. This is picked up by the native class receiving the *cordova.exec()* call, and essentially maps to a class's method.

< args >

Arguments to pass into the native environment.

The *cordova.exec()* function is the key to your plugin—it provides the link between the JavaScript and the native APIs. For more information about *cordova.exec()*, see the Apache Cordova documentation.

Example: Using the PPS interface

In the example, called `com.qnx.demo`, we build a plugin that provides methods and events for creating, updating, and watching a simple PPS object. The example links to the PPS utilities file, `ppsUtils.js`, which is included as part of the `webplatform.js` file in the SDK directory. The namespace for `ppsUtils.js` is `qnx.webplatform.pps`. The files in the demo follow the structure described in the *"Structure of a plugin"* (p. 25) section.

Files in the example

This example includes the following files:

- `plugin.xml` in the `com.qnx.demo` directory. This file declares the namespace of the plugin, and describes the file structure.
- `demo.js` and `index.js` in the `com.qnx.demo/src/blacberry10` directory. The `demo.js` file initializes the extension and defines functions for handling events and returning PPS data. The `index.js` file sets up the events that are triggered.
- `client.js` in the `com.qnx.demo/www` directory. This file defines the externally visible methods that apps can use.

Let's look at some of these files in more detail.

demo.js

The `demo.js` file provides the core functionality of the plugin. First, we link to the `qnx.webplatform.pps` namespace and declare some variables we'll use later:

```
var _pps = qnx.webplatform.pps,
    _readerPPS,
    _writerPPS,
    _triggerUpdate;
```

Now we'll create the demo PPS object. There's additional code that handles errors and so on, but let's focus on the PPS object's creation:

```
_readerPPS = _pps.createObject("/pps/qnxcar/demo", _pps.PPSMode.DELTA);
_readerPPS.onNewData = function(event) {
            if (_triggerUpdate && event && event.data) {
                _triggerUpdate(event.data);
...

_writerPPS = _pps.createObject("/pps/qnxcar/demo", _pps.PPSMode.DELTA);

...
```

The preceding code creates a new PPS object (`/pps/qnxcar/demo`) with two handles: one for reading (`_readerPPS`) and one for writing (`_writerPPS`). These handles give us access to the methods in `qnx.webplatform.pps` that we can use for

manipulating the PPS data. The `_triggerUpdate` object is used in handling events. Here, we use it in defining the action to take when new data is available in our PPS object.

The `demo.js` file also defines the functions for handling the event trigger (*setTriggerUpdate()*) and for getting (*get()*) and setting (*set()*) the PPS data. These functions are exported so that they can be called from the `index.js` file:

```
/**
 * Sets the trigger function to call when an event is fired
 * @param trigger {Function} The trigger function to call
 *         when an event is fired
 */
setTriggerUpdate: function(trigger) {
    _triggerUpdate = trigger;
},

/**
 * Returns the demo object
 * @returns {Object} The demo object
 */
get: function(settings) {
     return _readerPPS.data.demo;
},

/**
 * Set a demo object field
 * @param {String} key The data key
 * @param {Mixed} value The data value
 */
set: function(key, value) {
     var data = {};
     data[key] = value;
     _writerPPS.write(data);
}
```

index.js

In the `index.js` file, we define the functions that we'll make available to our clients through the `client.js` file:

```
/**
* Starts triggering events
* @param {Function} success Function to call if the operation is a success
* @param {Function} fail Function to call if the operation fails
* @param {Object} args The arguments supplied
* @param {Object} env Environment variables
*/
startEvents: function(success, fail, args, env) {
    _eventResult = new PluginResult(args, env)
    try {
        _demo.setTriggerUpdate(function (data) {
            _eventResult.callbackOk(data, true);
        });
        _eventResult.noResult(true);
    } catch (e) {
        _eventResult.error("error in startEvents: " + JSON.stringify(e), false);
    }
},

/**
 * Stops triggering events
 * @param {Function} success Function to call if the operation is a success
 * @param {Function} fail Function to call if the operation fails
 * @param {Object} args The arguments supplied
 * @param {Object} env Environment variables
```

```
 */
stopEvents: function(success, fail, args, env) {
   var result = new PluginResult(args, env);
   try {
       //disable the event trigger
        _demo.setTriggerUpdate(null);
        result.ok(undefined, false);

        //cleanup
        _eventResult.noResult(false);
        delete _eventResult;
    } catch (e) {
        result.error("error in stopEvents: " + JSON.stringify(e), false);
    }
},

/**
 * Returns system settings
 * @param success {Function} Function to call if the operation is a success
 * @param fail {Function} Function to call if the operation fails
 * @param args {Object} The arguments supplied
 * @param env {Object} Environment variables
 */
get: function(success, fail, args, env) {
    var result = new PluginResult(args, env)
    try {
        var fixedArgs = _wwfix.parseArgs(args);
        var data = _demo.get();
        result.ok(data, false);
    } catch (e) {
        result.error(JSON.stringify(e), false);
    }
},

/**
 * Sets one or more system settings
 * @param success {Function} Function to call if the operation is a success
 * @param fail {Function} Function to call if the operation fails
 * @param args {Object} The arguments supplied
 * @param env {Object} Environment variables
 */
set: function(success, fail, args, env) {
    var result = new PluginResult(args, env)
    try {
        var fixedArgs = _wwfix.parseArgs(args);
        _demo.set(fixedArgs.key, fixedArgs.value);
        result.ok(undefined, false);
    } catch (e) {
        result.error(JSON.stringify(e), false);
    }
}
```

In the preceding code, we define four functions:

- *startEvents()* and *stopEvents()*, which use the *setTriggerUpdate()* function we defined in `index.js`. This function is referenced as `_demo.setTriggerUpdate()`.
- *get()*, which uses the *get()* function we defined in `index.js` (referenced here as `_demo.get()`).
- *set()*, which uses the *set()* function we defined in `index.js` (referenced here as `_demo.set()`).

client.js

The `client.js` file defines the public-facing interface, or client side, of our plugin. This is where we use the *cordova.exec()* function to call the PPS functions we've defined in our server-side files.

First, we declare our variables. We'll use `_ID` in our calls to *cordova.exec()*:

```
var _self = {},
    _ID = "com.qnx.demo",
    _utils = cordova.require('cordova/utils'),
    _watches = {};
```

We create a function to hande PPS update events. This function is also passed to *cordova.exec()*, but isn't accessible to other applications:

```
/**
 * Handles update events for this plugin
 * @param data {Array} The updated data provided by the event
 * @private
 */
function onUpdate(data) {
    var keys = Object.keys(_watches);
    for (var i=0; i<keys.length; i++) {
        setTimeout(_watches[keys[i]](data), 0);
    }
}
```

Finally, we define our public-facing functions and export them. For each one, we call *cordova.exec()* and pass it the name of the function to call from `index.js`:

```
/**
 * Watch for PPS object changes
 * @param {Function} callback The function to call when a change is detected.
 * @return {String} An ID for the added watch.
 * @example
 *
 * //define a callback function
 * function myCallback(myData) {
 *        //just send data to log
 *        console.log("Changed data: " , myData);
 *        }
 * }
 *
 * var watchId = car.demo.watchDemo(myCallback);
 */
_self.watchDemo = function (callback) {
    var watchId = _utils.createUUID();

    _watches[watchId] = callback;
    if (Object.keys(_watches).length === 1) {
        window.cordova.exec(onUpdate, null, _ID, 'startEvents', null, false);
    }

    return watchId;
}

/**
 * Stop watching for PPS changes
 * @param {Number} watchId The watch ID as returned by car.demo.watchDemo().
 * @example
 *
 * car.sensors.cancelWatch(watchId);
 */
_self.cancelWatch = function (watchId) {
    if (_watches[watchId]) {
        delete _watches[watchId];
        if (Object.keys(_watches).length === 0) {
            window.cordova.exec(null, null, _ID, 'stopEvents', null, false);
        }
    }
}

/**
 * Get the value of the demo PPS object
 * @returns {Object} The demo object contents.
 */
```

```
_self.get = function () {
        var value = null,
        success = function (data, response) {
            value = data;
        },
        fail = function (data, response) {
            throw data;
        };

    try {
        window.cordova.exec(success, fail, _ID, 'get', null);
    } catch (e) {
        console.error(e);
    }
    return value;
}

/**
 * Set a demo object field
 * @param {String} key The data key
 * @param {Mixed} value The data value
 */
_self.set = function (key, value) {
    window.cordova.exec(null, null, _ID, 'set', { key: key, value: value });
}

module.exports = _self;
```

Applications that use our plugin will need to define a callback function for Cordova to call when *watchDemo()* is successful.

