

FPT Software

HTML5 Programming Coursebook, FPT Software, 2012

Mobile Programming

FSB.FMC



2012

Contents At Glance

Chapter 1: HTML5	3
Chapter 2: PhoneGap.....	44
Chapter 3: Sencha Touch	55

Chapter 1: HTML5



What is HTML5?

HTML5 will be the new standard for HTML.

The previous version of HTML, HTML 4.01, came in 1999. The web has changed a lot since then.

HTML5 is still a work in progress. However, the major browsers support many of the new HTML5 elements and APIs.

How Did HTML5 Get Started?

HTML5 is a cooperation between the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG).

WHATWG was working with web forms and applications, and W3C was working with XHTML 2.0. In 2006, they decided to cooperate and create a new version of HTML.

Some rules for HTML5 were established:

- New features should be based on HTML, CSS, DOM, and JavaScript
- Reduce the need for external plugins (like Flash)
- Better error handling
- More markup to replace scripting
- HTML5 should be device independent
- The development process should be visible to the public

The HTML5 <!DOCTYPE>

In HTML5 there is only one <!doctype> declaration, and it is very simple:

```
< !DOCTYPE html>
```

Minimum HTML5 Document

Below is a simple HTML5 document, with the minimum of required tags:

```
< !DOCTYPE html>
< html>
< head>
< title>Title of the document</title>
< /head>

< body>
The content of the document.....
< /body>

< /html>
```

HTML5 - New Features

Some of the most interesting new features in HTML5:

- The <canvas> element for 2D drawing
- The <video> and <audio> elements for media playback
- Support for local storage
- New content-specific elements, like <article>, <footer>, <header>, <nav>, <section>
- New form controls, like calendar, date, time, email, url, search

Browser Support for HTML5

HTML5 is not yet an official standard, and no browsers have full HTML5 support.

But all major browsers (Safari, Chrome, Firefox, Opera, Internet Explorer) continue to add new HTML5 features to their latest versions.

New Elements in HTML5

The internet has changed a lot since HTML 4.01 became a standard in 1999.

Today, some elements in HTML 4.01 are obsolete, never used, or not used the way they were intended to. These elements are removed or re-written in HTML5.

To better handle today's internet use, HTML5 includes new elements for better structure, better form handling, drawing, and for media content.

New Semantic/Structural Elements

HTML5 offers new elements for better structure:

Tag	Description
<article>	Defines an article
<aside>	Defines content aside from the page content
<bdi>	Isolates a part of text that might be formatted in a different direction from other text outside it
<command>	Defines a command button that a user can invoke
<details>	Defines additional details that the user can view or hide
<summary>	Defines a visible heading for a <details> element
<figure>	Specifies self-contained content, like illustrations, diagrams, photos, code listings, etc.
<figcaption>	Defines a caption for a <figure> element
<footer>	Defines a footer for a document or section
<header>	Defines a header for a document or section
<hgroup>	Groups a set of <h1> to <h6> elements when a heading has multiple levels
<mark>	Defines marked/highlighted text
<meter>	Defines a scalar measurement within a known range (a gauge)
<nav>	Defines navigation links
<progress>	Represents the progress of a task
<ruby>	Defines a ruby annotation (for East Asian typography)
<rt>	Defines an explanation/pronunciation of characters (for East Asian typography)
<rp>	Defines what to show in browsers that do not support ruby annotations
<section>	Defines a section in a document
<time>	Defines a date/time
<wbr>	Defines a possible line-break

New Media Elements

HTML5 offers new elements for media content:

Tag	Description
<audio>	Defines sound content
<video>	Defines a video or movie
<source>	Defines multiple media resources for <video> and <audio>
<embed>	Defines a container for an external application or interactive content (a plug-in)
<track>	Defines text tracks for <video> and <audio>

The new <canvas> Element

Tag	Description
<canvas>	Used to draw graphics, on the fly, via scripting (usually JavaScript)

New Form Elements

HTML5 offers new form elements, for more functionality:

Tag	Description
<datalist>	Specifies a list of pre-defined options for input controls
<keygen>	Defines a key-pair generator field (for forms)
<output>	Defines the result of a calculation

Removed Elements

The following HTML 4.01 elements are removed from HTML5:

- <acronym>
- <applet>
- <basefont>
- <big>
- <center>
- <dir>
-
- <frame>
- <frameset>
- <noframes>
- <strike>
- <tt>

HTML5 Video

Many modern websites show videos. HTML5 provides a standard for showing them.

Video on the Web

Until now, there has not been a standard for showing a video/movie on a web page.

Today, most videos are shown through a plug-in (like flash). However, different browsers may have different plug-ins.

HTML5 defines a new element which specifies a standard way to embed a video/movie on a web page: the <video> element.

Browser Support



Internet Explorer 9, Firefox, Opera, Chrome, and Safari support the <video> element.

Note: Internet Explorer 8 and earlier versions, do not support the < video> element.

HTML5 Video - How It Works

To show a video in HTML5, this is all you need:

```
<video width="320" height="240" controls="controls">  
  <source src="movie.mp4" type="video/mp4" />  
  <source src="movie.ogg" type="video/ogg" />  
  Your browser does not support the video tag.  
</video>
```



The control attribute adds video controls, like play, pause, and volume.

It is also a good idea to always include width and height attributes. If height and width are set, the space required for the video is reserved when the page is loaded. However, without these attributes, the browser does not know the size of the video, and cannot reserve the appropriate space to it. The effect will be that the page layout will change during loading (while the video loads).

You should also insert text content between the `<video>` and `</video>` tags for browsers that do not support the `<video>` element.

The `<video>` element allows multiple `<source>` elements. `<source>` elements can link to different video files. The browser will use the first recognized format.

Video Formats and Browser Support

Currently, there are 3 supported video formats for the `<video>` element: MP4, WebM, and Ogg:

Browser	MP4	WebM	Ogg
Internet Explorer 9	YES	NO	NO
Firefox 4.0	NO	YES	YES
Google Chrome 6	YES	YES	YES
Apple Safari 5	YES	NO	NO
Opera 10.6	NO	YES	YES

- MP4 = MPEG 4 files with H264 video codec and AAC audio codec
- WebM = WebM files with VP8 video codec and Vorbis audio codec
- Ogg = Ogg files with Theora video codec and Vorbis audio codec

HTML5 Video Tags

Tag	Description
<code><video></code>	Defines a video or movie
<code><source></code>	Defines multiple media resources for media elements, such as <code><video></code> and <code><audio></code>
<code><track></code>	Defines text tracks in media players

HTML5 Video + DOM

The HTML5 <video> element also has methods, properties, and events.

There are methods for playing, pausing, and loading, for example. There are properties (e.g. duration, volume, seeking) that you can read or set. There are also DOM events that can notify you, for example, when the <video> element begins to play, is paused, is ended, etc.

Example: please visit http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_video_js_prop.

HTML5 <video> - Methods, Properties, and Events

The table below lists the video methods, properties, and events supported by most browsers:

Methods	Properties	Events
play()	currentSrc	play
pause()	currentTime	pause
load()	videoWidth	progress
canPlayType	videoHeight	error
	duration	timeupdate
	ended	ended
	error	abort
	paused	empty
	muted	emptied
	seeking	waiting
	volume	loadedmetadata
	height	
	width	

Note: Of the video properties, only videoWidth and videoHeight are immediately available. The other properties are available after the video's meta data has loaded.

HTML5 Audio

HTML5 provides a standard for playing audio files.

Until now, there has not been a standard for playing audio files on a web page.

Today, most audio files are played through a plug-in (like flash). However, different browsers may have different plug-ins.

HTML5 defines a new element which specifies a standard way to embed an audio file on a web page: the <audio> element.

Browser Support



Internet Explorer 9, Firefox, Opera, Chrome, and Safari support the <audio> element.

Note: Internet Explorer 8 and earlier versions, do not support the < audio> element.

HTML5 Audio - How It Works

To play an audio file in HTML5, this is all you need:

Example

```
<audio controls="controls">
  <source src="song.ogg" type="audio/ogg" />
  <source src="song.mp3" type="audio/mpeg" />
  Your browser does not support the audio element.
</audio>
```

The control attribute adds audio controls, like play, pause, and volume.

You should also insert text content between the <audio> and </audio> tags for browsers that do not support the <audio> element.

The <audio> element allows multiple <source> elements. <source> elements can link to different audio files. The browser will use the first recognized format.

Audio Formats and Browser Support

Currently, there are 3 supported file formats for the <audio> element: MP3, Wav, and Ogg:

Browser	MP3	Wav	Ogg
---------	-----	-----	-----

Internet Explorer 9	YES	NO	NO
Firefox 4.0	NO	YES	YES
Google Chrome 6	YES	YES	YES
Apple Safari 5	YES	YES	NO
Opera 10.6	NO	YES	YES

HTML5 Audio Tags

Tag	Description
<u><audio></u>	Defines sound content
<u><source></u>	Defines multiple media resources for media elements, such as <video> and <audio>

HTML5 Drag and Drop

Drag and drop is a part of the HTML5 standard.

Drag and drop is a very common feature. It is when you "grab" an object and drag it to a different location.

In HTML5, drag and drop is part of the standard, and any element can be draggable.

Browser Support



Internet Explorer 9, Firefox, Opera 12, Chrome, and Safari 5 support drag and drop.

Note: Drag and drop does not work in Safari 5.1.2.

Make an Element Draggable

First of all: To make an element draggable, set the draggable attribute to true:

```
<img draggable="true" />
```

What to Drag - ondragstart and setData()

Then, specify what should happen when the element is dragged.

In the example above, the ondragstart attribute calls a function, drag(event), that specifies what data to be dragged.

The dataTransfer.setData() method sets the data type and the value of the dragged data:

```
function drag(ev)
{
    ev.dataTransfer.setData("Text",ev.target.id);
}
```

In this case, the data type is "Text" and the value is the id of the draggable element ("drag1").

Where to Drop - ondragover

The ondragover event specifies where the dragged data can be dropped.

By default, data/elements cannot be dropped in other elements. To allow a drop, we must prevent the default handling of the element.

This is done by calling the event.preventDefault() method for the ondragover event:

```
event.preventDefault()
```

Do the Drop - ondrop

When the dragged data is dropped, a drop event occurs.

In the example above, the ondrop attribute calls a function, drop(event):

```
function drop(ev)
{
    ev.preventDefault();
    var data=ev.dataTransfer.getData("Text");
    ev.target.appendChild(document.getElementById(data));
}
```

Code explained:

- Call preventDefault() to prevent the browser default handling of the data (default is open as link on drop)
- Get the dragged data with the dataTransfer.getData("Text") method. This method will return any data that was set to the same type in the setData() method
- The dragged data is the id of the dragged element ("drag1")
- Append the dragged element into the drop element

Real example: please visit

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_draganddrop2.

HTML5 Canvas

The <canvas> element is used to draw graphics, on the fly, on a web page.

What is Canvas?

The HTML5 <canvas> element is used to draw graphics, on the fly, via scripting (usually JavaScript).

The <canvas> element is only a container for graphics, you must use a script to actually draw the graphics.

A canvas is a drawable region defined in HTML code with height and width attributes.

Canvas has several methods for drawing paths, boxes, circles, characters, and adding images.

Browser Support



Internet Explorer 9, Firefox, Opera, Chrome, and Safari support the <canvas> element.

Note: Internet Explorer 8 and earlier versions, do not support the < canvas> element.

Create a Canvas

A canvas is specified with the <canvas> element.

Specify the id, width, and height of the <canvas> element:

```
< canvas id="myCanvas" width="200" height="100"></canvas>
```

Draw With JavaScript

The <canvas> element has no drawing abilities of its own.

All drawing must be done inside a JavaScript:

```
<script type="text/javascript">
  var c=document.getElementById("myCanvas");
  var ctx=c.getContext("2d");
  ctx.fillStyle="#FF0000";
  ctx.fillRect(0,0,150,75);
</script>
```

Code explained:

JavaScript uses the id to find the <canvas> element:

```
var c=document.getElementById("myCanvas");
```

Then, create a context object:

```
var ctx=c.getContext("2d");
```

The getContext("2d") object is a built-in HTML5 object, with many methods to draw paths, boxes, circles, characters, images and more.

The next two lines draws a red rectangle:

```
ctx.fillStyle="#FF0000";  
ctx.fillRect(0,0,150,75);
```

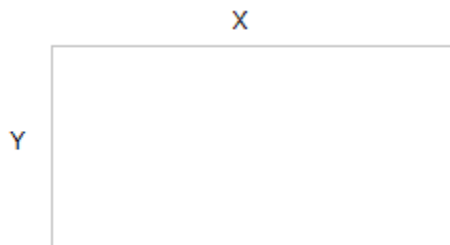
The fillStyle attribute makes it red, and the fillRect attribute specifies the shape, position, and size.

Understanding Coordinates

The fillRect property above had the parameters (0,0,150,75).

This means: Draw a 150x75 rectangle on the canvas, starting at the top left corner (0,0).

The canvas' X and Y coordinates are used to position drawings on the canvas.



Example: please visit:

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_canvas_line

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_canvas_circle

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_canvas_gradient

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_canvas_image

HTML5 Geolocation

HTML5 Geolocation is used to locate a user's position.

Locate the User's Position

The HTML5 Geolocation API is used to get the geographical position of a user.

Since this can compromise user privacy, the position is not available unless the user approves it.

Browser Support



Internet Explorer 9, Firefox, Chrome, Safari and Opera support Geolocation.

Note: Geolocation is much more accurate for devices with GPS, like iPhone.

HTML5 - Using Geolocation

Use the `getCurrentPosition()` method to get the user's position.

The example below is a simple Geolocation example returning the latitude and longitude of the user's position:

Example:

```
<script>
  var x=document.getElementById("demo");
  function getLocation()
  {
    if (navigator.geolocation)
    {
      navigator.geolocation.getCurrentPosition(showPosition);
    }
    else{x.innerHTML="Geolocation is not supported by this browser.";}
  }
  function showPosition(position)
  {
    x.innerHTML="Latitude: " + position.coords.latitude + 
      "<br />Longitude: " + position.coords.longitude;
  }
</script>
```

Example explained:

- Check if Geolocation is supported
- If supported, run the `getCurrentPosition()` method. If not, display a message to the user

- If the `getCurrentPosition()` method is successful, it returns a coordinates object to the function specified in the parameter (`showPosition`)
- The `showPosition()` function gets the displays the Latitude and Longitude

The example above is a very basic Geolocation script, with no error handling.

Handling Errors and Rejections

The second parameter of the `getCurrentPosition()` method is used to handle errors. It specifies a function to run if it fails to get the user's location:

Example

```
function showError(error)
{
    switch(error.code)
    {
        case error.PERMISSION_DENIED:
            x.innerHTML="User denied the request for Geolocation."
            break;
        case error.POSITION_UNAVAILABLE:
            x.innerHTML="Location information is unavailable."
            break;
        case error.TIMEOUT:
            x.innerHTML="The request to get user location timed out."
            break;
        case error.UNKNOWN_ERROR:
            x.innerHTML="An unknown error occurred."
            break;
    }
}
```

Error Codes:

- Permission denied - The user did not allow Geolocation
- Position unavailable - It is not possible to get the current location
- Timeout - The operation timed out

Displaying the Result in a Map

To display the result in a map, you need access to a map service that can use latitude and longitude, like Google Maps:

Example

```
function showPosition(position)
{
    var latlon=position.coords.latitude+","+position.coords.longitude;

    var img_url="http://maps.googleapis.com/maps/api/staticmap?center="+
        +latlon+"&zoom=14&size=400x300&sensor=false";
```

```
document.getElementById("mapholder").innerHTML="<img src='"+img_url+"' />";
}
```

In the example above we use the returned latitude and longitude data to show the location in a Google map (using a static image).

The `getCurrentPosition()` Method - Return Data

The `getCurrentPosition()` method returns an object if it is successful. The latitude, longitude and accuracy properties are always returned. The other properties below are returned if available.

Property	Description
<code>coords.latitude</code>	The latitude as a decimal number
<code>coords.longitude</code>	The longitude as a decimal number
<code>coords.accuracy</code>	The accuracy of position
<code>coords.altitude</code>	The altitude in meters above the mean sea level
<code>coords.altitudeAccuracy</code>	The altitude accuracy of position
<code>coords.heading</code>	The heading as degrees clockwise from North
<code>coords.speed</code>	The speed in meters per second
<code>timestamp</code>	The date/time of the response

Geolocation object - Other interesting Methods

`watchPosition()` - Returns the current position of the user and continues to return updated position as the user moves (like the GPS in a car).

`clearWatch()` - Stops the `watchPosition()` method.

The example below shows the `watchPosition()` method. You need an accurate GPS device to test this (like iPhone):

Example

```
<script>
var x=document.getElementById("demo");
function getLocation()
{
  if (navigator.geolocation)
  {
    navigator.geolocation.watchPosition(showPosition);
  }
  else{x.innerHTML="Geolocation is not supported by this browser.";}
}
function showPosition(position)
{
  x.innerHTML="Latitude: " + position.coords.latitude +
    "<br />Longitude: " + position.coords.longitude;
}
</script>
```

You can try example here:

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_geolocation_watchposition.

HTML5 Web Storage

HTML5 web storage, a better local storage than cookies.

What is HTML5 Web Storage?

With HTML5, web pages can store data locally within the user's browser.

Earlier, this was done with cookies. However, Web Storage is more secure and faster. The data is not included with every server request, but used ONLY when asked for. It is also possible to store large amounts of data, without affecting the website's performance.

The data is stored in key/value pairs, and a web page can only access data stored by itself.

Browser Support



Web storage is supported in Internet Explorer 8+, Firefox, Opera, Chrome, and Safari.

Note: Internet Explorer 7 and earlier versions, do not support web storage.

localStorage and sessionStorage

There are two new objects for storing data on the client:

- localStorage - stores data with no expiration date
- sessionStorage - stores data for one session

Before using web storage, check browser support for localStorage and sessionStorage:

```
if(typeof(Storage)!="undefined")
{
    // Yes! localStorage and sessionStorage support!
    // Some code.....
}
else
{
    // Sorry! No web storage support..
}
```

The localStorage Object

The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.

Example

```
localStorage.lastname="Smith";
document.getElementById("result").innerHTML="Last name: "
    + localStorage.lastname;
```

Example explained:

- Create a localStorage key/value pair with key="lastname" and value="Smith"
- Retrieve the value of the "lastname" key and insert it into the element with id="result"

Tip: Key/value pairs are always stored as strings. Remember to convert them to another format when needed.

The following example counts the number of times a user has clicked a button. In this code the value string is converted to a number to be able to increase the counter:

Example

```
if (localStorage.clickcount)
{
    localStorage.clickcount=Number(localStorage.clickcount)+1;
}
else
{
    localStorage.clickcount=1;
}
document.getElementById("result").innerHTML="You have clicked the button " +
    localStorage.clickcount + " time(s).";
```

The sessionStorage Object

The sessionStorage object is equal to the localStorage object, **except** that it stores the data for only one session. The data is deleted when the user closes the browser window.

The following example counts the number of times a user has clicked a button, in the current session:

Example

```
if (sessionStorage.clickcount)
{
    sessionStorage.clickcount=Number(sessionStorage.clickcount)+1;
}
else
{
    sessionStorage.clickcount=1;
}
document.getElementById("result").innerHTML="You have clicked the button " +
    sessionStorage.clickcount + " time(s) in this session.";
```

You can try examples here:

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_webstorage_local_clickcount

http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_webstorage_session_clickcount

HTML5 Application Cache

With HTML5 it is easy to make an offline version of a web application, by creating a cache manifest file.

What is Application Cache?

HTML5 introduces application cache, which means that a web application is cached, and accessible without an internet connection.

Application cache gives an application three advantages:

1. Offline browsing - users can use the application when they're offline
2. Speed - cached resources load faster
3. Reduced server load - the browser will only download updated/changed resources from the server

Browser Support



Application cache is supported in all major browsers, except Internet Explorer.