Chapter 6
Enhancing Performance

The following best practices will help you develop more efficient apps.

1. Avoid Canvas.

   Canvas and SVG elements aren't optimal for use on mobile and embedded platforms, because proper hardware acceleration for these elements hasn't yet been finalized.

2. Avoid 2D transformations.

   Use 3D instead. For example, instead of `translateX(x)` use `translate3d(x,y,z)`. This will force hardware acceleration of the translation. You can use similar methods for most other transformations. **Avoid animating with JavaScript libraries!**

3. Avoid opacity, rounded corners, and gradients.

   These are a significant drain on performance when they're redrawn often. If used sparingly and mostly on static objects, they shouldn't impede performance, but if you mix these elements with animations, buttons, or anything that gets redrawn often, performance will suffer. Consider using images instead of heavy CSS.

4. Remove elements from the DOM when modifying them.

   This technique is especially helpful when you're updating several DOM fields at once. For example, if you're scrolling through a list of 100 contacts and you want to refresh them, updating them one by one will cause the list to be redrawn 100 times. But if you remove the entire list, update the contacts in memory, and then add the list again, this does only *two* redraws.

5. Hide elements you don't need.

   Adding `display:none` to elements that don't need to be displayed will prevent them from being rendered.

6. Avoid libraries intended for desktop use.

   Some JavaScript libraries are designed for use on a desktop browser with a powerful CPU. Try to limit the number of third-party JavaScript libraries included in your app or try to seek out versions optimized for mobile use.

7. Use image sprites.

   These are useful for preloading active element states (e.g., buttons with a “pressed” state).

Chapter 7
WebLauncher's JavaScript APIs

By using the Cordova framework and the Browser Chrome APIs, you can customize your browser chrome. The following API categories are available:

- *application* (p. 36)—functions and properties for the current application
- *webinspector* (p. 37)—functions related to the Web Inspector tool
- *webview* (p. 38)—functions to manage your webviews

WebLauncher `application` APIs

The application framework provides functionality to the browser engine to allow it to support apps. This environment lets you create and deploy apps built from web technologies (HTML5, CSS3, and JavaScript) with plugins that provide access to the underlying device hardware and native services, just like native C/C++ apps.

The following WebLauncher `application` APIs are available for you to implement your own browser chrome:

- `application.adduri`
- `application.bindToNetworkDevice`
- `application.coverSize`
- `application.extendTerminate`
- `application.getenv`
- `application.isDeviceLocked`
- `application.isForeground`
- `application.lockRotation`
- `application.minimizeWindow`
- `application.newWallpaper`
- `application.notifyRotateComplete`
- `application.realpath`
- `application.requestExit`
- `application.rotate`
- `application.setKeyboardTracking`
- `application.setPooled`
- `application.setSwipeStart`
- `application.setenv`
- `application.systemFontFamily`
- `application.systemFontSize`
- `application.systemRegion`
- `application.unlockRotation`
- `application.unsetenv`
- `application.updateCover`
- `applicationWindow.flushContext`
- `applicationWindow.setSize`
- `applicationWindow.setVisible`

WebLauncher `webinspector` APIs

Web Inspector is a useful debugging and profiling development tool for web content. Use this tool to troubleshoot and optimize your web content for your apps. The tool itself includes various features and capabilities, including inspection, profiling, console integration, and more.

The following WebLauncher `webinspector` APIs are available for you to implement your own browser chrome:

- `webInspector.enabled`
- `webInspector.enabledForWebView`
- `webInspector.port`
- `webInspector.setEnabled`
- `webInspector.setEnabledForWebView`

WebLauncher `webview` APIs

Every HTML5 app has its own WebView. A WebView is a view that's rendered by the web engine to display an app. The browser engine provides a set of core classes that you can use to display web content in a window. By default, the browser engine implements the most basic functionality of a browser, such as the ability to follow links and to download and display content. You can use the engine's functionality at the most basic level to display web content in your app or you can use APIs to create your own full-featured, customized, web-based app.

The QNX SDK for Apps and Media provides a multiprocess architecture that allows for multiple WebViews to share a common engine instance or run in their own engine instance. Trusted apps are run "in process", sharing a web engine instance. Other apps are run "out of process", protected from each other by process boundaries, each with their own web engine instance. You can implement each WebView with a separate JavaScript application framework, such as jQuery or Sencha Touch.

The following WebLauncher `webview` APIs are available for you to implement your own browser chrome:

- `webview.addKnownSSLCertificate`
- `webview.addKnownSSLCertificates`
- `webview.addOriginAccessWhitelistEntry`
- `webview.applicationSwipeInEvent`
- `webview.applicationWindowGroup`
- `webview.arguments`
- `webview.assignFocus`
- `webview.autofillTextField`
- `webview.automationLog`
- `webview.backgroundColor`
- `webview.bitmapZoom`
- `webview.canGoBack`
- `webview.canGoForward`
- `webview.cancelVibration`
- `webview.captureContents`
- `webview.certificateInfo`
- `webview.cleanupSSLCertificateDetails`
- `webview.clearAutofillData`
- `webview.clearBackForwardList`
- `webview.clearBrowsingData`
- `webview.clearCache`

- webview.clearCookies
- webview.clearCredentials
- webview.clearDatabases
- webview.clearFocus
- webview.clearHistory
- webview.clearLocalStorage
- webview.clearWebFileSystem
- webview.contentRectangle
- webview.continueSSLHandshaking
- webview.convertIDNtoReadableStringByLanguage
- webview.create
- webview.createFadeColorWindow
- webview.currentContext
- webview.defaultFontSize
- webview.defaultTextEncoding
- webview.delete
- webview.destroy
- webview.destroyFadeColorWindow
- webview.destroyIfNotRejectedByUser
- webview.devicePixelRatio
- webview.dialogResponse
- webview.downloadCancel
- webview.downloadPause
- webview.downloadRemove
- webview.downloadResume
- webview.downloadRetry
- webview.downloadUpdate
- webview.downloadUrl
- webview.enableQnxJavaScriptObject
- webview.encryptionInfo
- webview.eventNames
- webview.executeJavaScript
- webview.executeJavaScriptFunction
- webview.extraHttpHeaders
- webview.fadeColorWindowPlatformHandle
- webview.favicon
- webview.fileSystemAPISandboxed
- webview.findString
- webview.focusNextField

- webview.focusPreviousField
- webview.forcedTextEncoding
- webview.fullScreenVideoCapable
- webview.fullScreenVideoExited
- webview.getCookies
- webview.getSSLCertificateDetails
- webview.goBack
- webview.goForward
- webview.handleContextMenuResponse
- webview.handleWebInspectorMessageToBackend
- webview.historyLength
- webview.historyPosition
- webview.initialize
- webview.isActive
- webview.isAllPropertyChangedEventsEnabled
- webview.isAllWebEventsEnabled
- webview.isAlwaysShowKeyboardOnFocus
- webview.isAnyPropertyChangedEventsEnabled
- webview.isAnyWebEventsEnabled
- webview.isAutoDeferNetworkingAndJavaScript
- webview.isBlockPopups
- webview.isDeferNetworkingAndJavaScript
- webview.isEnableCookies
- webview.isEnableCredentialAutofill
- webview.isEnableCrossSiteXHR
- webview.isEnableDNSPrefetch
- webview.isEnableDefaultOverScrollBackground
- webview.isEnableDialogRequestedEvents
- webview.isEnableDiskCache
- webview.isEnableDownloadableBinaryFonts
- webview.isEnableFineCursorControl
- webview.isEnableFormAutofill
- webview.isEnableGeolocation
- webview.isEnableInputMethodSupport
- webview.isEnableInputNotifications
- webview.isEnableJavaScript
- webview.isEnableLocalAccessToAllCookies
- webview.isEnableMediaRTSP
- webview.isEnableNetworkResourceRequestedEvents