HTML5 Cache Manifest Example

The example below shows an HTML document with a cache manifest (for offline browsing):

Example

```
<!DOCTYPE HTML>
<html manifest="demo.appcache">

<body>
The content of the document.....
</body>

</html>
```

Cache Manifest Basics

To enable application cache, include the manifest attribute in the document's < html> tag:

```
< !DOCTYPE HTML>
<html manifest="demo.appcache">
...
</html>
```

Every page with the manifest attribute specified will be cached when the user visits it. If the manifest attribute is not specified, the page will not be cached (unless the page is specified directly in the manifest file).

The recommended file extension for manifest files is: ".appcache"

💡 A manifest file needs to be served with the **correct MIME-type**, which is "text/cache-manifest". Must be configured on the web server.

The Manifest File

The manifest file is a simple text file, which tells the browser what to cache (and what to never cache).

The manifest file has three sections:

- **CACHE MANIFEST** - Files listed under this header will be cached after they are downloaded for the first time
- **NETWORK** - Files listed under this header require a connection to the server, and will never be cached
- **FALLBACK** - Files listed under this header specifies fallback pages if a page is inaccessible

CACHE MANIFEST

The first line, CACHE MANIFEST, is required:

```
CACHE MANIFEST
/theme.css
/logo.gif
/main.js
```

The manifest file above lists three resources: a CSS file, a GIF image, and a JavaScript file. When the manifest file is loaded, the browser will download the three files from the root directory of the web site. Then, whenever the user is not connected to the internet, the resources will still be available.

NETWORK

The NETWORK section below specifies that the file "login.asp" should never be cached, and will not be available offline:

```
NETWORK:
login.asp
```

An asterisk can be used to indicate that all other resources/files require an internet connection:

```
NETWORK:
*
```


FALLBACK

The FALLBACK section below specifies that "offline.html" will be served in place of all files in the /html5/ catalog, in case an internet connection cannot be established:

```
FALLBACK:  
/html5/ /offline.html
```

Note: The first URI is the resource, the second is the fallback.


Updating the Cache

Once an application is cached, it remains cached until one of the following happens:

- The user clears the browser's cache
- The manifest file is modified (see tip below)
- The application cache is programmatically updated

Example - Complete Cache Manifest File

```
CACHE MANIFEST  
# 2012-02-21 v1.0.0  
/theme.css  
/logo.gif  
/main.js  
  
NETWORK:  
login.asp  
  
FALLBACK:  
/html5/ /offline.html
```

 **Tip:** Lines starting with a "#" are comment lines, but can also serve another purpose. An application's cache is only updated when its manifest file changes. If you edit an image or change a JavaScript function, those changes will not be re-cached. Updating the date and version in a comment line is one way to make the browser re-cache your files.

Notes on Application Cache

Be careful with what you cache.

Once a file is cached, the browser will continue to show the cached version, even if you change the file on the server. To ensure the browser updates the cache, you need to change the manifest file.

Note: Browsers may have different size limits for cached data (some browsers have a 5MB limit per site).

HTML5 Web Workers

A web worker is a JavaScript running in the background, without affecting the performance of the page.

What is a Web Worker?

When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.

A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page. You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background.

Browser Support



Web workers are supported in all major browsers, except Internet Explorer.

Check Web Worker Support

Before creating a web worker, check whether the user's browser supports it:

```
if(typeof(Worker)!="undefined")
{
    // Yes! Web worker support!
    // Some code.....
}
else
{
    // Sorry! No Web Worker support..
}
```

Create a Web Worker File

Now, let's create our web worker in an external JavaScript.

Here, we create a script that counts. The script is stored in the "demo_workers.js" file:

```
var i=0;

function timedCount()
{
    i=i+1;
    postMessage(i);
    setTimeout("timedCount()",500);
}

timedCount();
```

The important part of the code above is the **postMessage()** method - which is used to posts a message back to the HTML page.

Note: Normally web workers are not used for such simple scripts, but for more CPU intensive tasks.

Create a Web Worker Object

Now that we have the web worker file, we need to call it from an HTML page.

The following lines checks if the worker already exists, if not - it creates a new web worker object and runs the code in "demo_workers.js":

```
if(typeof(w)=="undefined")
{
    w=new Worker("demo_workers.js");
}
```

Then we can send and receive messages from the web worker.

Add an "onmessage" event listener to the web worker.

```
w.onmessage=function(event){
    document.getElementById("result").innerHTML=event.data;
};
```

When the web worker posts a message, the code within the event listener is executed. The data from the web worker is stored in event.data.

Terminate a Web Worker

When a web worker object is created, it will continue to listen for messages (even after the external script is finished) until it is terminated.

To terminate a web worker, and free browser/computer resources, use the terminate() method:

```
w.terminate();
```

Web Workers and the DOM

Since web workers are in external files, they do not have access to the following JavaScript objects:

- The window object
- The document object
- The parent object

Full example: please visit http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_webworker.

HTML5 Server-Sent Events

HTML5 Server-Sent Events allow a web page to get updates from a server.

Server-Sent Events - One Way Messaging

A server-sent event is when a web page automatically gets updates from a server.

This was also possible before, but the web page would have to ask if any updates were available. With server-sent events, the updates come automatically.

Examples: Facebook/Twitter updates, stock price updates, news feeds, sport results, etc.

Browser Support



Server-Sent Events are supported in all major browsers, except Internet Explorer.

Receive Server-Sent Event Notifications

The EventSource object is used to receive server-sent event notifications:

Example

```
var source=new EventSource("demo_sse.php");
source.onmessage=function(event)
{
    document.getElementById("result").innerHTML+=event.data + "<br />";
};
```

Example explained:

- Create a new EventSource object, and specify the URL of the page sending the updates (in this example "demo_sse.php")
- Each time an update is received, the onmessage event occurs
- When an onmessage event occurs, put the received data into the element with id="result"

Check Server-Sent Events Support

In the tryit example above there were some extra lines of code to check browser support for server-sent events:

```
if(typeof(EventSource)!="undefined")
{
    // Yes! Server-sent events support!
    // Some code.....
}
```

```
else
{
    // Sorry! No server-sent events support..
}
```

Server-Side Code Example

For the example above to work, you need a server capable of sending data updates (like PHP or ASP).

The server-side event stream syntax is simple. Set the "Content-Type" header to "text/event-stream". Now you can start sending event streams.

Code in PHP (demo_sse.php):

```
<?php
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');

$time = date('r');
echo "data: The server time is: {$time}\n\n";
flush();
?>
```

Code in ASP (VB) (demo_sse.asp):

```
<%
Response.ContentType="text/event-stream"
Response.Expires=-1
Response.Write("data: " & now())
Response.Flush()
%>
```

Code explained:

- Set the "Content-Type" header to "text/event-stream"
- Specify that the page should not cache
- Output the data to send (**Always** start with "data: ")
- Flush the output data back to the web page

The EventSource Object

In the examples above we used the onmessage event to get messages. But other events are also available:

Events	Description
onopen	When a connection to the server is opened
onmessage	When a message is received
onerror	When an error occurs

Example: please visit http://www.w3schools.com/html5/tryit.asp?filename=tryhtml5_sse.

HTML5 Input Types

HTML5 has several new input types for forms. These new features allow better input control and validation.

This chapter covers the new input types:

- color
- date
- datetime
- datetime-local
- email
- month
- number
- range
- search
- tel
- time
- url
- week

Note: Not all major browsers support all the new input types. However, you can already start using them; If they are not supported, they will behave as regular text fields.

Input Type: color

The color type is used for input fields that should contain a color.

Support browsers: 

Example

Select a color from a color picker:

Select your favorite color: `<input type="color" name="favcolor" />`

Input Type: date

The date type allows the user to select a date.

Support browsers: 

Example

Define a date control:

Birthday: `<input type="date" name="bday" />`

Input Type: datetime

The datetime type allows the user to select a date and time (with time zone).

Support browsers:     

Example

Define a date and time control (with time zone):

```
Birthday (date and time): <input type="datetime" name="bdaytime" />
```

Input Type: datetime-local

The datetime-local type allows the user to select a date and time (no time zone).

Support browsers:     

Example

Define a date and time control (no time zone):

```
Birthday (date and time): <input type="datetime-local" name="bdaytime" />
```

Input Type: email

The email type is used for input fields that should contain an e-mail address.

Support browsers:     

Example

Define a field for an e-mail address (will be automatically validated when submitted):

```
E-mail: <input type="email" name="usremail" />
```

Tip: Safari on iPhone recognizes the email type, and changes the on-screen keyboard to match it (adds @ and .com options).

Input Type: month

The month type allows the user to select a month and year.

Support browsers:     

Example

Define a month and year control (no time zone):

```
Birthday (month and year): <input type="month" name="bdaymonth" />
```

Input Type: number

The number type is used for input fields that should contain a numeric value.

You can also set restrictions on what numbers are accepted:

Support browsers: 

Example

Define a numeric field (with restrictions):

```
Quantity (between 1 and 5): <input type="number" name="quantity" min="1" max="5" />
```

Use the following attributes to specify restrictions:

- max - specifies the maximum value allowed
- min - specifies the minimum value allowed
- step - specifies the legal number intervals
- value - Specifies the default value

Input Type: range

The range type is used for input fields that should contain a value from a range of numbers.

You can also set restrictions on what numbers are accepted.

Support browsers: 

Example

Define a control for entering a number whose exact value is not important (like a slider control):


```
<input type="range" name="points" min="1" max="10" />
```

Use the following attributes to specify restrictions:

- max - specifies the maximum value allowed
- min - specifies the minimum value allowed
- step - specifies the legal number intervals
- value - Specifies the default value

Input Type: search

The search type is used for search fields (a search field behaves like a regular text field).


Support browsers: 

Example

Define a search field (like a site search, or Google search):

Search Google: `<input type="search" name="googlesearch" />`

Input Type: tel

Support browsers: not yet ()

Example

Define a field for entering a telephone number:

Telephone: `<input type="tel" name="usrtel" />`

Input Type: time

The time type allows the user to select a time.

Support browsers: 

Example

Define a control for entering a time (no time zone):

Select a time: `<input type="time" name="usr_time" />`

Input Type: url

The url type is used for input fields that should contain a URL address.

The value of the url field is automatically validated when the form is submitted.

Support browsers: 

Example

Define a field for entering a URL:

Add your homepage: `<input type="url" name="homepage" />`

Tip: Safari on iPhone recognizes the url input type, and changes the on-screen keyboard to match it (adds .com option).

Input Type: week

The week type allows the user to select a week and year.

Support browsers:     

Example

Define a week and year control (no time zone):

Select a week: `<input type="week" name="week_year" />`

HTML5 Form Elements

HTML5 has the following new form elements:

- `<datalist>`
- `<keygen>`
- `<output>`

Note: Not all major browsers support all the new form elements. However, you can already start using them; If they are not supported, they will behave as regular text fields.

HTML5 `<datalist>` Element

The `<datalist>` element specifies a list of pre-defined options for an `<input>` element.

The `<datalist>` element is used to provide an "autocomplete" feature on `<input>` elements. Users will see a drop-down list of pre-defined options as they input data.

Use the `<input>` element's `list` attribute to bind it together with a `<datalist>` element.

Support browsers: 

Example

An `<input>` element with pre-defined values in a `<datalist>`:

```
<input list="browsers" />

<datalist id="browsers">
  <option value="Internet Explorer">
  <option value="Firefox">
  <option value="Chrome">
  <option value="Opera">
  <option value="Safari">
</datalist>
```

HTML5 `<output>` Element

The `<output>` element represents the result of a calculation (like one performed by a script).

Support browsers: 

Example

Perform a calculation and show the result in an `<output>` element:

```
<form oninput="x.value=parseInt(a.value)+parseInt(b.value)">
  0<input type="range" name="a" value="50" />100
  +<input type="number" name="b" value="50" />
```

```
=<output name="x" for="a b"></output>  
</form>
```

HTML5 Form Attributes

HTML5 has several new attributes for `<form>` and `<input>`.

New attributes for `<form>`:

- `autocomplete`
- `novalidate`

New attributes for `<input>`:

- `autocomplete`
- `autofocus`
- `form`
- `formaction`
- `formenctype`
- `formmethod`
- `formnovalidate`
- `formtarget`
- `height` and `width`
- `list`
- `min` and `max`
- `multiple`
- `pattern` (regex)
- `placeholder`
- `required`
- `step`

`<form>` / `<input>` `autocomplete` Attribute

The `autocomplete` attribute specifies whether a form or input field should have `autocomplete` on or off.

When `autocomplete` is on, the browser automatically complete values based on values that the user has entered before.

Tip: It is possible to have `autocomplete` "on" for the form, and "off" for specific input fields, or vice versa.

Note: The `autocomplete` attribute works with `<form>` and the following `<input>` types: `text`, `search`, `url`, `tel`, `email`, `password`, `datepickers`, `range`, and `color`.

Support browsers: 

Example

An HTML form with `autocomplete` on (and off for one input field):

```
<form action="demo_form.asp" autocomplete="on">
  First name:<input type="text" name="fname" /><br />
  Last name: <input type="text" name="lname" /><br />
```

```
E-mail: <input type="email" name="email" autocomplete="off" /><br />
<input type="submit" />
</form>
```

Tip: In some browsers you may need to activate the autocomplete function for this to work.

<form> novalidate Attribute

The novalidate attribute is a boolean attribute.

When present, it specifies that the form-data (input) should not be validated when submitted.

Support browsers:     

Example

Indicates that the form is not to be validated on submit:

```
<form action="demo_form.asp" novalidate="novalidate">
  E-mail: <input type="email" name="user_email" />
  <input type="submit" />
</form>
```

<input> autofocus Attribute

The autofocus attribute is a boolean attribute.

When present, it specifies that an <input> element should automatically get focus when the page loads.

Support browsers:     

Example

Let the "First name" input field automatically get focus when the page loads:

```
First name:<input type="text" name="fname" autofocus="autofocus" />
```

<input> form Attribute

The form attribute specifies one or more forms an <input> element belongs to.

Tip: To refer to more than one form, use a space-separated list of form ids.

Support browsers:     

Example

An input field located outside the HTML form (but still a part of the form):

```
<form action="demo_form.asp" id="form1">
  First name: <input type="text" name="fname" /><br />
  <input type="submit" value="Submit" />
</form>
```

```
Last name: <input type="text" name="lname" form="form1" />
```

<input> formaction Attribute

The formaction attribute specifies the URL of a file that will process the input control when the form is submitted.

The formaction attribute overrides the action attribute of the <form> element.

Note: The formaction attribute is used with type="submit" and type="image".

Support browsers: 

Example

An HTML form with two submit buttons, with different actions:

```
<form action="demo_form.asp">
  First name: <input type="text" name="fname" /><br />
  Last name: <input type="text" name="lname" /><br />
  <input type="submit" value="Submit" /><br />
  <input type="submit" formaction="demo_admin.asp" value="Submit as admin" />
</form>
```

<input> formenctype Attribute

The formenctype attribute specifies how the form-data should be encoded when submitting it to the server (only for forms with method="post")

The formenctype attribute overrides the enctype attribute of the < form> element.

Note: The formenctype attribute is used with type="submit" and type="image".

Support browsers: 

Example

Send form-data that is default encoded (the first submit button), and encoded as "multipart/form-data" (the second submit button):

```
<form action="demo_post_enctype.asp" method="post">
  First name: <input type="text" name="fname" /><br />
  <input type="submit" value="Submit" />
  <input type="submit" formenctype="multipart/form-data" value="Submit as
Multipart/form-data" />
</form>
```

<input> formmethod Attribute

The formmethod attribute defines the HTTP method for sending form-data to the action URL.

The formmethod attribute overrides the method attribute of the < form> element.

Note: The formmethod attribute can be used with type="submit" and type="image".

Support browsers: 

Example

The second submit button overrides the HTTP method of the form:

```
<form action="demo_form.asp" method="get">
  First name: <input type="text" name="fname" /><br />
  Last name: <input type="text" name="lname" /><br />
  <input type="submit" value="Submit" />
  <input type="submit" formmethod="post" formaction="demo_post.asp"
value="Submit using POST" />
</form>
```

<input> formnovalidate Attribute

The novalidate attribute is a boolean attribute.

When present, it specifies that the <input> element should not be validated when submitted.

The formnovalidate attribute overrides the novalidate attribute of the < form> element.

Note: The formnovalidate attribute can be used with type="submit".

Support browsers: 

Example

A form with two submit buttons (with and without validation):

```
<form action="demo_form.asp">
  E-mail: <input type="email" name="userid" /><br />
  <input type="submit" value="Submit" /><br />
  <input type="submit" formnovalidate="formnovalidate" value="Submit without
validation" />
</form>
```

<input> formtarget Attribute

The formtarget attribute specifies a name or a keyword that indicates where to display the response that is received after submitting the form.

The formtarget attribute overrides the target attribute of the < form> element.

Note: The formtarget attribute can be used with type="submit" and type="image".

Support browsers: 

Example

A form with two submit buttons, with different target windows:

```
<form action="demo_form.asp">
  First name: <input type="text" name="fname" /><br />
  Last name: <input type="text" name="lname" /><br />
  <input type="submit" value="Submit as normal" />
  <input type="submit" formtarget="_blank" value="Submit to a new window" />
</form>
```

<input> height and width Attributes

The height and width attributes specify the height and width of an <input> element.

Note: The height and width attributes are only used with <input type="image">.

Tip: Always specify both the height and width attributes for images. If height and width are set, the space required for the image is reserved when the page is loaded. However, without these attributes, the browser does not know the size of the image, and cannot reserve the appropriate space to it. The effect will be that the page layout will change during loading (while the images load).

Support browsers: 

Example

Define an image as the submit button, with height and width attributes:

```
<input type="image" src="img_submit.gif" alt="Submit" width="48"
height="48"/>
```

<input> multiple Attribute

The multiple attribute is a boolean attribute.

When present, it specifies that the user is allowed to enter more than one value in the <input> element.

Note: The multiple attribute works with the following input types: email, and file.

Support browsers: 

Example

A file upload field that accepts multiple values:

```
Select images: <input type="file" name="img" multiple="multiple" />
```

<input> pattern Attribute

The pattern attribute specifies a regular expression that the <input> element's value is checked against.

Note: The pattern attribute works with the following input types: text, search, url, tel, email, and password.

Support browsers:     

Example

An input field that can contain only three letters (no numbers or special characters):

```
Country code: <input type="text" name="country_code" pattern="[A-Za-z]{3}"
title="Three letter country code" />
```

<input> placeholder Attribute

The placeholder attribute specifies a short hint that describes the expected value of an input field (e.g. a sample value or a short description of the expected format).

The hint is displayed in the input field when it is empty, and disappears when the field gets focus.

Note: The placeholder attribute works with the following input types: text, search, url, tel, email, and password.

Support browsers:     

Example

An input field with a placeholder text:

```
<input type="text" name="fname" placeholder="First name" />
```

<input> required Attribute

The required attribute is a boolean attribute.

When present, it specifies that an input field must be filled out before submitting the form.

Note: The required attribute works with the following input types: text, search, url, tel, email, password, date pickers, number, checkbox, radio, and file.

Support browsers:     

Example

A required input field:

Username:

Chapter 2: PhoneGap

Easily create apps with the only free open source framework that supports 7 mobile platforms.

Developing with **Adobe® PhoneGap™** gives you the freedom to create mobile applications for iOS, Android, Blackberry, Windows Phone, Palm WebOS, Bada and Symbian using the web code you know and love: **HTML, CSS and Javascript**



Overview

PhoneGap is an HTML5 app platform that allows you to author native applications with web technologies and get access to APIs and app stores:

- Build your app once with web-standards
-

Based on HTML5, PhoneGap leverages web technologies developers already know best.

- Wrap it with PhoneGap
-

Using the free open source framework or PhoneGap build you can get access to native APIs.

- Deploy to multiple platforms!
-

PhoneGap uses standards-based web technologies to bridge web applications and mobile devices.

With PhoneGap:

- Take advantage of HTML5 and CSS3
- Use JavaScript to write your code
- Access Native Features
- Deploy your app to Multiple Platforms
- Take advantage of PhoneGap Build
- Add PhoneGap plugins to your project
- Use Tools from the community
- Get help from the growing community

Supported Platforms

- iOS
- Android
- OS x
- Windows Phone
- webOS
- Symbian
- Bada

Supported Features

HTML5 offers new elements for better structure:

	iPhone 3GS+	Android	OS 5.x	OS 6.0+	Web OS	WP7	Symbian	Bada
ACCELEROMETER	✓	✓	✓	✓	✓	✓	✓	✓
CAMERA	✓	✓	✓	✓	✓	✓	✓	✓
COMPASS	✓	✓				✓		✓
CONTACTS	✓	✓	✓	✓		✓	✓	✓
FILE	✓	✓	✓	✓		✓		
GEOLOCATION	✓	✓	✓	✓	✓	✓	✓	✓
MEDIA	✓	✓				✓		
NETWORK	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (ALERT)	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (SOUND)	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (VIBRATION)	✓	✓	✓	✓	✓	✓	✓	✓
STORAGE	✓	✓	✓	✓	✓	✓	✓	✓

Getting Started with iOS

Requirements

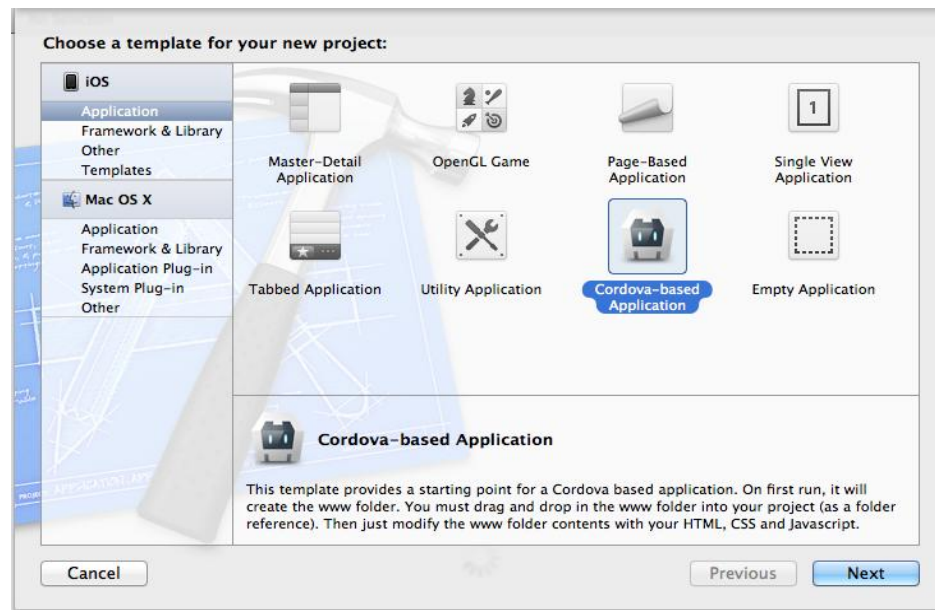
- XCode 4.x
- Intel-based computer with Mac OS X Lion (10.7)
- Necessary for installing on device:
 - Apple iOS device (iPhone, iPad, iPod Touch)
 - iOS developer certificate

Install iOS SDK and Apache Cordova

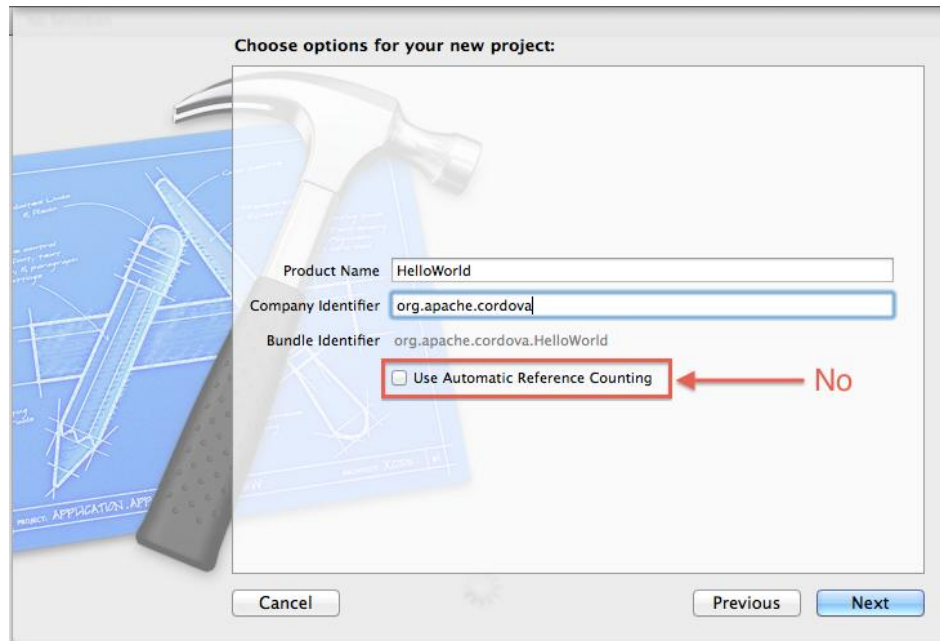
- Install XCode from the Mac App Store
- Download the latest release of Apache Cordova:
 - Extract its contents
 - Apache Cordova iOS is found under lib/ios

Setup New Project

- Launch XCode
- Select the File menu
- Select New -> New Project...
- Select Cordova-based Application from the list of templates



- Select the *Next* button
- Fill in the *Product Name* and *Company Identifier* for your app

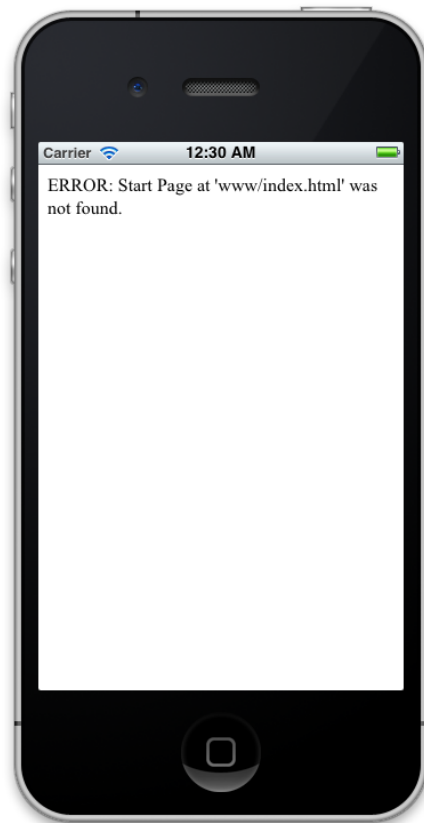


Note: Do not check *Use Automatic Reference Counting*.

- Select the *Next* button
- Choose a folder to save your new app
- Select the *Create* button

We've now created an Apache Cordova project. Next, we need to associate the project with a web directory. We need to do this step because of a limitation in Xcode project templates.

- Select the *Run* button in the top left corner:



Your build should succeed and launch in the iOS Simulator.

You should see an error in the iOS Simulator informing you that `www/index.html` was not found.

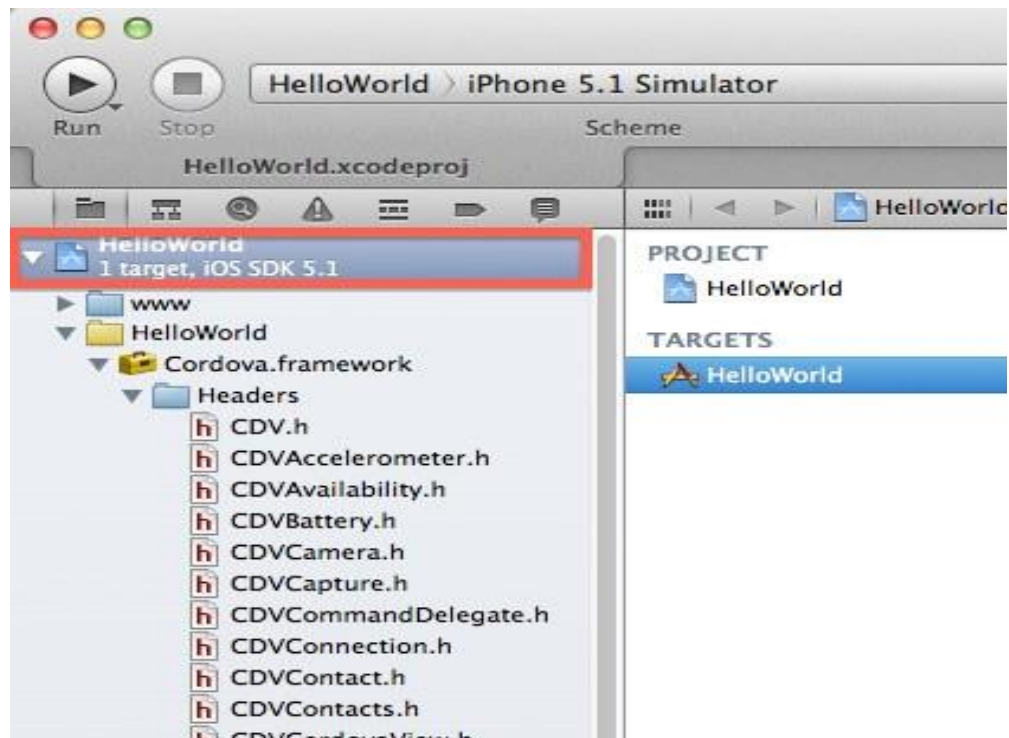
We can fix this by adding a folder to the project that references `www`.

- Right-click on the project icon in the *Project Navigator* (left sidebar) and select *Show in Finder*
- Using Finder, you should see a `www` directory inside your project

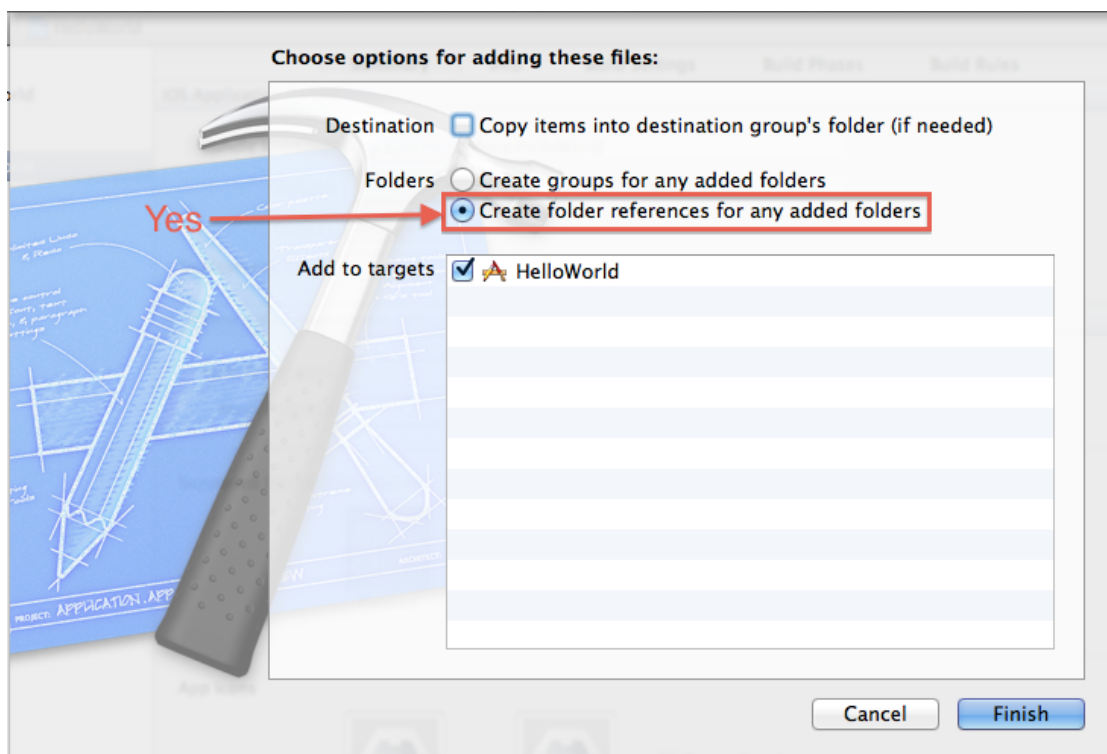


- Drag the `www` directory into XCode

A common mistake is to drag the `www` directory into your app's directory inside of Finder. Please follow the red highlighted section of the image below:



- After dragging `www` into Xcode, you will be prompted with a few options:



- Select *Create folder references for any added folders*
- Select the *Finish* button

Hello World

- Select the folder named *www* in the XCode *Project Navigator*
- Select the file *index.html*
- Add the following after `<body>`: `<h1></h1>`

You can also add any associated JavaScript and CSS files there as well.

Deploy to Simulator

- Change the *Active SDK* in the Scheme drop-down menu on the toolbar to iOS version *Simulator*
- Select the *Run* button in your project window's toolbar

Deploy to Device

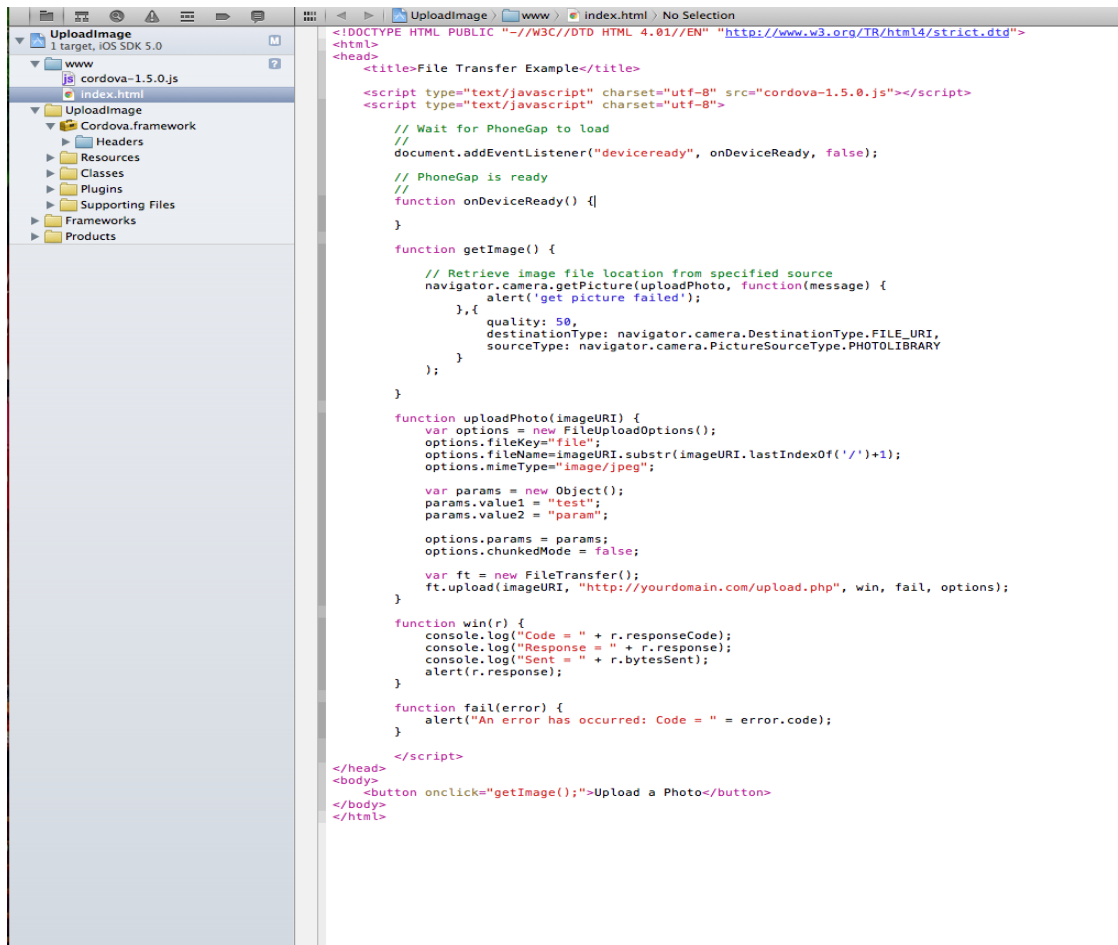
- Open *YourAppName-Info.plist*, under the *Supporting Files* group
- Change *BundleIdentifier* to the identifier provided by Apple or your own bundle identifier
- If you have a developer license, you can run the Assistant to register your app
- Change the *Active SDK* in the Scheme drop-down menu on the toolbar to *YourDeviceName*.

You will need to have your device connected via USB.

- Select the *Run* button in your project window's toolbar



Coding Style



API Documentation

Please visit link to access all API documents: <http://docs.phonegap.com/en/1.9.0/>.

Debugging PhoneGap apps

Debugging support for PhoneGap apps is not yet as well-developed as for native apps or for pure web apps. The big lack is that when working on your mobile device or simulator, you won't have breakpoints or stack traces. Nonetheless, there's still a lot you can do.

The main debugging techniques are:

- **Debug first on your desktop.** Since you're working with javascript/html/css - you can debug most of your app on your desktop with your favorite web environment (Firebug, etc.). Depending on your app, you may want to include code that mocks up (ie.: provides stubs for) the PhoneGap API, so your app doesn't croak on the desktop.
- **Use a Remote Web Inspector.** Not quite a full-fledged debugger, web inspectors give you a live view of your code while running on your mobile device and a limited ability to interact with it. No breakpoints or stack traces but you can edit css and basically see everything except the script tab in your favorite browser-based debugger.
- **console.log() and alerts Once deviceReady() has fired,** console output becomes available. For iOS, you can see it in the Debug Area in Xcode. For Android the tool LogCat

provides the console.log() printouts. LogCat is integrated to the Android Eclipse plugin so logs from device and emulator can be seen in the UI of Eclipse.

UI Development on PhoneGap

Choosing your UI framework precisely is serious stuff. Here's a little comparison to help you:

Framework	Sencha Touch	jQuery Mobile	jqTouch	iUI	Dojo mobile	qooxdoo Mobile
Approach	Full javascript	HTML progressive enhancement	HTML progressive enhancement	HTML progressive enhancement	Javascript	Javascript
Licence type	Multi-licensed	GPL / MIT	MIT	MIT	BSD	GNU Lesser General Public License (LGPL) or Eclipse Public License (EPL)
Compatibility	Webkit only	All devices	Webkit only	iOS, Android, Rim OS6, MeeGo, Bada, WebOS	Webkit	Desktop Browser: Webkit + Firefox Mobile Browsers: Android Native, iOS Safari, Webkit-Based
Pros	Pure Javascript Lots of widgets and utils Homogenous API and docs Professional grade	Easy learning curve SEO friendly Open source Easy theming Huge community, plugins	Easy learning curve SEO friendly Open source Easy theming Easy Animations	Lightweight		Object-oriented JavaScript Mobile-/Tablet-Support Open Source Nice and easy theming with LESS Great documentation and tooling (Test runner, Playground, Build Generator...)
Cons	Steep learning curve Not open source					

	Slow on old devices					
Typical usage	Pure javascript apps	Touch optimised websites, content oriented apps	Touch optimised websites			Native- or custom-themed apps for iOS and Android Devices (Mobile Phones AND Tablets).
Example apps	http://www.sencha.com/apps/	http://www.jqmgallery.com/		http://www.jui-js.org/gallery	http://demos.dojotoolkit.org/demos/demos.php?cat=mobile	http://qooxdoo.org/demos#mobile
API, docs	http://dev.sencha.com/deploy/touch/docs/	http://jquerymobile.com/demos	https://github.com/senchalabs/jQTouch/wiki		http://dojotoolkit.org/api/	http://manual.qooxdoo.org/devel/pages/mobile.html
Web site	http://sencha.com/products/touch	http://jquerymobile.com	http://jqtouch.com	http://www.jui-js.org/	http://dojotoolkit.org/	http://www.qooxdoo.org
Github	private	https://github.com/jquery/jquery-mobile	https://github.com/senchalabs/jQTouch	http://code.google.com/p/jui/	http://svn.dojotoolkit.org/	https://github.com/qooxdoo/qooxdoo

Chapter 3: Sencha Touch



Sencha Touch 2 Build Mobile Web Apps with HTML5



The Best Framework Just Got Better

Build HTML5 mobile apps for iPhone, Android, and BlackBerry.