- webview.isEnablePlugins
- webview.isEnableSoundOnAnchorElementTouchEvents
- webview.isEnableSpatialNavigation
- webview.isEnableTextSelectionControls
- webview.isEnableWebInspector
- webview.isEnableWebSockets
- webview.isLoadImages
- webview.isPluginFullScreen
- webview.isPrivateBrowsing
- webview.isPropertyChangedEventEnabled
- webview.isVideoFullScreen
- webview.isVisible
- webview.isWebEventEnabled
- webview.isZoomToFitWidthOnLoad
- webview.javaScriptInterruptTimeout
- webview.jsScreenWindowHandle
- webview.knownSSLCertificate
- webview.knownSSLCertificates
- webview.loadFile
- webview.loadProgress
- webview.loadString
- webview.loadStringWithBase
- webview.loadURL
- webview.location
- webview.lockProperties
- webview.log
- webview.maximumScale
- webview.minimumFontSize
- webview.minimumScale
- webview.notificationClicked
- webview.notificationClosed
- webview.notificationError
- webview.notificationShown
- webview.notifyApplicationOrientationDone
- webview.notifyContextMenuCancelled
- webview.notifyDataReceived
- webview.notifyDone
- webview.notifyHeaderReceived
- webview.notifyOpen

- webview.notifySystemLowMemory
- webview.notifyViewportChanged
- webview.openWindowResponse
- webview.originalLocation
- webview.overScrollColor
- webview.printToStderr
- webview.printToStdout
- webview.reload
- webview.removeAllKnownSSLCertificates
- webview.removeGeolocationFilter
- webview.removeKnownSSLCertificate
- webview.removeOriginAccessWhitelistEntry
- webview.requestCurrentContextUpdate
- webview.requestSession
- webview.restoreSession
- webview.scale
- webview.scrollBy
- webview.scrollPosition
- webview.secureType
- webview.securityInfo
- webview.sensitivity
- webview.setActive
- webview.setAllPropertyChangedEventsEnabled
- webview.setAllWebEventsEnabled
- webview.setAllowGeolocation
- webview.setAllowNotification
- webview.setAllowUserMedia
- webview.setAllowWebInspection
- webview.setAlwaysShowKeyboardOnFocus
- webview.setApplicationActivationState
- webview.setApplicationOrientation
- webview.setAutoDeferNetworkingAndJavaScript
- webview.setBackgroundColor
- webview.setBitmapZooming
- webview.setBlockPopups
- webview.setCookies
- webview.setDefaultFontSize
- webview.setDefaultTextEncoding
- webview.setDeferNetworkingAndJavaScript

- webview.setDevicePixelRatio
- webview.setEnableCookies
- webview.setEnableCredentialAutofill
- webview.setEnableCrossSiteXHR
- webview.setEnableDNSPrefetch
- webview.setEnableDefaultOverScrollBackground
- webview.setEnableDialogRequestedEvents
- webview.setEnableDiskCache
- webview.setEnableDownloadableBinaryFonts
- webview.setEnableFineCursorControl
- webview.setEnableFormAutofill
- webview.setEnableGeolocation
- webview.setEnableInputMethodSupport
- webview.setEnableInputNotifications
- webview.setEnableJavaScript
- webview.setEnableLocalAccessToAllCookies
- webview.setEnableMediaRTSP
- webview.setEnableNetworkResourceRequestedEvents
- webview.setEnablePlugins
- webview.setEnableSoundOnAnchorElementTouchEvents
- webview.setEnableSpatialNavigation
- webview.setEnableTextSelectionControls
- webview.setEnableWebSockets
- webview.setEnabledOutOfProcessWebInspector
- webview.setExtraHttpHeaders
- webview.setExtraPluginDirectory
- webview.setFadeColorWindowRect
- webview.setFadeWindowColor
- webview.setFileSystemAPISandboxed
- webview.setForcedTextEncoding
- webview.setFullScreenVideoCapable
- webview.setGeometry
- webview.setHistoryPosition
- webview.setJSWebViewBindings
- webview.setJavaScriptInterruptTimeout
- webview.setKeyboardVisibilityLocked
- webview.setKeyboardVisible
- webview.setLayerTilerPrefillRect
- webview.setLoadImages

- webview.setMinimumFontSize
- webview.setOverScrollColor
- webview.setPatternMatchingEnabled
- webview.setPopupWebView
- webview.setPrivateBrowsing
- webview.setPropertyChangedEventEnabled
- webview.setScreenPowerState
- webview.setScrollPosition
- webview.setScrolling
- webview.setSensitivity
- webview.setStandalone
- webview.setTemporaryViewportSize
- webview.setTextReflowMode
- webview.setUserAgent
- webview.setUserStyleSheetLocation
- webview.setViewport
- webview.setViewportHeight
- webview.setViewportInitialScale
- webview.setViewportMaximumScale
- webview.setViewportMinimumScale
- webview.setViewportTargetDensityDpi
- webview.setViewportUserScalable
- webview.setViewportWidth
- webview.setVisible
- webview.setWebEventEnabled
- webview.setWebInspectorEnabled
- webview.setZOrder
- webview.setZoomFactor
- webview.setZoomToFitWidthOnLoad
- webview.status
- webview.stop
- webview.submitForm
- webview.syncProxyCredential
- webview.textEncoding
- webview.textHasAttribute
- webview.textReflowMode
- webview.title
- webview.tooltip
- webview.unlockProperties

- webview.updateDisabledPluginFiles
- webview.updateGeolocationFilter
- webview.updateNotificationPermission
- webview.userAgent
- webview.userStyleSheetLocation
- webview.viewport
- webview.viewportHeight
- webview.viewportInitialScale
- webview.viewportMaximumScale
- webview.viewportMinimumScale
- webview.viewportTargetDensityDpi
- webview.viewportUserScalable
- webview.viewportWidth
- webview.webInspectorPort
- webview.windowUniqueId
- webview.zOrder
- webview.zoomFactor

Chapter 8
Debugging Web Apps

Included as part of WebKit, *Web Inspector* is a useful debugging and profiling development tool for web content. You can use this tool to troubleshoot and optimize your web content for your apps. The tool includes a variety of features and capabilities, such as inspection, profiling, console integration, and more.

By default, Web Inspector functionality is disabled. To use Web Inspector in conjunction with the browser, you must first enable it in the browser options. For a Cordova application, you enable Web Inspector by specifying a command-line flag at compile time. For instructions on enabling, see "*Enabling Web Inspector* (p. 48)" in this guide.

To begin debugging your app, you can install the Google Chrome browser or another WebKit browser for your platform. Once Web Inspector is enabled, you can access it using a supported device in conjunction with the Chrome browser.

Enabling Web Inspector

Use Web Inspector to debug and profile your apps. The browser uses a client-server architecture to make Web Inspector functionality available by acting as a webserver. You inspect the content remotely on a browser; use any WebKit-based browser on the same Wi-Fi network to navigate to the IP address and port number used by the browser and to begin inspecting the code.

To enable Web Inspector for your app:

1. Navigate to your `cordova` apps directory.
2. Run the following command:

   ```
   build debug
   ```

   To launch your compiled app, you'll need to know the port number your app is running on. You can use `netstat -a` to see all of the currently opened ports. The listening port numbers start at 1337.