With over 50 built-in components, state management, and a built-in MVC system, Sencha Touch 2 provides everything you need to create immersive mobile apps.



[Watch Video](#)

Overview

Sencha Touch 2, a high-performance HTML5 mobile application framework, is the cornerstone of the Sencha HTML5 platform. Built for enabling world-class user experiences, Sencha Touch 2 is the only framework that enables developers to build fast and impressive apps that work on iOS, Android, BlackBerry, Kindle Fire, and more.

Sencha Touch 2 includes an updated and easier to use API, enhanced MVC, and richer documentation. To harness local hardware and system services, Sencha Touch 2 provides access to a wider set of native device APIs, allowing HTML5 developers to take advantage of hardware features. To broaden the reach of apps created with Sencha Touch 2, a free native packager is now included enabling app distribution to the Apple App Store and the Android Market.

Smoother Scrolling and Animations

Sencha Touch 2 provides a user experience unparalleled in HTML5. Fluid animations and smooth scrolling make Sencha Touch 2 apps come alive, rivaling native technology. Lists, carousels, and other components scroll smoothly and naturally, with a high frame rate across a wide range of devices. The framework automatically uses the best scrolling mechanism for each device, resulting in a great experience everywhere.

Adaptive Layouts

Sencha Touch's novel layout engine leverages HTML5 in powerful ways to let developers build complex applications that respond, load, and layout in a snap. Switching from landscape to portrait happens nearly instantaneously, and applications load in fractions of a second as Sencha Touch 2's advanced layout engine ensures pixel perfection.

Native Packaging

Web applications work everywhere. But there are still a few features uniquely available to native apps — like camera access and app store distribution — that are essential to app developers. Sencha SDK Tools give you the best of both worlds, providing a way to seamlessly “wrap” your web app in a native shell. Whether you're on Mac or Windows, you're one command away from deploying to the Apple App Store or Android Market.

Sencha Touch 2 Device Support

Apple



Apple iPad iOS 4+



Apple iPhone 4S, 4 & 3GS iOS 4+



Apple iPod touch iOS 4+

BlackBerry



BlackBerry PlayBook BB Tablet OS 1



BlackBerry 9800 BB OS 6+

Android 2.3+ including these devices...



Amazon Kindle Fire Kindle OS 6.2.2



Asus Transformer Prime Android 4 w/ Chrome



Motorola Xoom 1, 2 Android 4.0.3+



Motorola Droid Bionic Android 2.3+



Motorola Droid X Android 2.3+



Motorola Droid 2, 3, 4, & 5 Android 2.2+



Motorola Atrix Android 2.3+



Samsung Galaxy S1, & S2 Android 2.3+



Samsung Galaxy Nexus Android 4 w/ Chrome



HTC Desire Android 2.2+



HTC Desire S Android 2.3+



HTC Evo 4G Android 2.3+

Screenshots



Buttons

[Back](#) **User Interface**

Buttons [Source](#)

Buttons

Forms

List

Nested List

Icons


Toolbars

Carousel

Tabs

Bottom Tabs

Overlays



Normal **Round** Small

Decline **Round** Small

Confirm **Round** Small **Back**

Forms

[Back](#) **User Interface**

Forms [Source](#)

Buttons

Forms

List

Nested List

Icons


Toolbars

Carousel

Tabs

Bottom Tabs

Overlays



Basic Sliders Toolbars

Personal Info

Name*	Tom Roy
Password	
Email	me@sencha.com
Url	http://sencha.com
Spinner	0 - +
Cool	<input checked="" type="checkbox"/>
Start Date	07/10/2012 ▼
Rank	Master ▼
Bio	

Please enter the information above.

Favorite color

Red	<input type="radio"/>
Blue	<input type="radio"/>
Green	<input type="radio"/>
Purple	<input type="radio"/>

Disable fields

Reset

List

[Back](#) **User Interface**

Buttons

Forms

List

Nested List

Icons

Toolbars

Carousel

Tabs

Bottom Tabs

Overlays

List [Source](#)

Simple

Grouped

Disclosure

A

Alana Wiersma

Allan Disbrow

Allan Leyendecker

Althea Sturtz

Ashlee Kennerly

Avis Bueche

C

Carlene Summitt

Clayton Clear

Cody Herrell

Codv Savler

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X







Y

Z

Nested list

Back User Interface	Nested List Source
Buttons	Back Cars
Forms	Asia
List	United Kingdom
Nested List	Europe
Icons	United States
Toolbars	
Carousel	
Tabs	
Bottom Tabs	
Overlays	

Icons

Back User Interface	Icons Source
Buttons	     
Forms	
List	
Nested List	
Icons	<p>Both toolbars and tabbars have built-in, ready to use icons.</p> <p>Sencha Touch comes with over 300 icons that can optionally be included in your app via Sass and Compass.</p>
Toolbars	
Carousel	
Tabs	
Bottom Tabs	
Overlays	

Toolbars

Back

User Interface

Buttons

Forms

List

Nested List

Icons

Toolbars

Carousel

Tabs

Bottom Tabs

Overlays

Source

BackDefault2RoundOption 1Option 2Option 3ActionForward

Toolbars automatically come with **light** and **dark** UIs, but you can also make your own with Sass.

Carousel

Back

User Interface

Buttons

Forms

List

Nested List

Icons

Toolbars

Carousel

Tabs

Bottom Tabs

Overlays

Carousel

Source

You can also tap on either side of the indicators.

● ● ●

Carousels can also be vertical (swipe up)...

● ● ●

Tabs

Back

User Interface

Buttons

Forms

List

Nested List

Icons


Toolbars

Carousel

Tabs

Bottom Tabs

Overlays



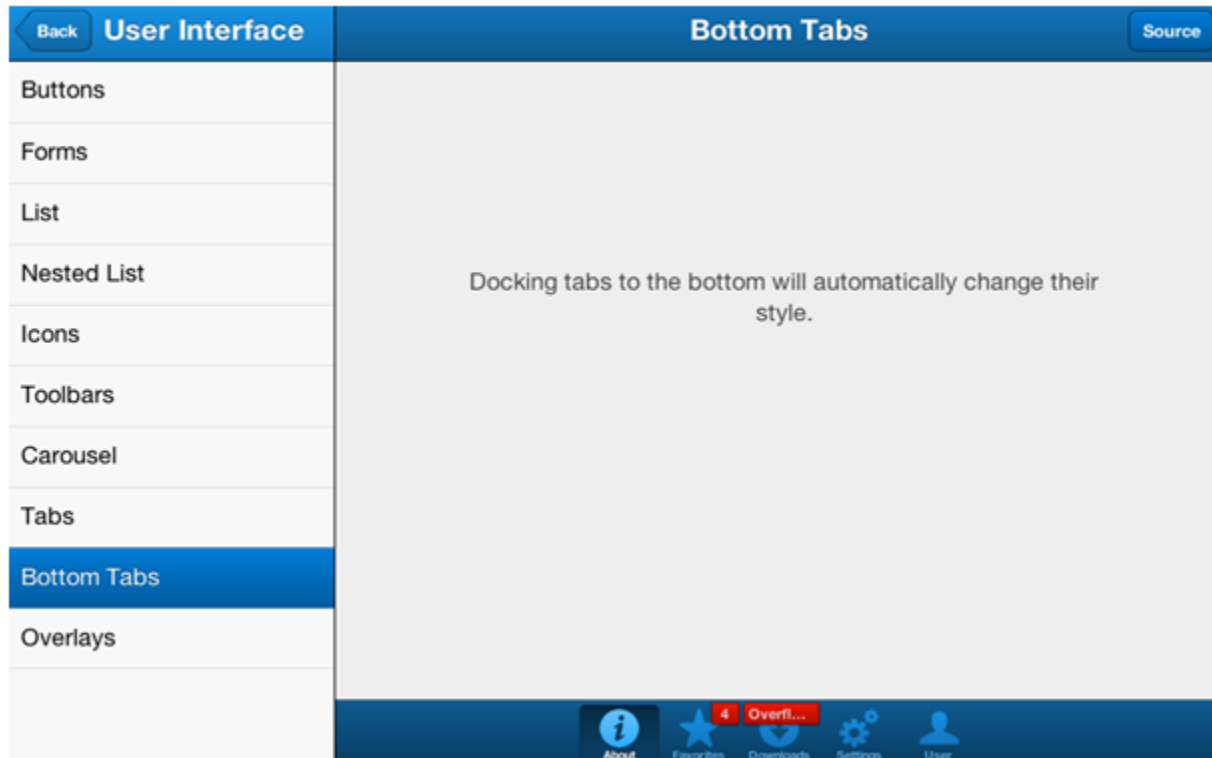
Tabs

Source

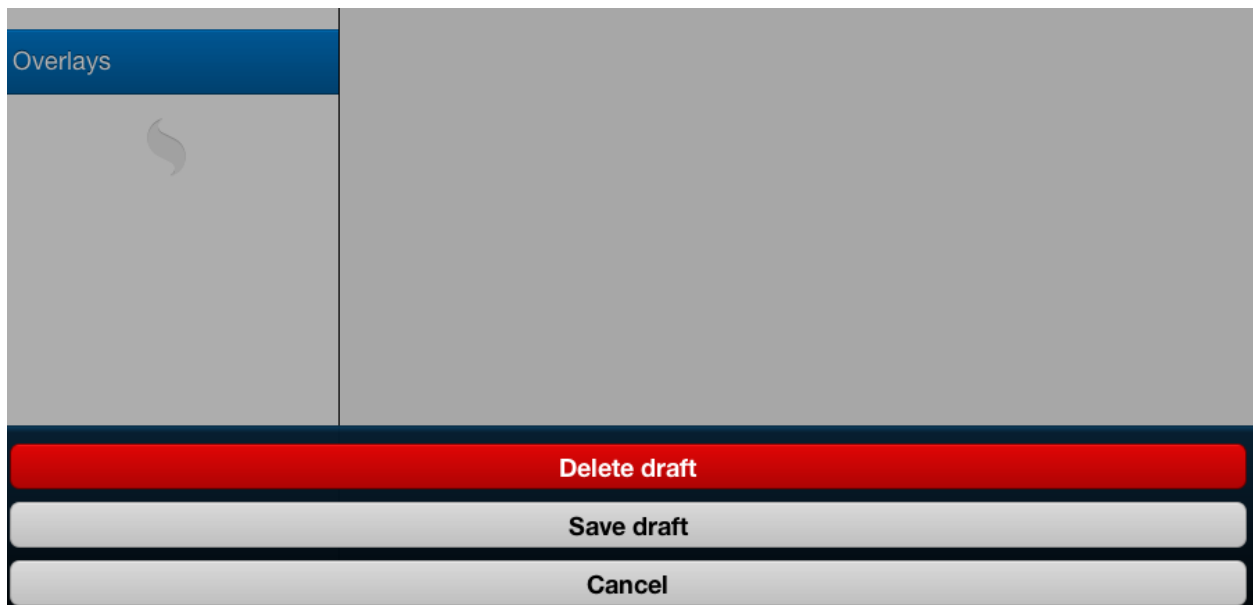
Tab 1Tab 2Tab 3

A TabPanel can use different animations by setting `layout.animation.`

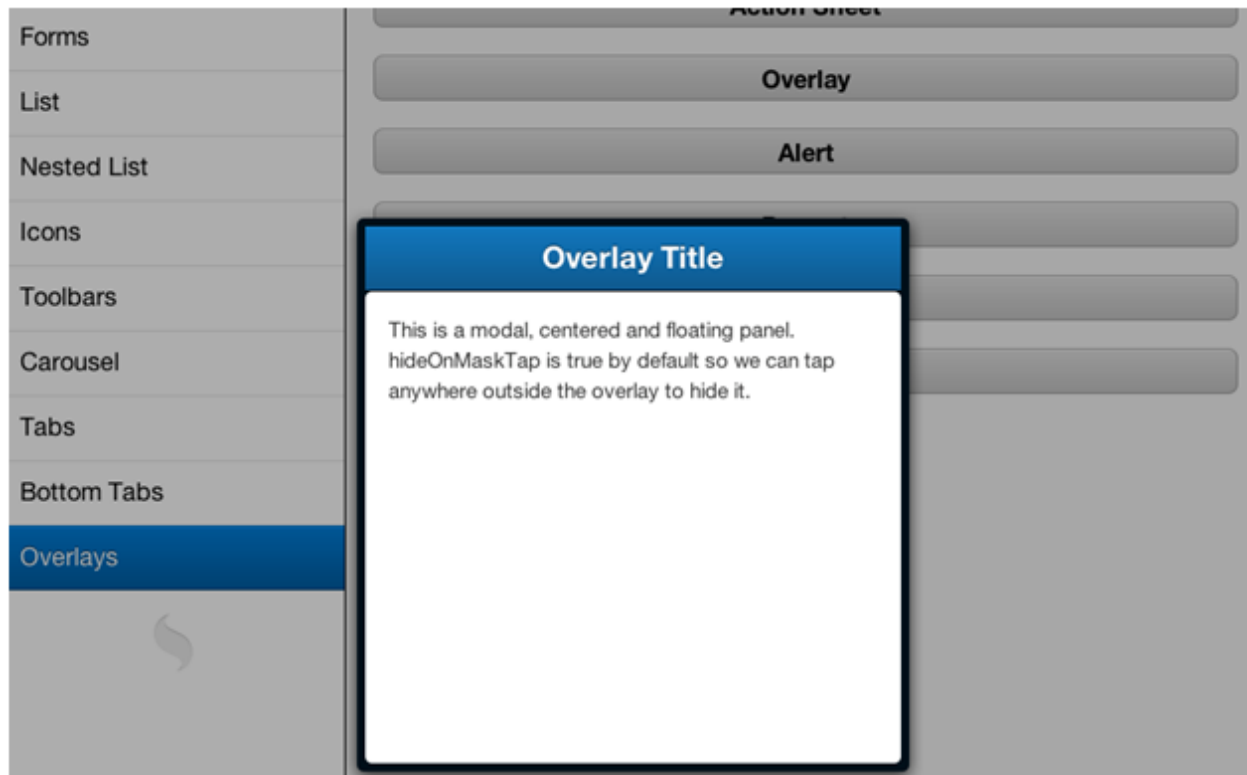
Bottom Tabs



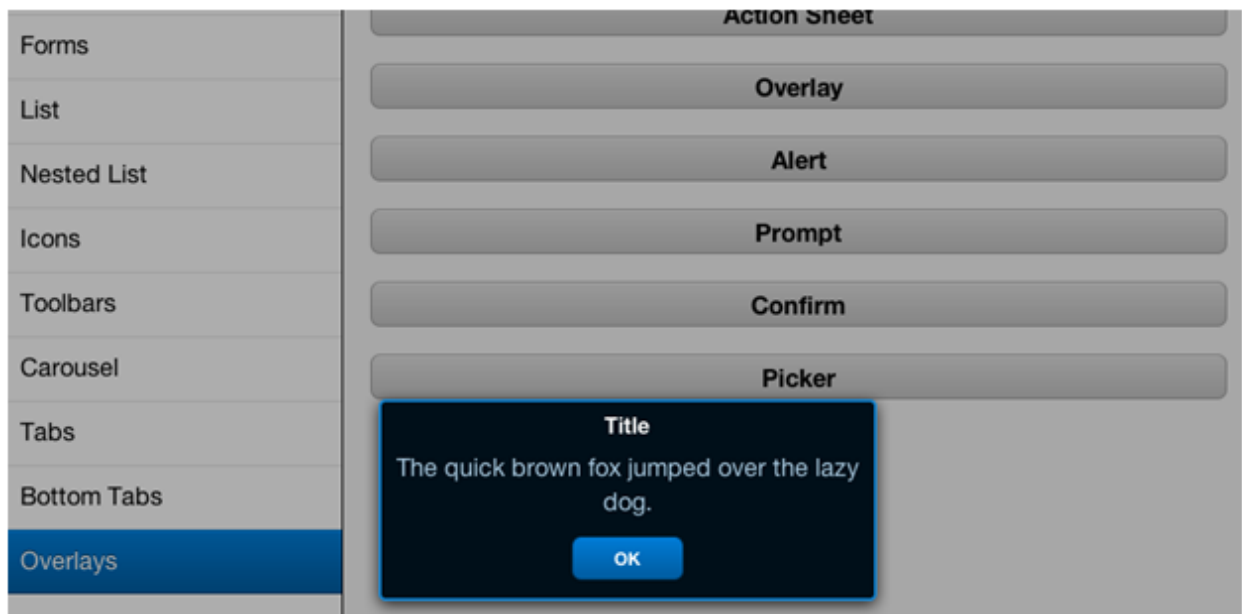
Action sheet



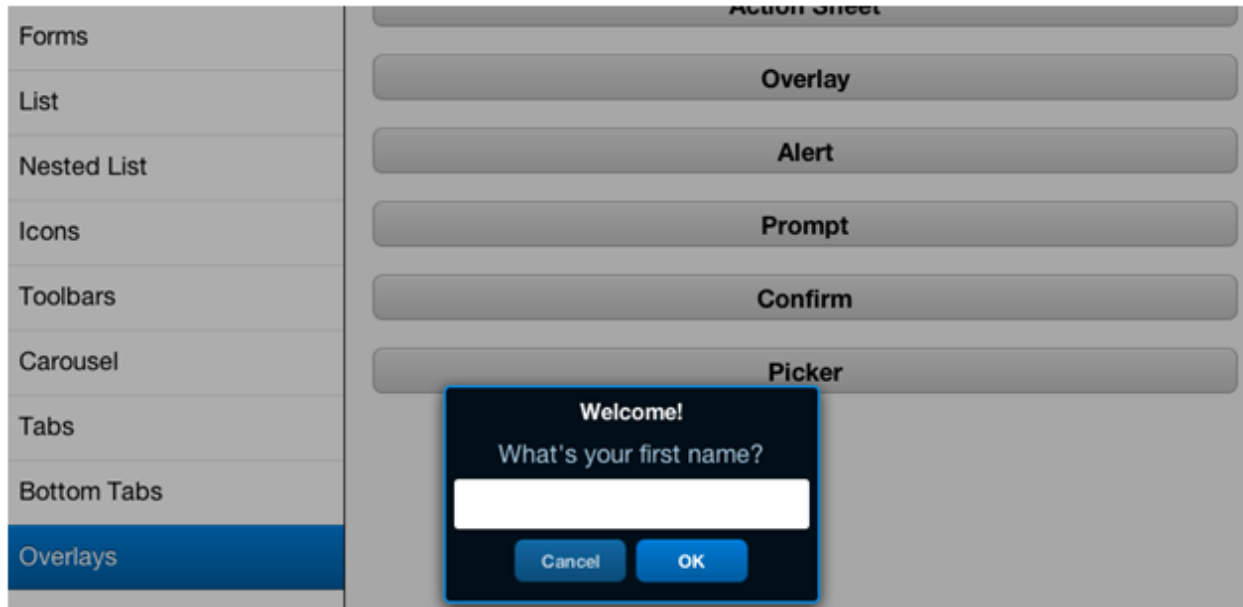
Popup



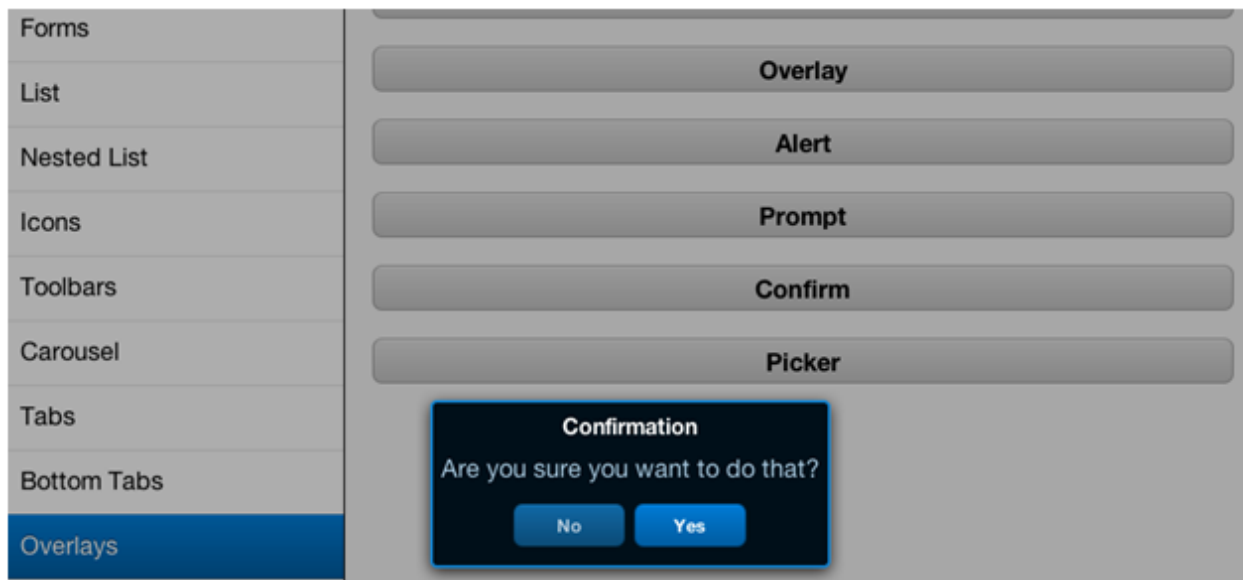
Alert



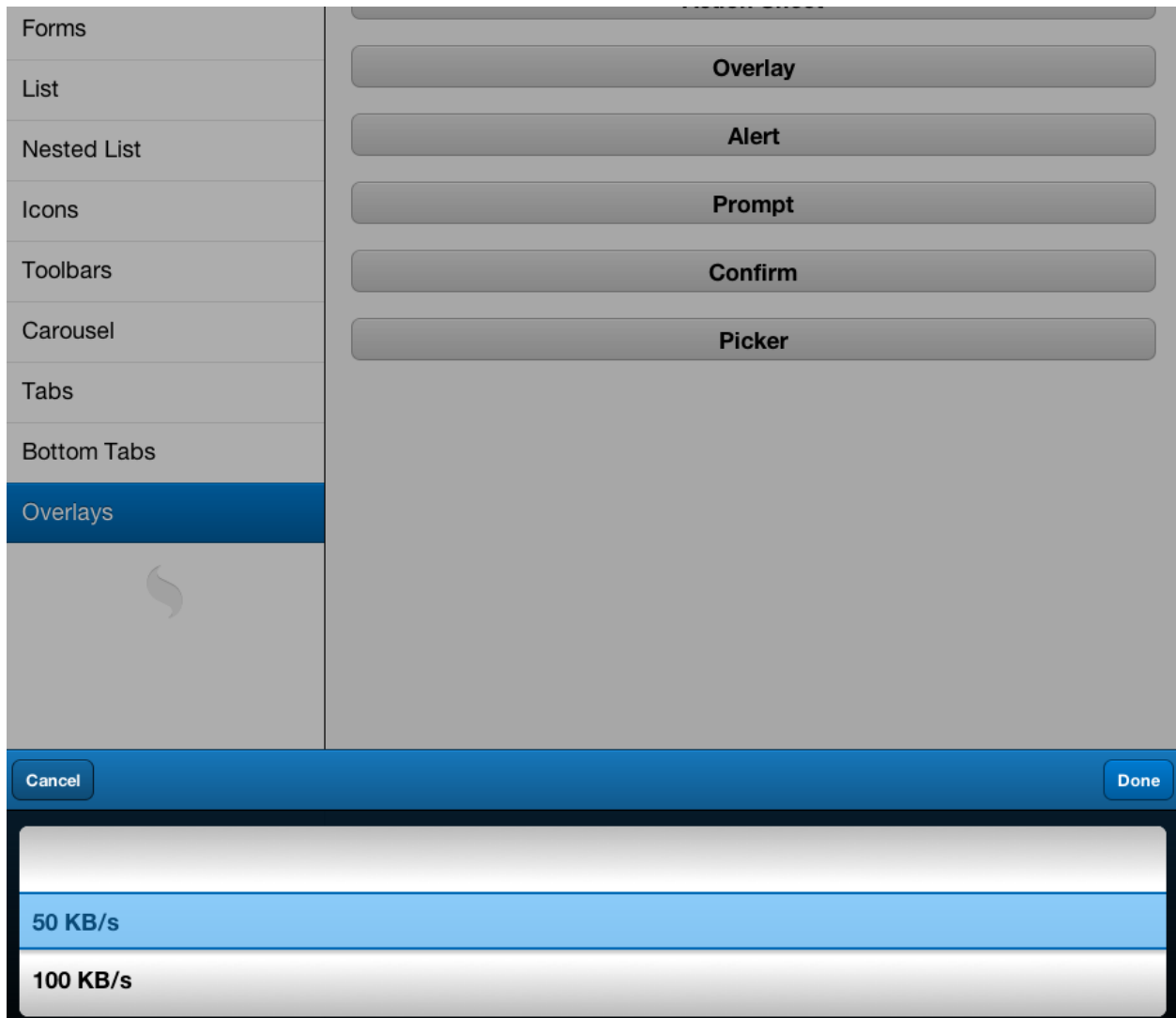
Prompt



Confirm



Picker



Getting started with Sencha Touch 2

Installation

First, you'll need to download the free Sencha Touch 2 SDK and SDK Tools from the Sencha website. You'll also need:

- A web server running locally on your computer
- A modern web browser; Chrome and Safari are recommended

If you are running IIS on Windows, please note that you must add application/x-json as a MIME Type for Sencha Touch to work properly.

Second, extract your SDK zip file to your projects directory. Ideally, this folder will be accessible by your HTTP server. For example, you should be able to navigate to *http://localhost/sencha-touch-2.0.0-gpl* from your web browser and see the Sencha Touch documentation.

You'll also need to run the SDK Tools installer. The SDK Tools will add the sencha command line tool to your path so that you can generate a fresh application template among other things. To check you have installed the SDK tools, change to your Sencha Touch directory and type the sencha command. For example:

```
$ cd ~/webroot/sencha-touch-2.0.0-gpl/  
$ sencha  
Sencha Command v2.0.0 for Sencha Touch 2  
Copyright (c) 2012 Sencha Inc.  
...
```

NOTE: You must be inside either the downloaded SDK directory or a generated Touch app when using the sencha command.

Your first application

Now you have Sencha Touch and the SDK Tools installed, lets generate an application. Make sure you're still in the Sencha Touch SDK folder, and type the following:

```
$ sencha generate app GS ../GS  
[INFO] Created file /Users/nickpoulden/Projects/sencha/GS/.senchasdk  
[INFO] Created file /Users/nickpoulden/Projects/sencha/GS/index.html  
[INFO] Created file /Users/nickpoulden/Projects/sencha/GS/app.js  
...
```

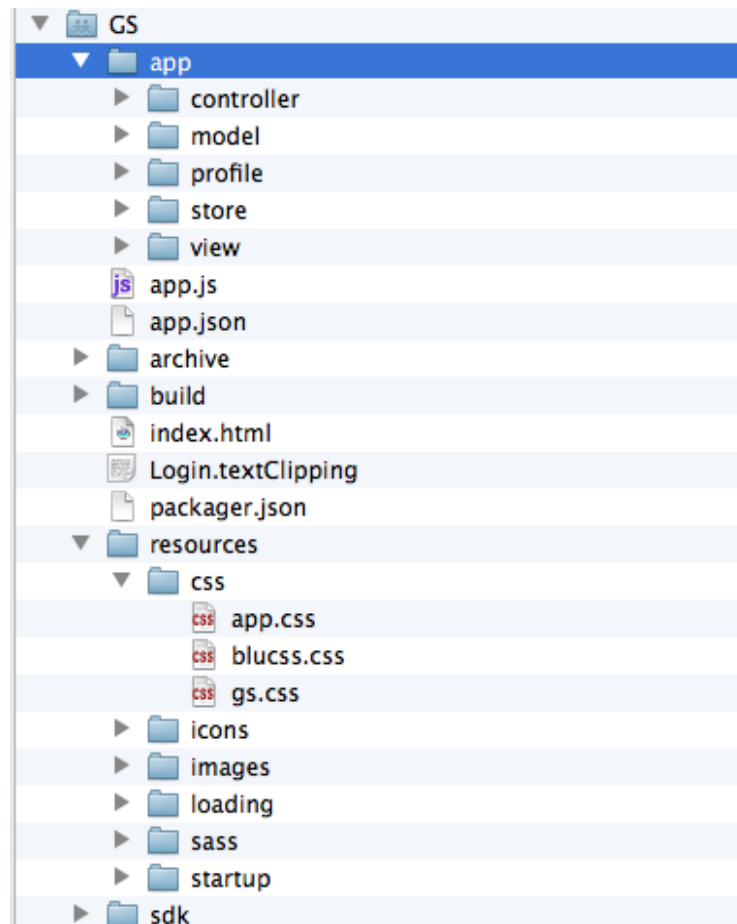
This will generate a skeleton Sencha Touch application namespaced to the GS variable (short for Getting Started) and located in the directory ../GS (one level up from the Sencha Touch SDK directory). The skeleton app contains all the files you need to create a Touch application, including the default index.html, a copy of the Touch SDK, CSS, images and configuration files for packaging your app for native.

Lets check if your application has generated successfully by opening it in a web browser. Assuming you extracted the SDK to your webroot folder, you should be able to navigate to <http://localhost/GS> and see the default app:



Explore the code

Open up the GS directory in your favorite IDE or Text editor. The directory structure looks like this:



Here's a description of each file and directory:

- **app** - directory containing the Models, Views, Controllers and Stores for your app.
- **app.js** - the main Javascript entry point for your app.
- **app.json** - your app configuration file - used by the Builder to create a minified version of your app.
- **index.html** - The HTML file for your app.
- **packager.json** - The configuration file used by the Packager to create native versions of your app for iOS and Android app stores.
- **resources** - directory containing the CSS and Images for your app
- **sdk** - A copy of the Sencha Touch SDK. You shouldn't need to change the contents of this folder.

Open *app.js*, the main entry point for your app, in your editor:

```

1  //<debug>
2  Ext.Loader.setPath({
3      'Ext': 'sdk/src'  ← Location of the Sencha Touch source files
4  });
5  //</debug>
6
7  Ext.application({
8      name: 'GS',  ← Your application namespace. All classes in your app
9                  will start with GS, e.g. GS.view.Main
10
11      requires: [
12          'Ext.MessageBox'  ← Array of required classes. This app uses a MessageBox,
13                             so we require it here.
14      ],
15
16      views: ['Main'],  ← A list of the View classes we use in our app.
17
18      icon: {
19          57: 'resources/icons/Icon.png',  ← A list of the icons used when users add
20          72: 'resources/icons/Icon~ipad.png',  ← your app to their home screen on iOS
21          114: 'resources/icons/Icon@2x.png',  ← devices. We specify different sized icons
22          144: 'resources/icons/Icon~ipad@2x.png'  ← which will be used under different conditions
23                                                     (iPad, retina screen, etc)
24      },
25
26      phoneStartupScreen: 'resources/loading/Homescreen.jpg',  ← If a user adds your app to
27      tabletStartupScreen: 'resources/loading/Homescreen~ipad.jpg',  ← their home screen on iOS,
28                                                                     these images will be shown
29                                                                     while the app loads.
30
31      launch: function() {
32          // Destroy the #appLoadingIndicator element  ←
33          Ext.fly('appLoadingIndicator').destroy();
34
35          // Initialize the main view
36          Ext.Viewport.add(Ext.create('GS.view.Main'));
37      },
38
39      onUpdated: function() {  ← This function is called if there is an update for your app once
40          Ext.Msg.confirm(  ← it has been installed in a production environment.
41              "Application Update",
42              "This application has just successfully been updated to the latest version. Reload now?",
43              function() {
44                  window.location.reload();
45              }
46          );
47      }
48  });

```

Components

Most of the visual classes you interact with in Sencha Touch are Components. Every Component in Sencha Touch is a subclass of *Ext.Component*, which means they can all:

- Render themselves onto the page using a template
- Show and hide themselves at any time
- Center themselves on the screen
- Enable and disable themselves

They can also do a few more advanced things:

- Float above other components (windows, message boxes and overlays)
- Change size and position on the screen with animation
- Dock other Components inside itself (useful for toolbars)
- Align to other components, allow themselves to be dragged around, make their content scrollable & more

What is a Container?

Every Component you create has all of the abilities above, but applications are made up of lots of components, usually nested inside one another. Containers are just like Components, but in addition to the capabilities above they can also allow you to render and arrange child Components inside them. Most apps have a single top-level Container called a Viewport, which takes up the entire screen. Inside of this are child components, for example in a mail app the Viewport Container's two children might be a message List and an email preview pane.

Containers give the following extra functionality:

- Adding child Components at instantiation and run time
- Removing child Components
- Specifying a Layout

Layouts determine how the child Components should be laid out on the screen. In our mail app example we'd use an HBox layout so that we can pin the email list to the left hand edge of the screen and allow the preview pane to occupy the rest. There are several layouts in Sencha Touch 2, each of which help you achieve your desired application structure, further explained in the Layout guide.

Instantiating Components

Components are created the same way as all other classes in Sencha Touch - using *Ext.create*. Here's how we can create a Panel:

```
var panel = Ext.create('Ext.Panel', {  
    html: 'This is my panel'  
});
```

This will create a Panel instance, configured with some basic HTML content. A Panel is just a simple Component that can render HTML and also contain other items. In this case we've created a Panel instance but it won't show up on the screen yet because items are not rendered immediately after being instantiated. This allows us to create some components and move them around before rendering and laying them out, which is a good deal faster than moving them after rendering.

To show this panel on the screen now we can simply add it to the global Viewport:

```
Ext.Viewport.add(panel);
```

Panels are also Containers, which means they can contain other Components, arranged by a layout. Let's revisit the above example now, this time creating a panel with two child Components and a hbox layout:

```
var panel = Ext.create('Ext.Panel', {
    layout: 'hbox',

    items: [
        {
            xtype: 'panel',
            flex: 1,
            html: 'Left Panel, 1/3rd of total size',
            style: 'background-color: #5E99CC;'
        },
        {
            xtype: 'panel',
            flex: 2,
            html: 'Right Panel, 2/3rds of total size',
            style: 'background-color: #759E60;'
        }
    ]
});

Ext.Viewport.add(panel);
```

This time we created 3 Panels - the first one is created just as before but the inner two are declared inline using an *xtype*. *Xtype* is a convenient way of creating Components without having to go through the process of using *Ext.create* and specifying the full class name, instead you can just provide the *xtype* for the class inside an object and the framework will create the components for you.

We also specified a layout for the top level panel - in this case *hbox*, which splits the horizontal width of the parent panel based on the 'flex' of each child. For example, if the parent Panel above is 300px wide then the first child will be flexed to 100px wide and the second to 200px because the first one was given flex: 1 and the second flex: 2.

Configuring Components

Whenever you create a new Component you can pass in configuration options. All of the configurations for a given Component are listed in the "Config options" section of its class docs page. You can pass in any number of configuration options when you instantiate the Component, and modify any of them at any point later. For example, we can easily modify the html content of a Panel after creating it:

```
//we can configure the HTML when we instantiate the Component
```

```

var panel = Ext.create('Ext.Panel', {
    fullscreen: true,
    html: 'This is a Panel'
});

//we can update the HTML later using the setHtml method:
panel.setHtml('Some new HTML');

//we can retrieve the current HTML using the getHtml method:
Ext.Msg.alert(panel.getHtml()); //alerts "Some new HTML"

```

Every config has a getter method and a setter method - these are automatically generated and always follow the same pattern. For example, a config called 'html' will receive *getHtml* and *setHtml* methods, a config called *defaultType* will receive *getDefaultType* and *setDefaultType* methods, and so on.

Using xtype

xtype is an easy way to create Components without using the full class name. This is especially useful when creating a Container that contains child Components. An *xtype* is simply a shorthand way of specifying a Component - for example you can use *xtype*: 'panel' instead of typing out `Ext.panel.Panel`.

Sample usage:

```

Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'fit',

    items: [
        {
            xtype: 'panel',
            html: 'This panel is created by xtype'
        },
        {
            xtype: 'toolbar',
            title: 'So is the toolbar',
            docked: 'top'
        }
    ]
});

```

List of xtypes

This is the list of all *xtypes* available in Sencha Touch 2:

xtype	Class
-----	-----
actionsheet	Ext.ActionSheet
audio	Ext.Audio
button	Ext.Button
component	Ext.Component
container	Ext.Container
image	Ext.Image

label	Ext.Label
loadmask	Ext.LoadMask
map	Ext.Map
mask	Ext.Mask
media	Ext.Media
panel	Ext.Panel
segmentedbutton	Ext.SegmentedButton
sheet	Ext.Sheet
spacer	Ext.Spacer
title	Ext.Title
titlebar	Ext.TitleBar
toolbar	Ext.Toolbar
video	Ext.Video
carousel	Ext.carousel.Carousel
carouselindicator	Ext.carousel.Indicator
navigationview	Ext.navigation.View
datepicker	Ext.picker.Date
picker	Ext.picker.Picker
pickerslot	Ext.picker.Slot
slider	Ext.slider.Slider
thumb	Ext.slider.Thumb
tabbar	Ext.tab.Bar
tabpanel	Ext.tab.Panel
tab	Ext.tab.Tab
viewport	Ext.viewport.Default

DataView Components

dataview	Ext.dataview.DataView
list	Ext.dataview.List
listitemheader	Ext.dataview.ListItemHeader
nestedlist	Ext.dataview.NestedList
dataitem	Ext.dataview.component.DataItem

Form Components

checkboxfield	Ext.field.Checkbox
datepickerfield	Ext.field.DatePicker
emailfield	Ext.field.Email
field	Ext.field.Field
hiddenfield	Ext.field.Hidden
input	Ext.field.Input
numberfield	Ext.field.Number
passwordfield	Ext.field.Password
radiofield	Ext.field.Radio
searchfield	Ext.field.Search
selectfield	Ext.field.Select
sliderfield	Ext.field.Slider
spinnerfield	Ext.field.Spinner
textfield	Ext.field.Text
textareafield	Ext.field.TextArea
textareainput	Ext.field.TextAreaInput
togglefield	Ext.field.Toggle
urlfield	Ext.field.Url
fieldset	Ext.form.FieldSet
formpanel	Ext.form.Panel

Adding Components to Containers

As we mentioned above, Containers are special Components that can have child Components arranged by a Layout. One of the code samples above showed how to create a Panel with 2 child Panels already defined inside it but it's easy to do this at run time too:

```
//this is the Panel we'll be adding below
var aboutPanel = Ext.create('Ext.Panel', {
    html: 'About this app'
});

//this is the Panel we'll be adding to
var mainPanel = Ext.create('Ext.Panel', {
    fullscreen: true,

    layout: 'hbox',
    defaults: {
        flex: 1
    },

    items: {
        html: 'First Panel',
        style: 'background-color: #5E99CC;'
    }
});

//now we add the first panel inside the second
mainPanel.add(aboutPanel);
```

Here we created three Panels in total. First we made the aboutPanel, which we might use to tell the user a little about the app. Then we create one called mainPanel, which already contains a third Panel in its items configuration, with some dummy text ("First Panel"). Finally, we add the first panel to the second by calling the add method on mainPanel.