Launching Web Inspector

Before you begin to use Web Inspector, verify the following:

- Your computer is connected to your target board via Ethernet.
- Your computer has a WebKit-based desktop browser (e.g., Google Chrome or Apple Safari).
- You have launched your application on the target (e.g., by manually starting `we blauncher` or by sending a command to the launcher's PPS object via a command console).

After you've launched the app you want to inspect, you'll need to manually connect Web Inspector.

To launch Web Inspector to begin inspecting web content:

1. On your host computer, open your WebKit-based browser.
2. In the address bar, type the IP address of the device and specify the port number used by the application serving the content. Note that different instances of the browser will be at different ports.

   You'll be prompted with a list of page titles for content that Web Inspector has loaded into memory, such as browser content or an HTML5 app.
3. Click any of the page titles to begin using Web Inspector to debug and profile your web content.
   Web Inspector opens and displays the **Elements** panel.

In the Sources view, once you enable debugging to ensure you have the latest source files in the list, you'll need to refresh the current page by using the shortcut key **CTRL+R**.

In the Console view, you can browse objects, test extensions, and test APIs without having to build and deploy your app.

Debugging and profiling using Web Inspector

Web Inspector allows you to inspect and debug your webpage source code, inspecting web content displayed through the browser or in HTML5 applications. You can use Web Inspector to manipulate the DOM, edit and debug JavaScript code, analyze resource requests, and audit the performance of web content and web apps in near real time.

Using the WebKit-based browser on your desktop, you can navigate to the IP address and port number used by the server application and begin inspecting the code.

Web Inspector contains a number of panels that provides different functionality you can use to help improve the appearance and performance of your webpage:

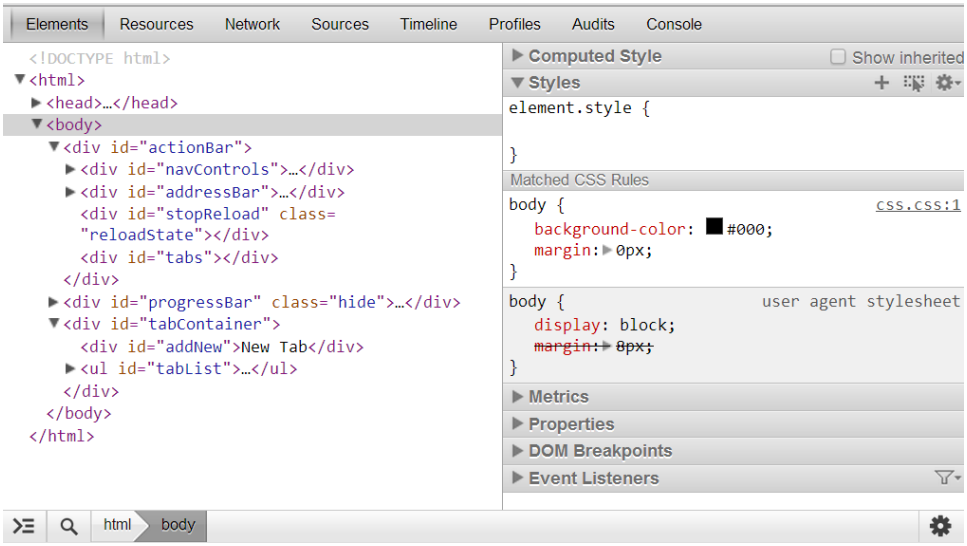
Panel	Description
Elements (p. 51)	Inspect the DOM of the current webpage and adjust settings for attributes and CSS properties. Changes you make are reflected in the browser.
Resources (p. 54)	Display information about all the resources used by the current webpage.
Network (p. 56)	Display information about each HTTP request made as resources are requested, received, and displayed in the browser or in your HTML5 app.
Sources (p. 58)	Debug JavaScript code. You can set breakpoints and step through your code to locate and correct issues.
Timeline (p. 60)	View how much time it takes for the browser to load and render the webpage and its resources.
Profiles (p. 62)	Examine how your JavaScript code utilizes memory. With the Profiles panel, you can pinpoint programmatic inefficiencies.
Audits (p. 65)	Examine the network utilization and webpage performance. The tool suggests ways to improve performance.
Console	A command-line utility that lets you debug JavaScript or HTML parsing errors.

Optimizing layout and style

As you try to achieve an optimal layout for small screens such as those on BlackBerry smartphones, the **Elements** panel can be a very useful tool. The **Elements** panel allows you to view the DOM and trace style values for an element to see where values are applied, how they have been inherited, and where style values have been superseded. You can adjust style settings to tweak the appearance of the webpage components to achieve the desired result. The changes you make to the webpage are applied in near real time in the BlackBerry Browser, so you can see how your changes affect the appearance of the content. Once you achieve the results you want, you can propagate the updated values into the source file.

The Elements panel

You can click the **Elements** icon on the toolbar to display the **Elements** panel.



The **Elements** panel is divided into two sections. On the left is the document pane, which displays the DOM tree of the HTML source document. Each element is displayed as a separate node. You can expand the nodes of the DOM tree to view the children of a container element. The document pane of the **Elements** panel is a good tool to use to view the source of a page. Since the panel displays the page as a tree, the document is easy to view and to navigate, even when the original webpage is minified or poorly formatted and difficult to read. Within the document pane, you can edit aspects of the DOM, such as attribute values or text.

On the right is a set of collapsible panes that display various pieces of information related to the element currently selected in the document pane. Some of these panes, such as **Computed Style** and **Event Listeners**, are informative; you use them simply to track information about the element. Other panes are editable and let you change

the styles or properties associated with the selected element. You can edit content in the following panes:

- **Styles**: The **Styles** pane is divided into sections that show each matched CSS rule and the associated style declarations. It also displays style values that have been inherited. Inherited values that have been overwritten by other style declarations are shown with strikethough text.
- **Metrics**: The **Metrics** pane provides a visual representation of the box model, which you can edit to optimize the layout of a container element on the screen. The box model refers to the amount of space a container element occupies in a rendered webpage. You can apply styles such as margins, borders, and padding to an element to adjust the size of the content block and improve the page layout.
- **Properties**: The **Properties** pane allows you to view the page as it is seen by JavaScript code—as a collection of DOM objects with associated property values. Although some of the property values are editable, in most cases it's easier to edit style values in the **Styles** pane.

Inspect and modify element styles

1. Click the **Elements** icon on the toolbar to display the **Elements** panel.
2. In the document pane, locate and select the element with the style you want to modify.
3. In the data sidebar, expand the **Styles** pane to display the style declarations applied to the selected element.
4. Perform any of the following actions:
   - To change the value for a style declaration, double-click the value in the Styles pane to make the value editable, then type the new value. You can use the **Tab** key to cycle through the declarations within a selector to modify more than one value.
   - To disable a style declaration, deselect the adjacent check box.
   - To add a new style declaration for a selector, double-click the white space below the last style declaration, then type the new declaration.
   - To modify the selector, double-click the selector, then type the new selector value.
5. When you're done, propagate your changes to the source document.

Inspect and modify the DOM

1. Click the **Elements** icon on the toolbar to display the **Elements** panel.
2. In the document pane, navigate to the node that you want to view or change.
3. Perform any of the following actions:

- To change the value of an attribute, double-click the value in the document pane to make the value editable, then type the new value. Use the **Tab** key to cycle through each of the element's attributes to modify more than one value.
- To change an attribute name, double-click the attribute name and type the new value.

4. When you're done, propagate the changes to the source document.

Modify the box model for an element

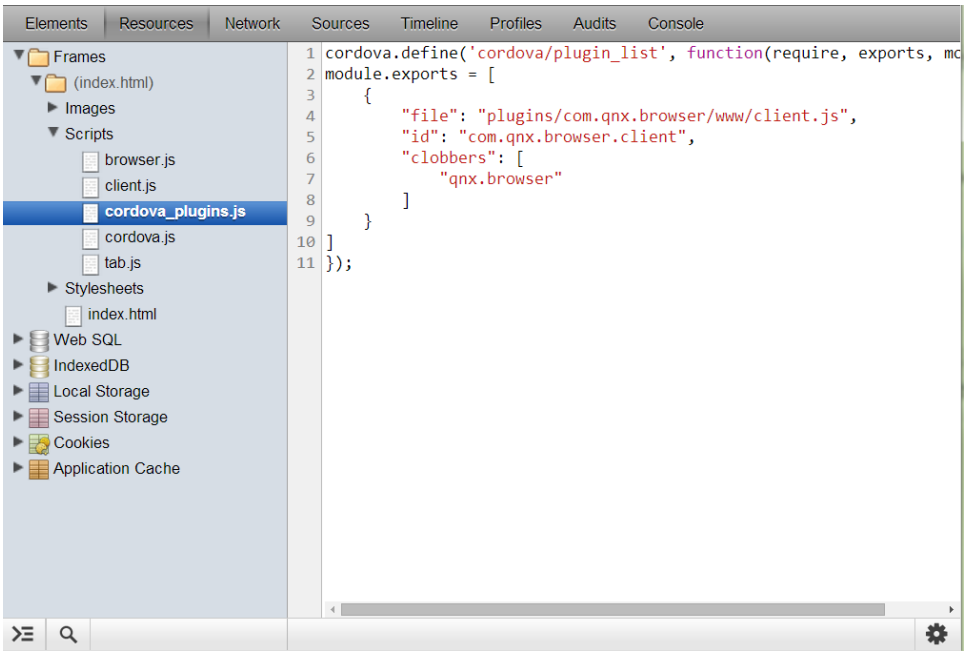
1. Click the **Elements** icon on the toolbar to display the **Elements** panel.
2. In the **Elements** panel, expand the **Metrics** pane to display the *box model* associated with the selected element.
3. Click any of the top, bottom, left, or right values, then type the new value. The changed value is propagated to the associated style declaration in the **Styles** pane.
4. When you're done, propagate the changes to the source document.

Analyzing page resources

The **Resources** panel allows you to view the complete set of resources that are loaded by a webpage. You can view and interact with resources such as CSS and JavaScript, check image content and information, and view which font sets are used on the page. You can also view and interact with any client-side resources created by your page, including cookies, databases, storage objects, and application cache.

The Resources panel

You can click the **Resources** icon on the toolbar to display the **Resources** panel.



The **Resources** panel shows a complete list of the resources that the WebKit engine must request and load to render the webpage, as well as any client-side resources created and used by the webpage. You can also use the Resources panel to view the content of any resource file. Resources are organized in the panel as follows:

- **Frames**: Contains the resources for each frame displayed in the content, including images, fonts, scripts, style sheets, and other content resources (e.g., embedded video or Flash files). Subframes within the main window are displayed as subfolders beneath the main Frames folder.
- **Databases**: Contains all the database tables that are associated with your content or app.
- **Local Storage**: Contains all *Local Storage* objects, that is, storage objects that persist after a browser session has ended.
- **Session Storage**: Contains all *Session Storage* objects, that is, storage objects that are valid only for the current browser session.

- **Cookies**: Contains all the cookies associated with the webpage or app.
- **Application Cache**: Contains the resources included in the manifest of an offline web application.

View resource content

1. Click the **Resources** icon on the toolbar to display the **Resources** panel.
2. In the list in the left pane of the **Resources** panel, double-click a category to show the resources and subgroups. Continue to drill down until you locate the resource you want to view.
3. Double-click the resource in the left pane. The right pane shows the contents of that resource. For example, selecting an image resource displays the image itself, along with the file size and URL of the image file. Selecting a script or style sheet shows the content of that script or style sheet.

View resource network information

*You can quickly see additional information about a specific resource by viewing the resource in the **Network** panel, which displays information such as file size and transfer rate information.*

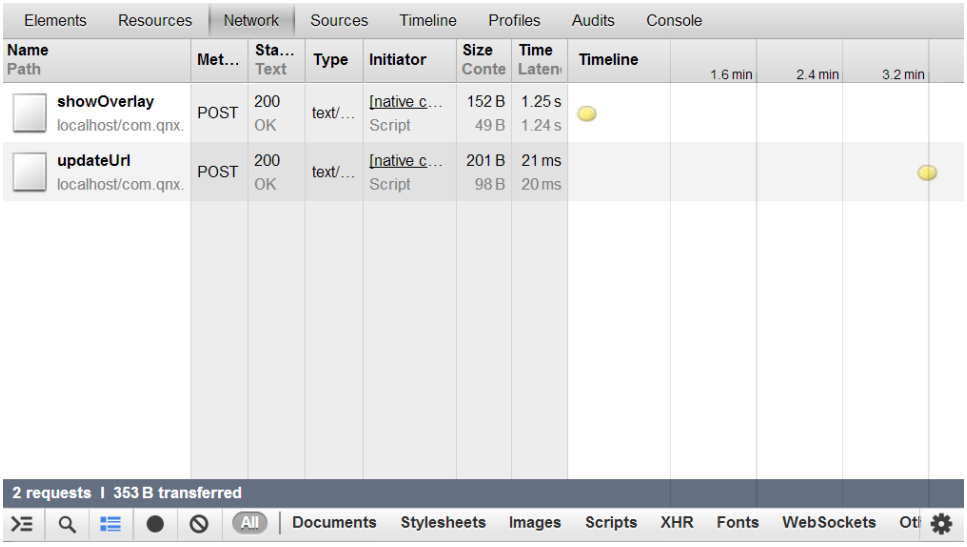
1. In the list in the left pane of the **Resources** panel, double-click a category to show the resources and subgroups. Continue to drill down until you locate the resource you're interested in.
2. Right-click the resource and click **Reveal in Network Panel**. Web Inspector opens the **Network** panel and briefly highlights the selected resource.

Analyzing network usage

The **Network** panel allows you to determine the network efficiency of your content. The panel displays information about each HTTP request made as the browser engine requests and downloads resources.

The Network panel

Click the **Network** icon in the toolbar to display the **Network** panel. Initially, the panel shows no information; you must reload the content on the device or simulator to allow Web Inspector to track the HTTP requests. When loading is complete, the **Network** panel displays a table.



By default, the table lists each of the requested resources in their requested order, and then charts the network activity as a waterfall timeline, with resources color-coded by type.

The waterfall timeline plots resources by the total time required to load the resource, from the initial request to the completion of the download. The pale segment of the resource bar in the chart represents the total latency, that is, the time the browser engine must wait from the moment it initially makes the request to the moment it receives the first packet of data for the resource. Two vertical lines on the chart show key page-load milestones:

- The blue line indicates the time when parsing of the content is complete and the `DOMContent` event fires.
- The red line indicates the time when all the resources have been loaded and the `load` event fires.

You can customize how the content is displayed in the **Network** panel by filtering based on type or sorting by any of the table headings. You can also reformat the chart to highlight different time measures.

Apply a filter to display a specific resource type

*By default, the **Networks** panel displays all resource requests in the table. The status bar at the bottom of the panel contains buttons that allow you to filter the resources displayed based on the resource type.*

1. Click the **Network** icon on the toolbar to display the **Network** panel.
2. If you haven't already done so, on the device or simulator, reload the page to allow Web Inspector to track and record network activity.
3. In the status bar at the bottom of the **Network** panel, choose the type of resource you want to display.

Change which time measure is displayed

By default, when you measure the network activity, Web Inspector charts the network activity in a waterfall timeline. You can reformat the chart to highlight different time measures.