In this case we gave our mainPanel another hbox layout, but we also introduced some defaults. These are applied to every item in the Panel, so in this case every child inside mainPanel will be given a flex: 1 configuration. The effect of this is that when we first render the screen only a single child is present inside mainPanel, so that child takes up the full width available to it. Once the mainPanel.add line is called though, the aboutPanel is rendered inside of it and also given a flex of 1, which will cause it and the first panel to both receive half the full width of the mainPanel.

Likewise, it's easy to remove items from a Container:

```
mainPanel.remove(aboutPanel);
```

After this line is run everything is back to how it was, with the first child panel once again taking up the full width inside mainPanel.

Showing and Hiding Components

Every Component in Sencha Touch can be shown or hidden with a simple API. Continuing with the mainPanel example above, here's how we can hide it:

```
mainPanel.hide();
```

This will hide the mainPanel itself and any child Components inside it. Showing the Component again is predictably easy:

```
mainPanel.show();
```

Again, this will restore the visibility of mainPanel and any child Components inside it.

Events

All Components fire events, which you can listen to and take action on. For example, whenever a Text field is typed into, its 'change' event is fired. You can listen to that event by simply passing a listeners config:

```
Ext.create('Ext.form.Text', {
    label: 'Name',
    listeners: {
        change: function(field, newValue, oldValue) {
            myStore.filter('name', newValue);
        }
    }
});
```

Every time the value of the text field changes, the 'change' event is fired and the function we provided called. In this case we filtered a Store by the name typed in to it, but we could have provided any other function there.

Lots of events are fired by Sencha Touch components, allowing you to easily hook into most aspects of an Application's behavior. They can also be specified after the Component has been created.

For example, let's say you have a dashboard that polls for live updates, but you don't want it to poll if the dashboard is not visible, you could simply hook into the dashboard's show and hide events:

```
dashboard.on({
    hide: MyApp.stopPolling,
    show: MyApp.startPolling
});
```

Docking

Sencha Touch has the ability to dock Components within other Containers. For example we may be using an hbox layout and want to place a banner to the top - docking gives us an easy way to do this without having to nest Containers inside one another:



The Layout Guide has a full discussion of docking and all of the other layout options.

Destroying Components

Because most mobile devices have a limited amount of memory it is often a good idea to destroy Components when you know you won't need them any more. It's not the first thing you should be thinking of when you create an app, but it's a good way to optimize the experience your users get when you push the app into production. Destroying a Component is easy:

```
mainPanel.destroy();
```

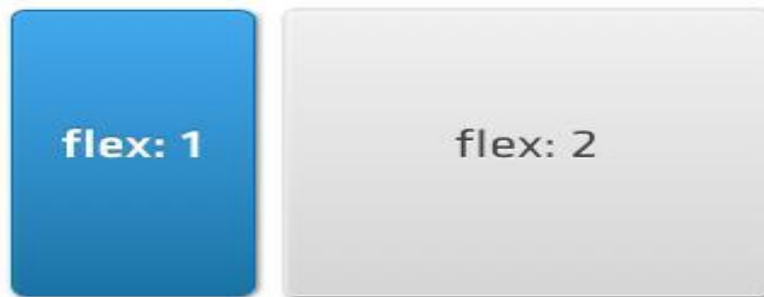
This will remove the mainPanel from the DOM and remove any event listeners it has set up on specific DOM elements. It will also destroy any instances that the Panel uses internally, and call destroy on all of its child components. After a Component is destroyed all of its children will also be gone, it will no longer be in the DOM and any references to it will no longer be valid.

Layouts

Intro and HBox

Layouts describe the sizing and positioning on Components in your app. For example, an email client might have a list of messages pinned to the left, taking say one third of the available width, and a message-viewing panel in the rest of the screen.

We can achieve this using an hbox layout and its ability to 'flex' items within it. Flexing means we divide the available area up based on the *flex* of each child component, so to achieve our example above we can set up flexes like this:



The code for this is simple, we just need to specify the 'hbox' layout on any Container, then supply a flex for each of the child components (Panels in this case):

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'hbox',
    items: [
        {
            xtype: 'panel',
            html: 'message list',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'message preview',
            flex: 2
        }
    ]
});
```

This creates a Container that fills the screen, and inside of it creates a message list panel and a preview panel. Their relative flex configs of 1 and 2 means that the message list will take up one third of the available width, with the preview taking the remaining two thirds. If our Container was 300px wide, the first item (flex: 1) will be 100px wide and the second (flex: 2) will be 200px wide.

The hbox is one of the most useful layouts as it can arrange components in a wide variety of horizontal combinations. See the HBox documentation for more examples.

VBox Layout

VBox is much like HBox, just vertical instead of horizontal. Again, we can visualize this easily as a set of stacked boxes:



The code to create this is almost identical to the example above - we just traded layout: *'hbox'* for layout: *'vbox'*:

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'vbox',
    items: [
        {
            xtype: 'panel',
            html: 'message list',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'message preview',
            flex: 2
        }
    ]
});
```

If our Container was 300px tall, the first panel (flex: 1) will be 100px tall, and the second (flex: 2) will be 200px tall. See the VBox documentation for more examples.

Card Layout

Sometimes you want to show several screens worth of information but you've only got a small screen to work with. TabPanels and Carousels both enable you to see one screen of many at a time, and underneath they both use a Card Layout.

Card Layout takes the size of the Container it is applied to and sizes the currently active item to fill the Container completely. It then hides the rest of the items, allowing you to change which one is currently visible but only showing one at once:



Here the gray box is our Container, and the blue box inside it is the currently active card. The three other cards are hidden from view, but can be swapped in later. While it's not too common to create

Card layouts directly, you can do so like this:

```
var panel = Ext.create('Ext.Panel', {
    layout: 'card',
    items: [
        {
            html: "First Item"
        },
        {
            html: "Second Item"
        },
        {
            html: "Third Item"
        },
        {
            html: "Fourth Item"
        }
    ]
});

panel.setActiveItem(1);
```

Here we create a Panel with a Card Layout and later set the second item active (the active item index is zero-based, so 1 corresponds to the second item). Normally you're better off using a Tab Panel or a Carousel.

Fit Layout

Fit Layout is probably the simplest layout available. All it does is make a child component fit to the full size of its parent Container:



For example, if you have a parent Container that is 200px by 200px and give it a single child component and a 'fit' layout, the child component will also be 200px by 200px:

```
var panel = Ext.create('Ext.Panel', {
    width: 200,
    height: 200,
    layout: 'fit',

    items: {
        xtype: 'panel',
        html: 'Also 200px by 200px'
    }
});

Ext.Viewport.add(panel);
```

Please note that if you add more than one item into a container with a fit layout, only the first item will be visible. If you want to switch between multiple items use the Card layout instead.

Docking

Every layout is capable of docking items inside it. Docking enables you to place additional Components at the top, right, bottom or left edges of the parent Container, resizing the other items as necessary.

For example, coming back to our first hbox layout above, let's imagine we want to dock another component at the top, like this:



This is often used for things like toolbars and banners, and is easy to achieve using the docked: 'top' configuration:

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'hbox',
    items: [
        {
            docked: 'top',
            xtype: 'panel',
            height: 20,
            html: 'This is docked to the top'
        },
        {
            xtype: 'panel',
            html: 'message list',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'message preview',
            flex: 2
        }
    ]
});
```

You can add any number of docked items by simply providing the dock configuration on the child components you want docked. You can also dock items on any side, for example if we want to do this with our previous vbox example:



We can achieve it by specifying docked: 'left':

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'vbox',
    items: [
        {
            docked: 'left',
            xtype: 'panel',
            width: 100,
            html: 'This is docked to the left'
        },
        {
            xtype: 'panel',
            html: 'message list',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'message preview',
            flex: 2
        }
    ]
});
```

You can add multiple docked items on each side (for example docking several items in the 'bottom' position).

The class system

Sencha Touch 2 uses the state of the art class system developed for Ext JS 4. It makes it easy to create new classes in JavaScript, providing inheritance, dependency loading, mixins, powerful configuration options, and lots more.

At its simplest, a class is just an object with some functions and properties attached to it. For instance, here's a class for an animal, recording its name and a function that makes it speak:

```
Ext.define('Animal', {
    config: {
        name: null
    },

    constructor: function(config) {
        this.initConfig(config);
    },

    speak: function() {
        alert('grunt');
    }
});
```

We now have a class called *Animal*, where each animal can have a name and speak. To create a new instance of animal, we just use *Ext.create*:

```
var bob = Ext.create('Animal', {
    name: 'Bob'
});

bob.speak(); //alerts 'grunt'
```

Here we created an *Animal* called Bob and commanded it to speak. Now that we've created a class and instantiated it, we can start improving what we have. At the moment we don't know Bob's species, so let's clear that up with a *Human* subclass:

```
Ext.define('Human', {
    extend: 'Animal',

    speak: function() {
        alert(this.getName());
    }
});
```

Now we've got a new class that inherits from *Animal*, therefore gaining all of its functions and configurations. We actually overrode the *speak* function because most humans are smart enough to say their name instead of grunt. Now, let's make Bob a human:

```
var bob = Ext.create('Human', {
    name: 'Bob'
});

bob.speak(); //alerts 'Bob'
```

We used a magical function when adding the *Human* subclass. You'll notice that we didn't actually define a *getName* function on our *Animal* class, so where did it come from? Part of the class system is the ability to give classes configuration options, which each automatically give you the following:

- a getter function that returns the current value, in this case *getName()*.
- a setter function that sets a new value, in this case *setName()*.
- an applier function called by the setter that lets you run a function when a configuration changes, in this case *applyName()*.
- a updater function called by the setter than run when the value for a configuration changes, in this case *updateName()*.

The getter and setter functions are generated for free and are the recommended way to store data in a class. Every component in Sencha Touch uses the class system and the generated functions always follow the same pattern so if you know a config you already know how to get and set its value.

It also makes your code cleaner. For example, if you wanted to always ask the user if she really wants to change Bob's name, you can just define an *applyName* function that will automatically be called:

```
Ext.define('Human', {
    extend: 'Animal',

    applyName: function(newName, oldName) {
        return confirm('Are you sure you want to change name to ' + newName + '?')? newName :
        oldName;
    }
});
```

We're just using the browser's built in confirm function, which opens a dialog asking the user the question and offering "Yes" and "No" as answers. The applier functions allow you to cancel the name change if you return false. As it happens the confirm function will return either new or old name depending on whether the user clicks Yes or No.

If we make a new Bob and try to change his name, but then click No when prompted, his name won't change after all:

```
var bob = Ext.create('Person', {
    name: 'Bob'
});

bob.setName('Fred'); //opens a confirm box, but we click No

bob.speak(); //still alerts 'Bob'
```

The apply function is also a great place where you should *transform* your value. Remember whatever this returns with will be the new value for this configuration. A good example of this would be to pretend a title to the name:

```
Ext.define('Human', {
    extend: 'Animal',

    applyName: function(newName, oldName) {
        return 'Mr. ' + newName;
    }
});
```


The other config method is update. The update method (*updateName()* in this case) is only called when the value of the config changes. So in the following case, *updateName()* would not be called:

```
var bob = Ext.create('Person', {
    name: 'Bob'
});

bob.setName('Bob'); // The name is the same, so update is not called
```

So when we use the update method, the config is changing. This should be the place where you update your component, or do whatever you need to do when the value of your config changes. In this example, we will show an alert:

```
Ext.define('Human', {
    extend: 'Animal',

    updateName: function(newName, oldName) {
        alert('Name changed. New name is: ' + newName);
    }
});
```

Bear in mind that the update and apply methods get called on component instantiation too, so in the following example, we would get 2 alerts:

```
// creating this will cause the name config to update, therefor causing the alert
var bob = Ext.create('Person', {
    name: 'Bob'
});

// changing it will cause the alert to show again
bob.setName('Ed');
```

We've basically already learned the important parts of classes, as follows:

- All classes are defined using `Ext.define`, including your own classes
- Most classes extend other classes, using the `extend` syntax
- Classes are created using `Ext.create`, for example
- `Ext.create('SomeClass', {some: 'configuration'})`
- Always use the config syntax to get automatic getters and setters and have a much cleaner codebase

At this point you can already go about creating classes in your app, but the class system takes care of a few more things that will be helpful to learn are a few other things the class system does.

Dependencies and Dynamic Loading

Most of the time, classes depend on other classes. The Human class above depends on the Animal class because it extends it - we depend on Animal being present to be able to define Human. Sometimes you'll make use of other classes inside a class, so you need to guarantee that those classes are on the page. Do this with required syntax:

```
Ext.define('Human', {
    extend: 'Animal',

    requires: 'Ext.MessageBox',

    speak: function() {
        Ext.Msg.alert(this.getName(), "Speaks...");
    }
});
```

When you create a class in this way, Sencha Touch checks to see if *Ext.MessageBox* is already loaded and if not, loads the required class file immediately with AJAX.

Ext.MessageBox itself may also have classes it depends on, which are then also loaded automatically in the background. Once all the required classes are loaded, the Human class is defined and you can start using *Ext.create* to instantiate people. This works well in development mode as it means you don't have to manage the loading of all your scripts yourself, but not as well in production because loading files one by one over an internet connection is rather slow.

In production, we really want to load just one JavaScript file, ideally containing only the classes that our application actually uses. This is really easy in Sencha Touch 2 using the JSBuilder tool, which analyzes your app and creates a single file build that contains all of your classes and only the framework classes your app actually uses.

Each approach has its own pros and cons, but can we have the good parts of both without the bad, too? The answer is yes, and we've implemented the solution in Sencha Touch 2.

Naming Conventions

Using consistent naming conventions throughout your code base for classes, namespaces and filenames helps keep your code organized, structured, and readable.

Classes

Class names may only contain alphanumeric characters. Numbers are permitted but are discouraged in most cases, unless they belong to a technical term. Do not use underscores, hyphens, or any other non-alphanumeric character. For example:

- MyCompany.useful_util.Debug_Toolbar is discouraged
- MyCompany.util.Base64 is acceptable

Class names should be grouped into packages where appropriate and properly namespaced using object property dot notation (.). At the minimum, there should be one unique top-level namespace followed by the class name. For example:

- `MyCompany.data.CoolProxy`
- `MyCompany.Application`

The top-level namespaces and the actual class names should be in CamelCase, everything else should be all lower-cased. For example:

- `MyCompany.form.action.AutoLoad`

Classes that are not distributed by Sencha should never use `Ext` as the top-level namespace.

Acronyms should also follow CamelCase convention, for example:

- `Ext.data.JsonProxy` instead of `Ext.data.JSONProxy`
- `MyCompany.util.HtmlParser` instead of `MyCompany.parser.HTMLParser`
- `MyCompany.server.Http` instead of `MyCompany.server.HTTP`

Source Files

The names of the classes map directly to the file paths in which they are stored. As a result, there must only be one class per file. For example:

- `Ext.mixin.Observable` is stored in `path/to/src/Ext/mixin/Observable.js`
- `Ext.form.action.Submit` is stored in `path/to/src/Ext/form/action/Submit.js`
- `MyCompany.chart.axis.Numeric` is stored in `path/to/src/MyCompany/chart/axis/Numeric.js`

`path/to/src` is the directory of your application's classes. All classes should stay under this common root and should be properly namespaced for the best development, maintenance, and deployment experience.

Methods and Variables

Similarly to class names, method and variable names may only contain alphanumeric characters. Numbers are permitted but are discouraged in most cases, unless they belong to a technical term. Do not use underscores, hyphens, or any other nonalphanumeric character.

Method and variable names should always use CamelCase. This also applies to acronyms.

Here are a few examples:

Acceptable method names: `encodeUsingMd5()`

- `getHtml()` instead of `getHTML()`
- `getJsonResponse()` instead of `getJSONResponse()`
- `parseXmlContent()` instead of `parseXMLContent()`

Acceptable variable names:

- `var isGoodName`
- `var base64Encoder`
- `var xmlReader`
- `var httpServer`

Properties

Class property names follow the same convention as method and variable names, except the case when they are static constants. Static class properties that are constants should be all upper-cased, for example:

- `Ext.MessageBox.YES = "Yes"`
- `Ext.MessageBox.NO = "No"`
- `MyCompany.alien.Math.PI = "4.13"`

Working with classes in Sencha Touch 2.0

Declaration

The Old Way

If you've developed with Sencha Touch 1.x, you are certainly familiar with *Ext.extend* to create a class:

```
var MyPanel = Ext.extend(Object, { ... });
```

This approach is easy to follow when creating a new class that inherits from another. Other than direct inheritance, however, there wasn't a fluent API for other aspects of class creation, such as configuration, statics, and mixings. We will be reviewing these items in detail shortly.

Let's take a look at another example:

```
My.cool.Panel = Ext.extend(Ext.Panel, { ... });
```

In this example we want to namespace our new class and make it extend from *Ext.Panel*. There are two concerns we need to address:

- *My.cool* needs to be an existing object before we can assign *Panel* as its property.
- *Ext.Panel* needs to exist/be loaded on the page before it can be referenced.

The first item is usually solved with *Ext.namespace* (aliased by *Ext.ns*). This method recursively traverses through the object/property tree and creates them if they don't exist yet. The boring part is you need to remember adding them above *Ext.extend* all the time, like this:

```
Ext.ns('My.cool');  
My.cool.Panel = Ext.extend(Ext.Panel, { ... });
```

The second issue, however, is not easy to address because *Ext.Panel* might depend on many other classes that it directly/indirectly inherits from, and in turn, these dependencies might depend on other classes to exist. For that reason, applications written before Sencha Touch 2 usually include the whole library in the form of *ext-all.js* even though they might only need a small portion of the framework.

The New Way

Sencha Touch 2 eliminates all those drawbacks with just one single method you need to remember for class creation: *Ext.define*. Its basic syntax is as follows:

```
Ext.define(className, members, onClassCreated);
```

Let's look at each part of this:

- *className* is the class name
- *members* is an object represents a collection of class members in key-value pairs
- *onClassCreated* is an optional function callback to be invoked when all dependencies of this class are ready, and the class itself is fully created. Due to the new asynchronous nature of

class creation, this callback can be useful in many situations. These will be discussed further in Section IV.

Example

```
Ext.define('My.sample.Person', {
    name: 'Unknown',

    constructor: function(name) {
        if (name) {
            this.name = name;
        }
    },

    eat: function(foodType) {
        alert(this.name + " is eating: " + foodType);
    }
});

var aaron = Ext.create('My.sample.Person', 'Aaron');
aaron.eat("Salad"); // alert("Aaron is eating: Salad");
```

Note we created a new instance of `My.sample.Person` using the `Ext.create()` method. We could have used the new keyword (`new My.sample.Person()`). However it is recommended that you always use `Ext.create` since it allows you to take advantage of dynamic loading.

Configuration

In Sencha Touch 2, we introduce a dedicated `config` property that is processed by the powerful `Ext.Class` preprocessors before the class is created. Features include:

- Configurations are completely encapsulated from other class members.
- Getter and setter, methods for every config property are automatically generated into the class prototype during class creation if the class does not have these methods already defined.
- An `apply` method is also generated for every config property. The auto-generated setter method calls the `apply` method internally before setting the value. Override the `apply` method for a config property if you need to run custom logic before setting the value. If `apply` does not return a value then the setter will not set the value. For an example see `applyTitle` below.