1. Click the **Network** icon on the toolbar to display the **Network** panel.
2. If you haven't already done so, on the device or simulator, reload the page to allow Web Inspector to track and record network activity.
3. In the drop-down list above the chart, select one of the following:
   - **Timeline**: Displays the network activity in a waterfall timeline.
   - **Start time**: Highlights the time when each resource was requested.
   - **Response time**: Highlights the time when the resource is initially received.
   - **End time**: Highlights the time when the resource is completely loaded.
   - **Duration**: Displays the total length of time it takes to load the resource.
   - **Latency**: Displays the amount of delay between the start time value and the response time value.

Reorder the list of resources

1. Click the **Networks** icon on the toolbar to display the **Networks** panel.
2. If you haven't already done so, on the device or simulator, reload the page to allow Web Inspector to track and record network activity.
3. Click a column heading to reorder the list based on the column data.

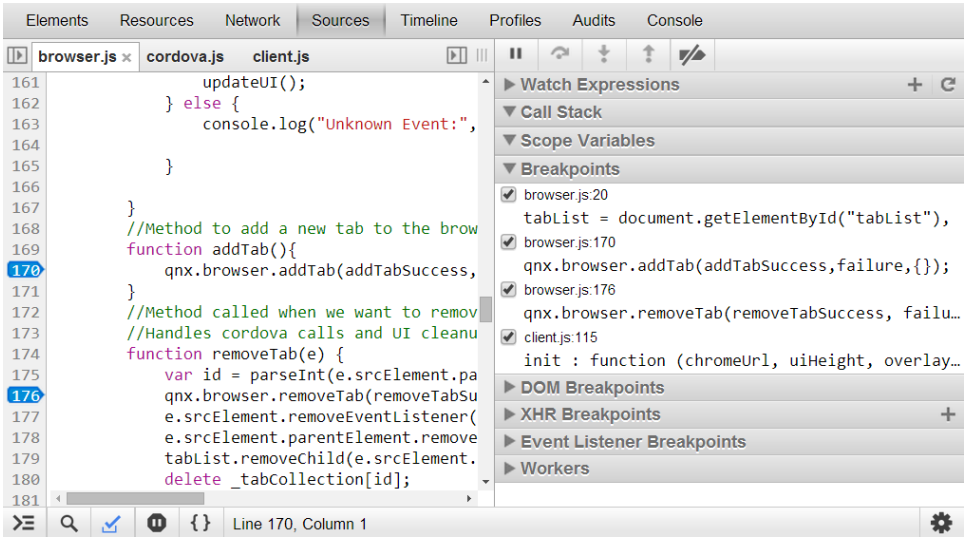
Debugging scripts

The **Sources** panel allows you to debug the JavaScript code used by your webpage. By allowing you to set breakpoints and to step through your code, the Web Inspector can help to locate and correct problems within your code. When you determine that the script is functioning as intended, you can copy the changes back into the source file.

To use the **Sources** panel, you must first enable debugging. When you first view the Sources panel, Web Inspector prompts you to enable debugging for just the current session or for all sessions.

The Sources panel

You can click the **Sources** icon in the toolbar to display the **Sources** panel. If you haven't already enabled debugging, Web Inspector prompts you to do so.



The **Sources** panel is divided into two sections. On the left is the document pane, which allows you to view and debug JavaScript. On the right is a set of collapsible panes that display information related to the displayed script.

A toolbar at the top of the **Sources** panel allows you to choose the script file you want to inspect and to cycle between open scripts. It also provides a set of controls that allow you to step through the script displayed in the document pane.

Set and use breakpoints

1. Click the Sources icon on the toolbar to display the Sources panel.
2. In the line gutter of the document pane, click the line where you want to set a breakpoint. A breakpoint marker appears in the line gutter and the new breakpoint is added to the **Breakpoints** pane, identified by the script filename and line number. The execution of the script pauses at the specified breakpoint.

3. Perform any of the following actions:

   - To continue the execution of the script beyond the current breakpoint, click the **Continue** button in the **Sources** panel toolbar.
   - To display the line of code associated with the breakpoint in the documents pane, click the breakpoint entry in the **Breakpoints** pane. The document pane displays and highlights the associated line.
   - To disable a single breakpoint without removing it, in the **Breakpoints** pane, uncheck the breakpoint. The execution of the script no longer pauses at the disabled breakpoint.
   - To deactivate or activate all the breakpoints listed in the **Breakpoints** pane without removing them, toggle the breakpoint activation switch at the right side of the **Scripts** panel toolbar.
   - To remove a breakpoint, locate and click the breakpoint marker in the line gutter of the document pane. The marker no longer appears in the line gutter and the breakpoint is removed from the **Breakpoints** pane.

Pause script execution

You can pause the script at any time to get a snapshot of the call stack and variable values.

1. Click the **Sources** icon on the toolbar to display the **Sources** panel.
2. In the **Sources** panel toolbar, click the **Pause** button.

When the script pauses, the last line of JavaScript to be executed is highlighted. The call stack and the current in-scope variable values appear in the appropriate panes at the right of the panel.

Pause script execution on exceptions

You can configure Web Inspector to pause the execution of scripts whenever exceptions are thrown. A tri-state toggle allows you to specify whether to pause for all exceptions, for only uncaught exceptions, or for no exceptions.

1. Click the **Sources** icon on the toolbar to display the **Sources** panel.
2. Use the **Exceptions** button in the status bar at the bottom to choose one of the following behaviors:

   - To pause on all exceptions, click the **Exceptions** button until the icon turns blue.
   - To pause only on uncaught exceptions, click the **Exceptions** button until the icon turns red.
   - To not pause on any exceptions, click the **Exceptions** button until the icon turns gray.

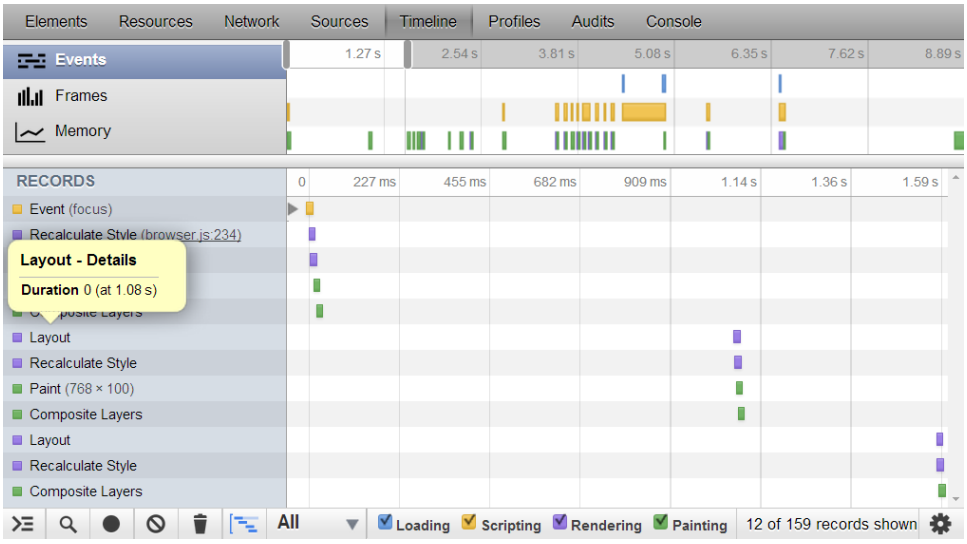
Analyzing loading, script execution, and rendering times

You can use the **Timeline** panel to analyze the time it takes to complete the different activities that the browser engine must perform to completely load and render your webpage.

The Timeline panel

Initially, the panel displays no information, so you must click the **Record** button in the status bar to allow Web Inspector to record the browser engine activity.

As it records browser engine activity, Web Inspector adds data to the **Timeline** panel.



Note that all browser engine activity pauses when the device is locked or the browser or HTML5 application are minimized. In order for Web Inspector to record any activity, the browser or HTML5 application must be the active application and the device or simulator screen mustn't be locked.

The **Timeline** panel is divided into two panes:

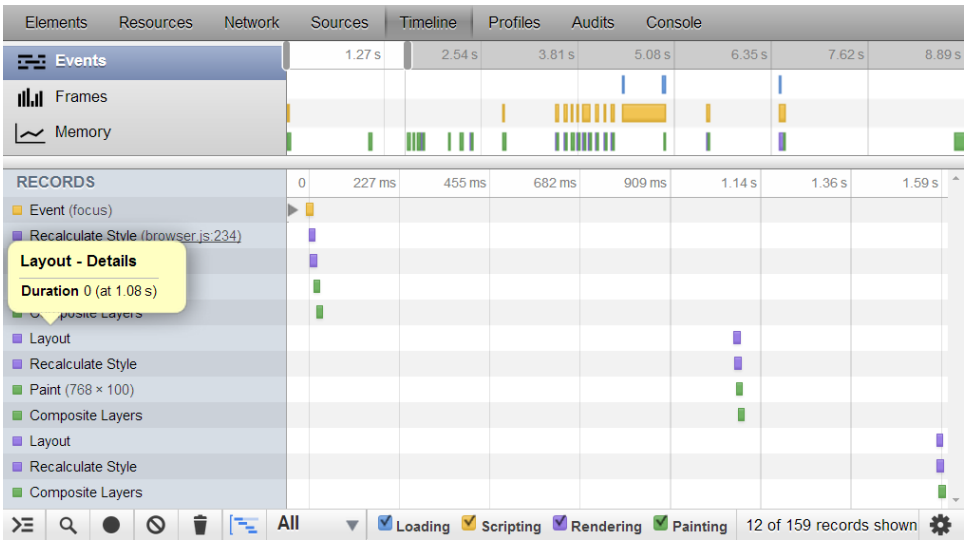
- In the top pane, the **Timeline** panel allows you select which timeline view you want to display. You can choose three views:
  - **Events**: Shows the time it takes for the browser engine to complete each of the events required to completely load the content.
  - **Frames**: Shows the browser engine activity for each screen refresh.
  - **Memory**: Shows memory consumption over time.
- In the lower pane, the **Timeline** panel shows a waterfall timeline for the timespan that was selected in the top pane. The data in the timeline is determined by the mode you select in the top pane of the timeline's panel.

Record browser engine activity

1. Click the **Timeline** icon on the toolbar to display the **Timeline** panel.
2. In the status bar at the bottom of the **Timeline** panel, click the record button to begin recording browser engine activity. While Web Inspector is recording, the record button turns red.

Constrain the display to a specific time span

*You can constrain the time span shown in the timeline. In the top pane of the **Timeline** panel, the portion of time displayed in the lower pane is represented by a white background. Two gray slider handles at the top left and right edges of this white background allow you to increase or decrease the selected timespan displayed in the timeline.*



1. Click the **Timeline** icon on the toolbar to display the **Timeline** panel.
2. If necessary, record the browser engine activity to generate timeline data.
3. In the top pane of the **Timeline** panel, click and drag a gray slider handle to increase or decrease the time span.

Filter which events are displayed

*By default, the **Timeline** panel shows all events in the table. The status bar at the bottom of the panel contains check boxes that allow you to show and hide events based on type.*

1. If necessary, record the browser engine activity to generate timeline data.
2. In the status bar at the bottom of the **Timeline** panel, deselect the event types you want to remove from the timeline.

Analyzing memory usage and processing demands

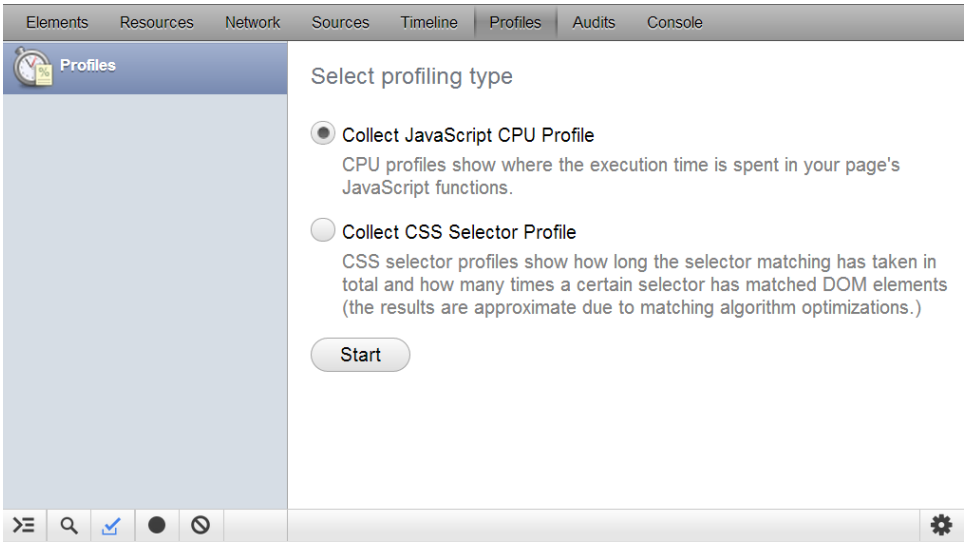
The **Profiles** panel allows you to analyze the memory usage and processing demands of your content. You can use the **Profiles** panel to create a performance profile for your JavaScript and CSS files.

- For JavaScript files, Web Inspector examines and reports on the CPU usage for each function. You can view the CPU usage for a particular function and for the number of times that function was called.
- For CSS files, Web Inspector examines the processing demands for each selector. Web Inspector records the amount of time it took to search for matches for a particular selector and for the total number of matches for that selector.

To use the **Profiles** panel, you must first enable profiling. When you first view the **Profiles** panel, Web Inspector prompts you to enable profiling for just the current session or for all sessions.

The Profiles panel

If you haven't already enabled profiling, Web Inspector prompts you to do so.

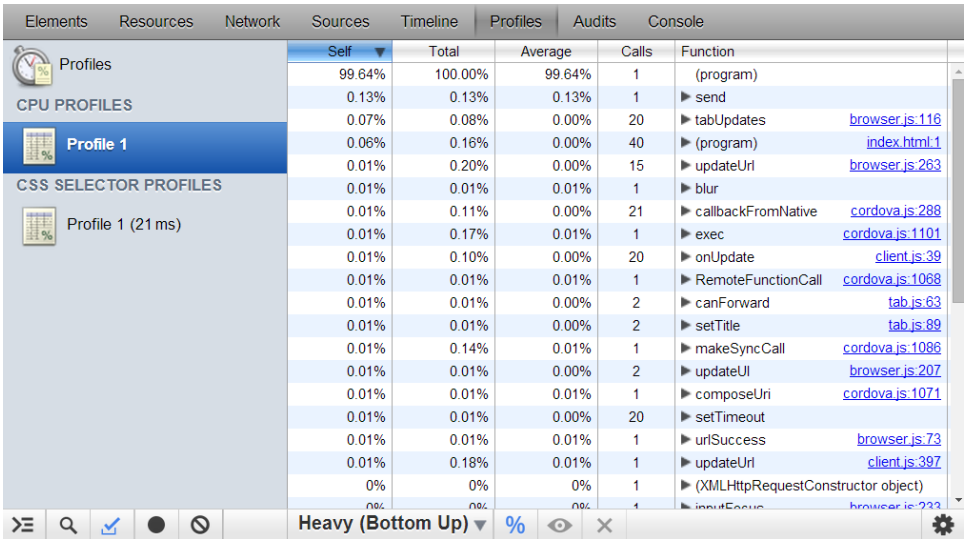


Profile the memory usage of your scripts

To profile memory usage:

1. Click the Profiles icon on the toolbar to display the Profiles panel.
2. On the Profiles panel, select Collect JavaScript CPU Profile.
3. To start profiling your memory usage, click Start. The button turns red as the Web Inspector is recording the memory usage.