Here's an example:

```
Ext.define('My.own.WindowBottomBar', {});

Ext.define('My.own.Window', {

    /** @readonly */
    isWindow: true,

    config: {
        title: 'Title Here',

        bottomBar: {
            enabled: true,
```

```

        height: 50,
        resizable: false
    },

    constructor: function(config) {
        this.initConfig(config);
    },

    applyTitle: function(title) {
        if (!Ext.isString(title) || title.length === 0) {
            console.log('Error: Title must be a valid non-empty string');
        }
        else {
            return title;
        }
    },

    applyBottomBar: function(bottomBar) {
        if (bottomBar && bottomBar.enabled) {
            if (!this.bottomBar) {
                return Ext.create('My.own.WindowBottomBar', bottomBar);
            }
            else {
                this.bottomBar.setConfig(bottomBar);
            }
        }
    }
});

```

And here's an example of how it can be used:

```

var myWindow = Ext.create('My.own.Window', {
    title: 'Hello World',
    bottomBar: {
        height: 60
    }
});

```

```
console.log(myWindow.getTitle()); // logs "Hello World"
```

```
myWindow.setTitle('Something New');
console.log(myWindow.getTitle()); // logs "Something New"
```

```
myWindow.setTitle(null); // logs "Error: Title must be a valid non-empty string"
```

```
myWindow.setBottomBar({ height: 100 }); // Bottom bar's height is changed to 100
```

Statics

Static members can be defined using the statics config, as follows:

```

Ext.define('Computer', {
    statics: {
        instanceCount: 0,
    }
});

```

```

    factory: function(brand) {
        // 'this' in static methods refer to the class itself
        return new this({brand: brand});
    },

    config: {
        brand: null
    },

    constructor: function(config) {
        this.initConfig(config);

        // the 'self' property of an instance refers to its class
        this.self.instanceCount++;
    }
});

var dellComputer = Computer.factory('Dell');
var appleComputer = Computer.factory('Mac');

alert(appleComputer.getBrand()); // using the auto-generated getter to get the value of a config property.
Alerts "Mac"

alert(Computer.instanceCount); // Alerts "2"

```

Error Handling and debugging

Sencha Touch 2 includes some useful features that will help you with debugging and error handling.

You can use `Ext.getDisplayName()` to get the display name of any method. This is especially useful for throwing errors that have the class name and method name in their description, such as:

```
throw new Error(['+ Ext.getDisplayName(arguments.callee) +'] Some message here');
```

When an error is thrown in any method of any class defined using `Ext.define()`, you should see the method and class names in the call stack if you are using a WebKit based browser (Chrome or Safari). For example, here is what it would look like in Chrome:

✖ ▼ Uncaught Error	Panel.js:9
Ext.define.doSomething	Panel.js:9
Ext.application.launch	app.js:15
Ext.define.onBeforeLaunch	Application.js:185
(anonymous function)	Application.js:154
isEvent	ext-debug.js:16171
fire	ext-debug.js:16320
Ext.EventManager.onDocumentReady	ext-debug.js:16478
Loader.Ext.Loader.onReady.fn	ext-debug.js:7501
Loader.Ext.Loader.onReady	ext-debug.js:7506
Ext.onReady	ext-debug.js:17171
Ext.define.constructor	Application.js:148
Ext.Class.Class.newClass	ext-debug.js:5208
anonymous	:3
Manager.Ext.ClassManager.instantiate	ext-debug.js:6228
(anonymous function)	ext-debug.js:2159
(anonymous function)	ext-debug.js:9149
isEvent	ext-debug.js:16171
fire	ext-debug.js:16320
Ext.EventManager.onDocumentReady	ext-debug.js:16478
Loader.Ext.Loader.onReady.fn	ext-debug.js:7501
Loader.Ext.Loader.triggerReady	ext-debug.js:7476
(anonymous function)	ext-debug.js:2145
Loader.Ext.Loader.refreshQueue	ext-debug.js:7085
Loader.Ext.Loader.refreshQueue	ext-debug.js:7086
Loader.Ext.Loader.refreshQueue	ext-debug.js:7086
Loader.Ext.Loader.refreshQueue	ext-debug.js:7086
Loader.Ext.Loader.refreshQueue	ext-debug.js:7086
Loader.Ext.Loader.refreshQueue	ext-debug.js:7086
Loader.Ext.Loader.onFileLoaded	ext-debug.js:7372
(anonymous function)	ext-debug.js:2145
onLoadFn	ext-debug.js:7104

Events

Events are fired whenever something interesting happens to one of your classes. For example, when any Component is rendered to the screen its painted event is fired. We can listen for that event by configuring a simple listeners config:

```
Ext.create('Ext.Panel', {
    html: 'My Panel',
    fullscreen: true,

    listeners: {
        painted: function() {
            Ext.Msg.alert('I was painted to the screen');
        }
    }
});
```

When you click the "Live Preview" button above you'll see the Panel rendered to the screen followed by our alert message. All of the events fired by a class are listed in its API page - for example Ext.Panel has 28 events at the time of writing.

Listening to Events

The painted event itself is useful in some cases but there are other events that you're much more likely to use. For example, Buttons fire tap events whenever they're tapped on:

```
Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'My Button',

    listeners: {
        tap: function() {
            alert("You tapped me");
        }
    }
});
```

We can add as many event listeners as we like. Here we're going to confound our user by calling this.hide() inside our tap listener to hide the Button, only to show it again a second later. When this.hide() is called, the Button is hidden and the hide event fired. The hide event in turn triggers our hide listener, which waits a second then shows the Button again:

```
Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'My Button',

    listeners: {
        tap: function() {
            this.hide();
        },
        hide: function() {
            //waits 1 second (1000ms) then shows the button again
            Ext.defer(function() {
```

```

        this.show();
    }, 1000, this);
    }
});

```

Event listeners are called every time an event is fired, so you can continue hiding and showing the button for all eternity.

Config-driven events

Most classes are reconfigurable at run time - e.g. you can change configurations like their height, width or content at any time and the Component will correctly update itself on screen. Many of these configuration changes trigger an event to be fired - for example 14 of Button's 24 events have names like widthchange, hiddenchange and centeredchange.

This time our tap handler is just going to call `this.setWidth()` to set a random width on our button. Our widthchange listener will immediately be informed of the change, along with the new and old width values:

```

Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me',

    listeners: {
        tap: function() {
            var randomWidth = 100 + Math.round(Math.random() * 200);

            this.setWidth(randomWidth);
        },
        widthchange: function(button, newWidth, oldWidth) {
            alert('My width changed from ' + oldWidth + ' to ' + newWidth);
        }
    }
});

```

Every event that ends in 'change' is fired as a result of a config option changing. Note that you listen to these events just like any other, it's just good to know the convention.

Adding Listeners Later

Every example so far has involved passing listeners in when the class is instantiated. If we already have an instance though, we can still add listeners later using the `on` function:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me'
});

myButton.on('tap', function() {
    alert("Event listener attached by .on");
});

```

You can add new listeners at any time this way. We can also combine these approaches, even listening to the same event more than once if we need to:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me',

    listeners: {
        tap: function() {
            alert('First tap listener');
        }
    }
});

myButton.on('tap', function() {
    alert("Second tap listener");
});

```

Both of your event listener functions are called, preserving the order they were added in.

Finally, we can specify multiple listeners using `.on`, just as we could with a listener configuration. Here's our random width setting button again:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me'
});

myButton.on({
    tap: function() {
        var randomWidth = 100 + Math.round(Math.random() * 200);

        this.setWidth(randomWidth);
    },
    widthchange: function(button, newWidth, oldWidth) {
        alert('My width changed from ' + oldWidth + ' to ' + newWidth);
    }
});

```

Removing Listeners

Just as we can add listeners at any time, we can remove them too, this time using `un`. In order to remove a listener, we need a reference to its function. In all of the examples above we've just passed a function straight into the listeners object or `.on` call, this time we're going to create the function a little earlier and link it into a variable called `doSomething`.

We'll pass our new `doSomething` function into our listeners object, which works just like before. This time however we'll add an `Ext.defer` function at the bottom that removes the listener again after 3 seconds. Clicking on the button in the first 3 seconds yields an alert message, after 3 seconds our listener is removed so nothing happens:

```

var doSomething = function() {
    alert('handler called');
};

var myButton = Ext.Viewport.add({

```

```

        xtype: 'button',
        text: 'My Button',
        centered: true,

        listeners: {
            tap: doSomething
        }
    });

    Ext.defer(function() {
        myButton.un('tap', doSomething);
    }, 3000);

```

In this example we're adding a button like before, but this time also adding a toggle button that will add and remove the tap listener as you toggle it. The listener starts off disabled, use the toggle button to enable and later disable it:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Listener Disabled'
});

var handler = function() {
    alert('listener called');
};

Ext.Viewport.add({
    xtype: 'togglefield',
    docked: 'bottom',
    label: 'Toggle Listener',

    listeners: {
        change: function(field, thumb, enabled) {
            if (enabled) {
                myButton.on('tap', handler);
                myButton.setText('Listener Enabled');
            } else {
                myButton.un('tap', handler);
                myButton.setText('Listener Disabled');
            }
        }
    }
});

```

Listener Options

There are a few additional options that we can pass into listeners.

a. Scope

Scope sets the value of this inside your handler function. By default this is set to the instance of the class firing the event, which is often (but not always) what you want. That's what allowed us to call this.hide() to hide the button in the second example near the start of this guide.

This time we'll create a Button and a Panel, then listen to the Button's 'tap' event with our handler running in Panel's scope. To do this we need to pass in an object instead of a handler function - this object contains the function plus the scope:

```
var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me'
});

var panel = Ext.create('Ext.Panel', {
    html: 'Panel HTML'
});

myButton.on({
    tap: {
        scope: panel,
        fn: function() {
            alert("Running in Panel's scope");
            alert(this.getHtml());
        }
    }
});
```

When you run this example, the value of this in the tap handler is the Panel. To see this we'd set the Panel's html configuration to 'Panel HTML' and then alerted this.getHtml() in our handler. When we tap the button we do indeed see the Panel's html being alerted.

b. Single

Sometimes we only want to listen to an event one time. The event itself might fire any number of times, but we only want to listen to it once. This is simple:

```
var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me',

    listeners: {
        tap: {
            single: true,
            fn: function() {
                alert("I will say this only once");
            }
        }
    }
});
```

c. Buffer

For events that are fired many times in short succession, we can throttle the number of times our listener is called by using the buffer configuration. In this case our button's tap listener will only be invoked once every 2 seconds, regardless of how many times you click it:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: 'Click me',

    listeners: {
        tap: {
            buffer: 2000,
            fn: function() {
                alert("I will say this only once every 2 seconds");
            }
        }
    }
});

```

d. Firing Your Own Events

Firing your own events is really simple - you can just call fireEvent with any event name. In this case we'll fire an event called myEvent that passes two arguments - the button itself and a number between 1 and 100:

```

var myButton = Ext.Viewport.add({
    xtype: 'button',
    centered: true,
    text: "Just wait 2 seconds",

    listeners: {
        myEvent: function(button, points) {
            alert('myEvent was fired! You score ' + points + ' points');
        }
    }
});

Ext.defer(function() {
    var number = Math.ceil(Math.random() * 100);

    myButton.fireEvent('myEvent', myButton, number);
}, 2000);

```

Once again we used Ext.defer to delay the function that fires our custom event, this time by 2 seconds. When the event is fired, our myEvent listener picks up on it and alerts the arguments we passed in.

Floating components

Centering a Component

You can center any component within its container in Sencha touch by using the centered configuration. This will always center the component, even when its parent containers size changes.

```
Ext.Viewport.add({
    xtype: 'panel',
    html: 'This is a centered panel',
    centered: true
});
```

When using centered, the width and height of the component is automatically set depending on the size of the content. However, if the content of the component is dynamic, like a scroller panel, the width and height must be set manually.

```
Ext.Viewport.add({
    xtype: 'panel',
    scrollable: true,
    html: 'This is a scrollable centered panel with some long content so it will scroll.',
    centered: true,
    width: 100,
    height: 100
});
```

Centered components are centered within their container. In the above examples we are adding a component into Ext.Viewport, so the component is centered in the center of the screen (as the Viewport is the full size of the screen). However, if we want, we can center a component within a random sized container.

Absolutely positioning a Component

You can also absolutely position a component in Sencha Touch using the top, right, bottom and left configurations of any component like this:

```
Ext.Viewport.add({
    xtype: 'panel',
    html: 'A floating component',
    top: 50,
    right: 5
});
```

And of course, because these position properties are all configurations, you can use the appropriate setters to change them at any time:

```
var panel = Ext.Viewport.add({
    xtype: 'panel',
    defaultType: 'button',
    layout: {
        type: 'vbox',
        align: 'stretch'
    },
    items: [
```



```

    {
      text: 'up',
      handler: function() {
        panel.setTop(panel.getTop() - 5);
      }
    },
    {
      text: 'down',
      handler: function() {
        panel.setTop(panel.getTop() + 5);
      }
    },
    {
      text: 'left',
      handler: function() {
        panel.setLeft(panel.getLeft() - 5);
      }
    },
    {
      text: 'right',
      handler: function() {
        panel.setLeft(panel.getLeft() + 5);
      }
    }
  ],
  top: 50,
  left: 5
});

```

Modal Components

Making a floating or centered container modal masks the rest of its parent container so there are less distraction for the user. You simply set the modal configuration to true.

```

Ext.Viewport.add({
  xtype: 'panel',
  html: 'This is a centered and modal panel',
  modal: true,
  centered: true
});

Ext.Viewport.setHtml('This content is in the viewport and masked
because the panel is modal. ');
Ext.Viewport.setStyleHtmlContent(true);

```

You can also use the *hideOnMaskTap* configuration to make the panel and mask disappear when a user taps on the mask:

```

Ext.Viewport.add({
  xtype: 'panel',
  html: 'This is a centered and modal panel',
  modal: true,
  hideOnMaskTap: true,
  centered: true
});

```

```
Ext.Viewport.setHtml('This content is in the viewport and masked  
because the panel is modal.<br /><br />You can also tap on the mask to  
hide the panel.');
```

```
Ext.Viewport.setStyleHtmlContent(true);
```

Environment detection

Commonly when building applications targeting mobile devices, you need to know specific information about the environment your application is running on. Perhaps the browser type, device name, or if specific functionality is available for use. The environment package in Sencha Touch gives you an easy API which allows you to find out all this information.

Operating System

You can detect the operating system your application is running on using *Ext.os.name*. This will return one of the following values:

- iOS
- Android
- webOS
- BlackBerry
- RIMTablet
- MacOS
- Windows
- Linux
- Bada
- Other

You can also use the *Ext.os.is* singleton to check if the current OS matches a certain operating system. For example, if you wanted to check if the current OS is Android, you could do:

```
if (Ext.os.is.Android) { ... }
```

You can do this with any of the above values:

```
if (Ext.os.is.MacOS) { ... }
```

You can also use it to detect if the device is an iPhone, iPad or iPod using *Ext.os.is*:

```
if (Ext.os.is.iPad) { ... }
```

The version of the OS can be also accessed using *Ext.os.version*:

```
alert('You are running: ' + Ext.os.name + ', version ' + Ext.os.version.version);
```

Browser

You can also find information about the browser you are running your application on by using *Ext.browser.name*. The lists of available values are:

- Safari
- Chrome
- Opera

- Dolphin
- webOSBrowser
- ChromeMobile
- Firefox
- IE
- Other

You can also use `Ext.browser.is` to check if the current browser is one of the above values:

```
if (Ext.browser.is.Chrome) { ... }
```

The `Ext.browser.is` singleton also has other useful information about the current browser that may be useful for your application:

- `Ext.browser.userAgent` - returns the current `userAgent`
- `Ext.browser.isSecure` - returns true if the current page is using SSL
- `Ext.browser.isStrict` - returns true if the browser is in strict mode
- `Ext.browser.engineName` - returns the browser engine name (WebKit, Gecko, Presto, Trident and Other)
- `Ext.browser.engineVersion` - returns the version of the browser engine

Features

You can use the `Ext.feature.has` singleton to check if a certain browser feature exists. For example, if you want to check if the browser supports canvas, you check do the following:

```
if (Ext.feature.has.Canvas) { ... }
```

The lists of available features are:

- Audio
- Canvas
- ClassList
- CreateContextualFragment
- Css3dTransforms
- CssAnimations
- CssTransforms
- CssTransitions
- DeviceMotion
- Geolocation
- History
- Orientation
- OrientationChange

- Range
- SQLiteDatabase
- Svg
- Touch
- Video
- Vml
- WebSockets

Using Ajax

Simple Requests with Ext.Ajax

AJAX requests are usually made to urls on the same domain as your application. For example, if your application is found at *http://myapp.com*, you can send AJAX requests to urls like *http://myapp.com/login.php* and *http://myapp.com/products/1.json* but not to other domains like *http://google.com*. This is because of browser security restrictions, but Sencha Touch does provide some alternatives to get around this, which we'll look at shortly (Cross-Domain Requests and JSON-P).

For now, here's how we can make an AJAX request to load data from a url on our domain:

```
Ext.Ajax.request({
  url: 'myUrl',
  callback: function(response) {
    console.log(response.responseText);
  }
});
```

Assuming your app is on *http://myapp.com*, this is going to send a GET request to *http://myapp.com/myUrl*. AJAX calls are asynchronous so once the response comes back, our callback function is called with the response. All we're doing above is logging the contents of the response to the console when the request has finished. Pretty simple so far, let's see what else we can do.

AJAX Options

Ext.Ajax takes a wide variety of options, including setting the method (GET, POST, PUT or DELETE), sending headers and setting params to be sent in the url. First let's see how to set the method so we send a POST request instead of GET:

```
Ext.Ajax.request({
  url: 'myUrl',
  method: 'POST',

  params: {
    username: 'Ed',
    password: 'not a good place to put a password'
  },

  callback: function(response) {
    console.log(response.responseText);
  }
});
```

When we set params like this, the request is automatically sent as a POST with the params object sent as form data. The request above is just like submitting a form with username and password fields.

If we wanted to send this as a GET request instead we can specify the method again, in which case our params are automatically escaped and appended to the url for us:

```
Ext.Ajax.request({
  url: 'myUrl',
  method: 'GET',
```

```

params: {
  username: 'Ed',
  password: 'bad place for a password'
},

callback: function(response) {
  console.log(response.responseText);
}
});

```

Which sends a request to:

http://mywebsite.com/myUrl?_dc=1329443875411&username=Ed&password=bad%20place%20for%20a%20password

You may have noticed that our last request created a url that contained "_dc=1329443875411". When you make a GET request like this, many web servers will cache the response and send you back the same thing every time you make the request. This speeds the web up, but is not always what you want. In fact in applications it's rarely what you want, so we "bust" the cache for you by adding a timestamp to every request. This tells the web server to treat it as a fresh, uncached request.

If you want to turn this behavior off, we can just set *disableCaching* to *false*:

```

Ext.Ajax.request({
  url: 'myUrl',
  method: 'GET',
  disableCaching: false,

  params: {
    username: 'Ed',
    password: 'bad place for a password'
  },

  callback: function(response) {
    console.log(response.responseText);
  }
});

```

Now our request no longer contains the cache busting string, and looks more like this:

<http://mywebsite.com/myUrl?username=Ed&password=bad%20place%20for%20a%20password>

Sending Headers

The final option we'll look at when it comes to customizing the request itself is the headers option. This enables you to send any custom headers you want to your server, which is often useful when the web server returns different content based on those headers. For example, if your web server returns JSON, XML or CSV based on which header it is passed, we can ask it for JSON like this:

```

Ext.Ajax.request({
  url: 'myUrl',

  headers: {
    "Content-Type": "application/json"
  },

```

```

    callback: function(response) {
        console.log(response.responseText);
    }
});

```

If you create a request like this and inspect it in Firebug/web inspector you'll see that the Content-Type header has been set to application/json. Your web server can pick up on this and send you the right response. You can pass any number of headers you like into the headers option.

Callback Options

Not every AJAX request succeeds. Sometimes the server is down, or your Internet connection drops, or something else bad happens. *Ext.Ajax* allows you to specify separate callbacks for each of these cases:

```

Ext.Ajax.request({
    url: 'myUrl',

    success: function(response) {
        console.log("Spiffing, everything worked");
    },

    failure: function(response) {
        console.log("Curses, something terrible happened");
    }
});

```

These do exactly what you'd expect them to do, and hopefully most of the time it is your success callback that gets called. It's pretty common to provide a success callback that updates the UI or does whatever else is needed by the application flow, and a failure handler that either retries the request or alerts the user that something went wrong.

You can provide *success/failure* and *callback* at the same time, so for this request if everything was ok our success function is called first, followed by the main *callback* function, otherwise it'll be *failure* followed by *callback*:

```

Ext.Ajax.request({
    url: 'myUrl',

    success: function(response) {
        console.log("Spiffing, everything worked");
    },

    failure: function(response) {
        console.log("Curses, something terrible happened");
    },

    callback: function(response) {
        console.log("It is what it is");
    }
});

```

Timeouts and Aborting Requests

Another way requests can fail is if the server took too long to respond and the request timed out. In

this case your *failure* function will be called, and the request object it is passed will have `timedout` true:

```
Ext.Ajax.request({
    url: 'myUrl',

    failure: function(response) {
        console.log(response.timedout); // logs true
    }
});
```

By default the timeout threshold is 30 seconds, but you can specify it per request by setting the `timeout` in millisecond. In this case our request will give up after 5 seconds:

```
Ext.Ajax.request({
    url: 'myUrl',
    timeout: 5000,

    failure: function(response) {
        console.log(response.timedout); // logs true
    }
});
```

It's also possible to abort requests that are currently outstanding. To do this you need to save a reference to the request object that *Ext.Ajax.request* gives you:

```
var myRequest = Ext.Ajax.request({
    url: 'myUrl',

    failure: function(response) {
        console.log(response.aborted); // logs true
    }
});

Ext.Ajax.abort(myRequest);
```

This time our failure callback is called and its response. `Aborted` property is set. We can use all of the error handling above in our apps:

```
Ext.Ajax.request({
    url: 'myUrl',

    failure: function(response) {
        if (response.timedout) {
            Ext.Msg.alert('Timeout', "The server timed out :(");
        } else if (response.aborted) {
            Ext.Msg.alert('Aborted', "Looks like you aborted the request");
        } else {
            Ext.Msg.alert('Bad', "Something went wrong with your request");
        }
    }
});
```

Cross-Domain Requests

A relatively new capability of modern browsers is called CORS, which stands for Cross-Origin Resource Sharing. This allows you to send requests to other domains without the usual security restrictions enforced by the browser. Sencha Touch 2 has support for CORS, though you'll probably need to do a little setup on your web server to enable it. If you're not familiar with what you need to do on your web server to enable CORS, a quick Google search should give you plenty of answers.

Assuming your server is set up though, sending a CORS request is easy:

```
Ext.Ajax.request({  
  url: 'http://www.somedomain.com/some/awesome/url.php',  
  withCredentials: true,  
  useDefaultXhrHeader: false  
});
```

Form Uploads

The final thing we'll cover is uploading forms. This is also really easy:

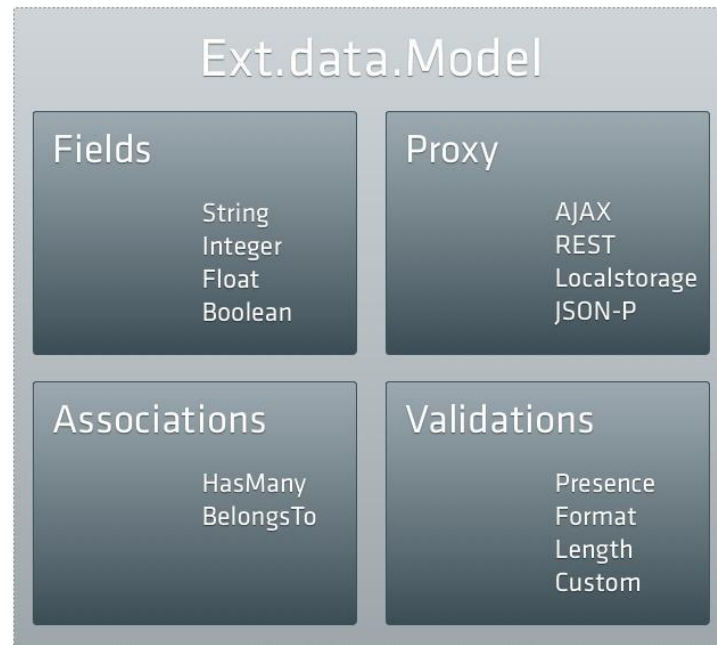
```
Ext.Ajax.request({  
  url: 'myUrl',  
  form: 'myFormId',  
  
  callback: function(response, successful) {  
    if (successful) {  
      Ext.Msg.alert('Success', 'We got your form submission');  
    } else {  
      Ext.Msg.alert('Fail', 'Hmm, that did not work');  
    }  
  }  
});
```

This finds a <form> tag on the page with id="myFormId", grabs its data and puts it into the request params object just like at the start of this guide. Then it submits it to the url you specified and calls your callbacks like normal.

Data

Models

At its simplest a Model is just a set of fields and their data. We're going to look at four of the principal parts of *Ext.data.Model* — *Fields*, *Proxies*, *Associations* and *Validations*.



Let's look at how we create a model now:

```
Ext.define('User', {  
    extend: 'Ext.data.Model',  
    config: {  
        fields: [  
            { name: 'id', type: 'int' },  
            { name: 'name', type: 'string' }  
        ]  
    }  
});
```

Using Proxies

Proxies are used by stores to handle the loading and saving of model data. There are two types of proxy: client and server. Examples of client proxies include Memory for storing data in the browser's memory and Local Storage which uses the HTML 5 local storage feature when available. Server proxies handle the marshaling of data to some remote server and examples include Ajax, JsonP, and Rest.

Proxies can be defined directly on a model, like so:

```
Ext.define('User', {
```

```

    extend: 'Ext.data.Model',

    config: {
        fields: ['id', 'name', 'age', 'gender'],
        proxy: {
            type: 'rest',
            url : 'data/users',
            reader: {
                type: 'json',
                root: 'users'
            }
        }
    }
});

// Uses the User Model's Proxy
Ext.create('Ext.data.Store', {
    model: 'User'
});

```

This helps in two ways. First, it's likely that every store that uses the User model will need to load its data the same way, so we avoid having to duplicate the proxy definition for each store. Second, we can now load and save model data without a store:

```

// Gives us a reference to the User class
var User = Ext.ModelMgr.getModel('User');

var ed = Ext.create('User', {
    name: 'Ed Spencer',
    age : 25
});

// We can save Ed directly without having to add him to a Store first because we
// configured a RestProxy this will automatically send a POST request to the url /users
ed.save({
    success: function(ed) {
        console.log("Saved Ed! His ID is "+ ed.getId());
    }
});

// Load User 1 and do something with it (performs a GET request to /users/1)
User.load(1, {
    success: function(user) {
        console.log("Loaded user 1: " + user.get('name'));
    }
});

```

There are also proxies that take advantage of the new capabilities of HTML5 - *LocalStorage* and *SessionStorage*. Although older browsers don't support these new HTML5 APIs, they're so useful that a lot of applications will benefit enormously by using them.

Associations

Models can be linked together with the Associations API. Most applications deal with many different models, and the models are almost always related. A blog authoring application might have models for

User, Post, and Comment. Each user creates posts and each post receives comments. We can express those relationships like so:

```
Ext.define('User', {
  extend: 'Ext.data.Model',
  config: {
    fields: ['id', 'name'],
    proxy: {
      type: 'rest',
      url: 'data/users',
      reader: {
        type: 'json',
        root: 'users'
      }
    }
  },

  hasMany: 'Post' // shorthand for { model: 'Post', name: 'posts' }
});

Ext.define('Comment', {
  extend: 'Ext.data.Model',

  config: {
    fields: ['id', 'post_id', 'name', 'message'],
    belongsTo: 'Post'
  }
});

Ext.define('Post', {
  extend: 'Ext.data.Model',

  config: {
    fields: ['id', 'user_id', 'title', 'body'],

    proxy: {
      type: 'rest',
      url: 'data/posts',
      reader: {
        type: 'json',
        root: 'posts'
      }
    },

    belongsTo: 'User',
    hasMany: { model: 'Comment', name: 'comments' }
  }
});
```

It's easy to express rich relationships between different models in your application. Each model can have any number of associations with other models and your models can be defined in any order. Once we have a model instance we can easily traverse the associated data. For example, to log all comments made on each post for a given user, do something like this:

```
// Loads User with ID 1 and related posts and comments using User's Proxy
User.load(1, {
  success: function(user) {
```

```

    console.log("User: " + user.get('name'));

    user.posts().each(function(post) {
        console.log("Comments for post: " + post.get('title'));

        post.comments().each(function(comment) {
            console.log(comment.get('message'));
        });
    });
}
});

```

Each of the hasMany associations we created above adds a new function to the Model. We declared that each User model hasMany Posts, which added the user.posts() function we used in the snippet above. Calling user.posts() returns a Store configured with the Post model. In turn, the Post model gets a comments() function because of the hasMany Comments association we set up.

Associations aren't just helpful for loading data, they're useful for creating new records too:

```

user.posts().add({
    title: 'Ext JS 4.0 MVC Architecture',
    body: 'It's a great Idea to structure your Ext JS Applications using the built in MVC
Architecture...'
});

user.posts().sync();

```

Here we instantiate a new Post, which is automatically given the User id in the user_id field. Calling sync() saves the new Post via its configured proxy. This, again, is an asynchronous operation to which you can pass a callback if you want to be notified when the operation completed.

The belongsTo association also generates new methods on the model. Here's how to use that:

```

// get the user reference from the post's belongsTo association
post.getUser(function(user) {
    console.log('Just got the user reference from the post: ' + user.get('name'))
});

// try to change the post's user
post.setUser(100, {
    callback: function(product, operation) {
        if (operation.wasSuccessful()) {
            console.log('Post's user was updated');
        } else {
            console.log('Post's user could not be updated');
        }
    }
});

```

Once more, the loading function (getUser) is asynchronous and requires a callback function to get at the user instance. The setUser method simply updates the foreign_key (user_id in this case) to 100 and saves the Post model. As usual, callbacks can be passed in that will be triggered when the save operation has completed, whether successfully or not.

Validations

Models have rich support for validating their data. To demonstrate this we're going to build upon the example we created that illustrated associations. First, let's add some validations to the User model:

```
Ext.define('User', {
    extend: 'Ext.data.Model',

    config: {
        fields: ...,

        validations: [
            { type: 'presence', field: 'name' },
            { type: 'length', field: 'name', min: 5 },
            { type: 'format', field: 'age', matcher: /\d+/ },
            { type: 'inclusion', field: 'gender', list: ['male', 'female'] },
            { type: 'exclusion', field: 'name', list: ['admin'] }
        ],

        proxy: ...
    }
});
```

Validations follow the same format as field definitions. In each case, we specify a field and a type of validation. The validations in our example are expecting the name field to be present and to be at least five characters in length, the age field to be a number, the gender field to be either "male" or "female", and the username to be anything but "admin". Some validations take additional optional configuration - for example the length validation can take min and max properties, format can take a matcher, etc. There are five validations built into Sencha Touch 2, and adding custom rules is easy. First, let's look at the ones built right in:

- presence simply ensures that the field has a value. Zero counts as a valid value but empty strings do not.
- length ensures that a string is between a minimum and maximum length. Both constraints are optional.
- format ensures that a string matches a regular expression format. In the example above we ensure that the age field is four numbers followed by at least one letter.
- inclusion ensures that a value is within a specific set of values (for example, ensuring gender is either male or female).
- exclusion ensures that a value is not one of the specific set of values (for example, blacklisting usernames like 'admin').

Now that we have a grasp of what the different validations do, let's try using them against a User instance. We'll create a user and run the validations against it, noting any failures:

```
// now lets try to create a new user with as many validation errors as we can
var newUser = Ext.create('User', {
    name: 'admin',
    age: 'twenty-nine',
    gender: 'not a valid gender'
});

// run some validation on the new user we just created
var errors = newUser.validate();
```

```

console.log('Is User valid?', errors.isValid()); // returns 'false' as there were validation errors
console.log('All Errors:', errors.items); // returns the array of all errors found on this model
instance

console.log('Age Errors:', errors.getByField('age')); // returns the errors for the age field

```

The key function here is `validate()`, which runs all of the configured validations and returns an `Errors` object. This simple object is just a collection of any errors that were found, plus some convenience methods such as `isValid()`, which returns `true` if there were no errors on any field, and `getByField()`, which returns all errors for a given field.

Stores

Models are typically used with a `Store`, which is basically a collection of model instances. Setting up a store and loading its data is simple:

```

Ext.create('Ext.data.Store', {
    model: 'User',
    proxy: {
        type: 'ajax',
        url: 'users.json',
        reader: 'json'
    },
    autoLoad: true
});

```

We configured our store to use an Ajax Proxy, providing the name of the URL from which to load data the Reader used to decode the data. In this case our server is returning JSON, so we've set up a `Json` Reader to read the response. The store auto-loads a set of `User` model instances from the URL `users.json`. The `users.json` URL should return a JSON string that looks something like this:

```

{
    success: true,
    users: [
        { id: 1, name: 'Ed' },
        { id: 2, name: 'Tommy' }
    ]
}

```

Inline data

Stores can also load data inline. Internally, `Store` converts each of the objects we pass in as data into `Model` instances:

```

Ext.create('Ext.data.Store', {
    model: 'User',
    data: [
        { firstName: 'Ed', lastName: 'Spencer' },
        { firstName: 'Tommy', lastName: 'Maintz' },
        { firstName: 'Aaron', lastName: 'Conran' },
        { firstName: 'Jamie', lastName: 'Avins' }
    ]
});

```



```
    ]
  });
```

Sorting and Grouping

Stores are able to perform sorting, filtering, and grouping locally, as well as to support remote sorting, filtering, and grouping:

```
Ext.create('Ext.data.Store', {
  model: 'User',

  sorters: ['name', 'id'],
  filters: {
    property: 'name',
    value    : 'Ed'
  },
  groupField: 'age',
  groupDir: 'DESC'
});
```

In the store we just created, the data will be sorted first by name then id; it will be filtered to only include users with the name Ed, and the data will be grouped by age in descending order. It's easy to change the sorting, filtering, and grouping at any time through the Store API.

Proxies

Proxies are used by stores to handle the loading and saving of model data. There are two types of proxy: client and server. Examples of client proxies include Memory for storing data in the browser's memory and Local Storage which uses the HTML 5 local storage feature when available. Server proxies handle the marshaling of data to some remote server and examples include Ajax, JsonP, and Rest.

Proxies can be defined directly on a model, like so:

```
Ext.define('User', {
  extend: 'Ext.data.Model',

  config: {
    fields: ['id', 'name', 'age', 'gender'],
    proxy: {
      type: 'rest',
      url : '/data/users.json',
      reader: {
        type: 'json',
        root: 'users'
      }
    }
  }
});

// Uses the User Model's Proxy
Ext.create('Ext.data.Store', {
  model: 'User'
});
```

This helps in two ways. First, it's likely that every store that uses the User model will need to load its data the same way, so we avoid having to duplicate the proxy definition for each store. Second, we can now load and save model data without a store:

```
// Gives us a reference to the User class
var User = Ext.ModelMgr.getModel('User');

var ed = Ext.create('User', {
    name: 'Ed Spencer',
    age : 25
});

// We can save Ed directly without having to add him to a Store first because we
// configured a RestProxy this will automatically send a POST request to the url /users
ed.save({
    success: function(ed) {
        console.log("Saved Ed! His ID is "+ ed.getId());
    }
});

// Load User 1 and do something with it (performs a GET request to /users/1)
User.load(1, {
    success: function(user) {
        console.log("Loaded user 1: " + user.get('name'));
    }
});
```

There are also proxies that take advantage of the new capabilities of HTML5 - LocalStorage and SessionStorage. Although older browsers don't support these new HTML5 APIs, they're so useful that a lot of applications will benefit enormously by using them.

Device APIs

Camera

This class allows you to use native APIs to take photos using the device camera.

When this singleton is instantiated, it will automatically select the correct implementation depending on the current device:

- Sencha Packager
- PhoneGap
- Simulator

Both the Sencha Packager and PhoneGap implementations will use the native camera functionality to take or select a photo. The Simulator implementation will simply return fake images.

You can use the capture function to take a photo:

```
Ext.device.Camera.capture({
    success: function(image) {
        imageView.setSrc(image);
    },
    quality: 75,
    width: 200,
    height: 200,
    destination: 'data'});
```

Connection

This class is used to check if the current device is currently online or not. It has three different implementations:

- Sencha Packager
- PhoneGap
- Simulator

Both the Sencha Packager and PhoneGap implementations will use the native functionality to determine if the current device is online. The Simulator version will simply use navigator.onLine.

When this singleton (*Ext.device.Connection*) is instantiated, it will automatically decide which version to use based on the current platform.

Determining if the current device is online:

```
alert(Ext.device.Connection.isOnline());
```

Checking the type of connection the device has:

```
alert('Your connection type is: ' + Ext.device.Connection.getType());
```

The available connection types are:

- UNKNOWN - Unknown connection
- ETHERNET - Ethernet connection
- WIFI - WiFi connection
- CELL_2G - Cell 2G connection
- CELL_3G - Cell 3G connection
- CELL_4G - Cell 4G connection
- NONE - No network connection

Geolocation

Provides access to the native Geolocation API when running on a device. There are three implementations of this API:

- Sencha Packager
- PhoneGap
- Browser

This class will automatically select the correct implementation depending on the device your application is running on.

Getting the current location:

```
Ext.device.Geolocation.getCurrentPosition({
    success: function(position) {
        console.log(position.coords);
    },
    failure: function() {
        console.log('something went wrong!');
    }
});
```

Watching the current location:

```
Ext.device.Geolocation.watchPosition({
    frequency: 3000, // Update every 3 seconds
    callback: function(position) {
        console.log('Position updated!', position.coords);
    },
    failure: function() {
        console.log('something went wrong!');
    }
});
```

Notification

It provides a cross device way to show notifications. There are 3 different implementations:

- Sencha Packager
- PhoneGap

- Simulator

When this singleton is instantiated, it will automatically use the correct implementation depending on the current device.

Both the Sencha Packager and PhoneGap versions will use the native implementations to display the notification. The Simulator implementation will use Ext.MessageBox for show and a simply animation when you call vibrate.

To show a simple notification:

```
Ext.device.Notification.show({
    title: 'Verification',
    message: 'Is your email address is: test@sencha.com',
    buttons: Ext.MessageBox.OKCANCEL,
    callback: function(button) {
        if (button == "ok") {
            console.log('Verified');
        } else {
            console.log('Nope.');
```

To make the device vibrate:

```
Ext.device.Notification.vibrate();
```

Orientation

This class provides you with a cross platform way of listening to when the orientation changes on the device your application is running on.

The orientation change event gets passes the alpha, beta and gamma values.

You can find more information about these values and how to use them on the W3C device orientation specification.

To listen to the device orientation, you can do the following:

```
Ext.device.Orientation.on({
    scope: this,
    orientationchange: function(e) {
        console.log('Alpha: ', e.alpha);
        console.log('Beta: ', e.beta);
        console.log('Gamma: ', e.gamma);
    }
});
```

Push

Provides a way to send push notifications to a device. Currently only available on iOS.

```
Ext.device.Push.register({
    type: Ext.device.Push.ALERT|Ext.device.Push.BADGE|Ext.device.Push.SOUND,
```

```
success: function(token) {  
    console.log('# Push notification registration successful:');  
    console.log('  token: ' + token);  
},  
failure: function(error) {  
    console.log('# Push notification registration unsuccessful:');  
    console.log('  error: ' + error);  
},  
received: function(notifications) {  
    console.log('# Push notification received:');  
    console.log('  ' + JSON.stringify(notifications));  
}  
});
```

----- END OF BOOK -----

HTML5 Programming Coursebook, FPT Software, 2012

Mobile Programming

FSB.FMC