4. To stop recording, click **Stop**. When you stop recording, the new profile is added under the CPU Profiles section in the left pane and the profile's contents are displayed in the right pane:



The results indicate the amount of time the browser engine spent executing each function during the recording process, along with the number of times each function was called. An excessive amount of time spent in any one function can indicate a problem with the code.

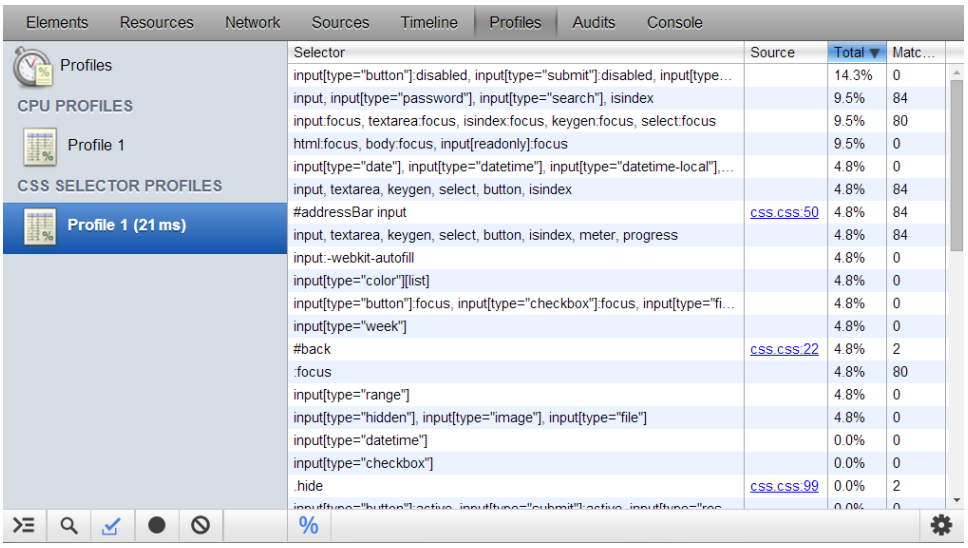
5. To sort the data, perform any of the following actions:

   - To sort by values in any column, double-click the column heading.
   - To display calls based on greatest impact on all exceptions or where they occurred in the call stack, in the status bar at the bottom of the panel, toggle between **Heavy (Bottom Up)** and **Tree (Top Down)**.
   - To specify whether values are presented as a time value or as a percentage of the total CPU usage required to process all the functions, toggle the percent button on or off.
   - To view a single function, select the call in the table and then click the focus button.
   - To exclude a single function from the data, select the function in the table and then click the exclude button.
   - To reload the original profile after focusing on or excluding a function, click the reload button.

Profile the performance of your CSS selectors

1. On the **Profiles** panel, select **Collect CSS Selector Profile**.

2. To start profiling your memory usage, click **Start**. The button turns red as Web Inspector records the memory usage.
3. To stop recording, click **Stop**. When you stop recording, the new profile is added under the **CSS Selector Profiles** section in the left pane and the profile's contents are displayed in the right pane:



The profile results indicate the amount of time the browser engine spent matching each selector in the associated style sheets, along with the total number of times the browser engine found a match for the selector.

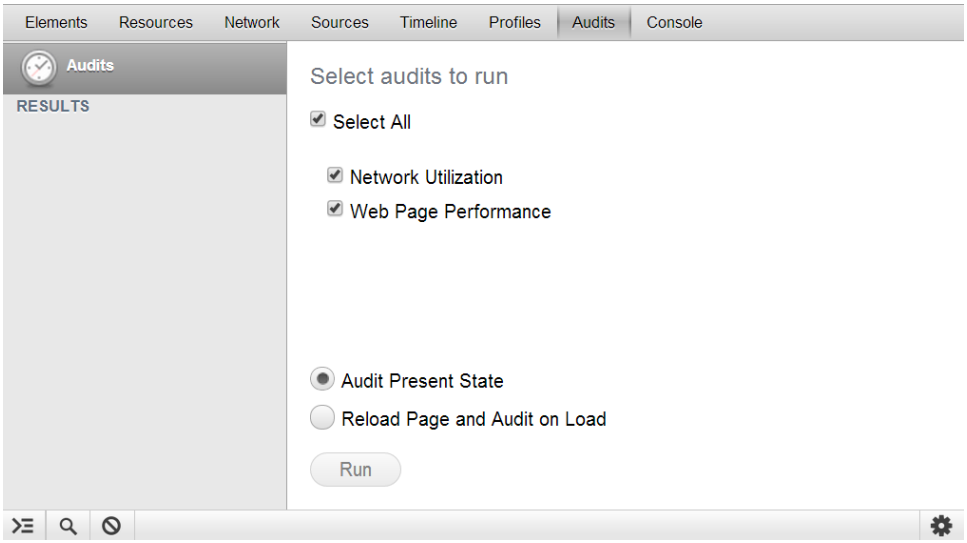
4. To specify whether the value of the **Total** column is presented as a time value or as a percentage of the total time required to process the CSS, toggle the percent button on or off.

Auditing your webpage

Web Inspector can audit your webpage for inefficiencies and, based on a set of best practices for web design, suggest changes you can make that can help improve network utilization and performance. The **Audits** panel provides a list of perceived inefficiencies in your webpage design. For example, Web Inspector can analyze your resources and determine where you might consider combining script files or style sheets. The **Audits** panel can also inform you where you've needlessly downloaded styles that aren't used, specify resources where you haven't set cache-control directives, and suggest other optimizations.

The **Audits** panel can be especially helpful when you design pages for mobile browsers. On mobile browsers, network latency can extend download times; constrained processing power tends to increase rendering time and to slow webpage performance. As a result, eliminating inefficiencies in your webpage design can have an significant positive impact.

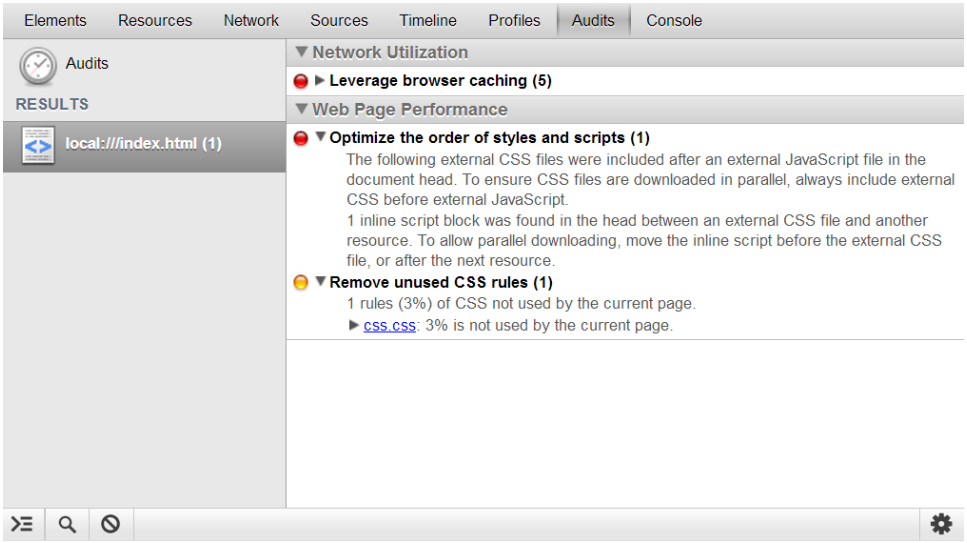
The Audits panel



The **Audits** panel lets you choose to:

- audit network utilization or page performance (or both)
- run the audit against the static page
- reload the page and run the audit as it loads.

Once you've run an audit, Web Inspector adds the report to the list at the left of the panel and shows the results in the main pane. The results suggest improvements you can make to your webpage to increase efficiencies.



Index