What’s new in HTML5

- Canvas
- Drag & drop
- History
- Inline editing
- Cross-document messaging
- Offline Apps
- Video & audio
Drawing on a Web page

- Possible just very recently
- SVG and canvas
  - Provide native drawing functionality on the Web
  - Completely integrated into HTML5 documents (part of DOM)
  - Can be styled with CSS
  - Can be controlled with JavaScript

Canvas

- HTML5 element and plugin-free 2D drawing API that enables to dynamically generate and render
  - Graphics
  - Charts
  - Images
  - Animation
- Canvas was originally introduced by Apple to be used in Mac OS
Canvas

- HTML5 element and plugin-free 2D drawing API that enables to dynamically generate and render graphics, charts, images, animation
- Scriptable bitmap canvas
  - Images that are drawn are final and cannot be resized
  - No object tree
  - Can be manipulated with JavaScript and styled with CSS
  - 2D Context
  - 3D Context (Web GL)
- Canvas was originally introduced by Apple to be used in Mac OS

Canvas

- The HTML5 canvas element uses JavaScript to draw graphics on a web page

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;canvas&gt;</td>
<td>For making graphics with a script</td>
</tr>
</tbody>
</table>

- A canvas is a rectangular area, where is possible to control every pixel
- The canvas element has several methods for drawing paths, boxes, circles, characters, and adding images

```html
<canvas id="myCanvas" width="200" height="100">
</canvas>
```
Browser support for canvas

- Mozilla Firefox 3.6.13
  - Canvas: 20/20
    - canvas element: Yes ✓
    - 2D context: Yes ✓
    - Text: Yes ✓
  - WebGL: 0/10
    - 3D context: No X

- Opera
  - Canvas: 20/20
    - canvas element: Yes ✓
    - 2D context: Yes ✓
    - Text: Yes ✓
  - WebGL: 0/10
    - 3D context: No X

- Support for 3D canvas (webGL): Chrome 5.0+

Example 1
Basic shape

- All drawing must be done in JavaScript

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
  Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
  var c=document.getElementById("myCanvas");
  var ctx=c.getContext("2d");
  ctx.fillStyle="#FF0000";
  ctx.fillRect(0,0,150,75);
</script>
</body>
</html>
```

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

- The fillStyle property can be a CSS color, a pattern, or a gradient; the default fillStyle is solid black.
- fillRect(x, y, width, height) draws a rectangle filled with the current fill style.
- The strokeStyle property is like fillStyle: it can be a CSS color, a pattern, or a gradient.
- strokeRect(x, y, width, height) draws an rectangle with the current stroke style; strokeRect doesn’t fill in the middle, it just draws the edges.
- clearRect(x, y, width, height) clears the pixels in the specified rectangle.
- moveTo(x, y) moves the pencil to the specified starting point.
- lineTo(x, y) draws a line to the specified ending point.
- The arc() method takes a center point (x, y), a radius, a start and end angle (in radians), and a direction flag (false for clockwise, true for counter-clockwise); the Math module that’s built into JavaScript can help to calculate radians.

- The font can be anything that fit in a CSS font rule. That includes font style, font variant, font weight, font size, line height, and font family.
- textAlign controls text alignment. It is similar to a CSS text-align rule; Possible values are start, end, left, right, and center.
- textBaseline controls where the text is drawn relative to the starting point; possible values are top, hanging, middle, alphabetic, ideographic, or bottom.
- The fillText() method draws the actual text.
- createLinearGradient(x0, y0, x1, y1) paints along a line from (x0, y0) to (x1, y1).
- createRadialGradient(x0, y0, r0, x1, y1, r1) paints along a cone between two circles. The first three parameters represent the start circle, with origin (x0, y0) and radius r0. The last three parameters represent the end circle, with origin (x1, y1) and radius r1.
Example 2
Path

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
    Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
    var c=document.getElementById("myCanvas");
    var ctx=c.getContext("2d");
    ctx.beginPath();
    ctx.moveTo(10,10);
    ctx.lineTo(150,50);
    ctx.lineTo(10,50);
    ctx.stroke();
</script>
</body>
</html>
```

Example 3
Circle

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
    Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
    var c=document.getElementById("myCanvas");
    var ctx=c.getContext("2d");
    ctx.fillStyle="#FF0000";
    ctx.beginPath();
    ctx.arc(70,18,15,0,Math.PI*2,true);
    ctx.closePath();
    ctx.fill();
</script>
</body>
</html>
```
Example 4
Gradients

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
    Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
    var c=document.getElementById("myCanvas");
    var ctx=c.getContext("2d");
    var grd=ctx.createLinearGradient(0,0,175,50);
    grd.addColorStop(0,"#FF0000");
    grd.addColorStop(1,"#00FF00");
    ctx.fillStyle=grd;
    ctx.fillRect(0,0,175,50);
</script>
</body>
</html>
```

Example 5
Image

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
    Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
    var c=document.getElementById("myCanvas");
    var ctx=c.getContext("2d");
    var img=new Image();
    img.src="img_flwr.png"
    ctx.drawImage(img,0,0);
</script>
</body>
</html>
```
Example 6

Text

```html
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">
    Your browser does not support the canvas element.
</canvas>
<script type="text/javascript">
    var c=document.getElementById("myCanvas");
    var cxt=c.getContext("2d");
    cxt.font = "bold 12px sans-serif";
    cxt.fillText("<canvas> is sweet!", 40, 50);
</script>
</body>
</html>
```

Pixel-based manipulation

- The 2D Context API provides three methods that help you draw pixel-by-pixel
  - `createImageData`
  - `getImageData`
  - `putImageData`
- Raw pixels are held in objects of type `ImageData`
- Each object has three properties: width, height and data
- The data property is of type `CanvasPixelArray`, holding a number of elements equal to `width*height*4`
  - For every pixel you can define the red, green, blue and alpha values of all the pixels, in the order you want them to appear (all the values range from 0 to 255, including alpha)
  - Pixels are ordered left to right, row by row, from top to bottom
Example 7 – Pixels

```html
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
window.addEventListener('load', function () {
    var elem = document.getElementById('myCanvas');
    var context = elem.getContext('2d');
    var imgd = false, w = 50, h = 50, x = 0, y = 0;
    imgd = context.createImageData(w, h);
    var pix = imgd.data;
    // Loop over each pixel.
    for (var i = 0, n = pix.length; i < n; i += 4) {
        pix[i] = 255; // the red channel
        pix[i+3] = 127; // the alpha channel
    }
    // Draw the ImageData object.
    context.putImageData(imgd, x, y);
});
</script>
</head>
<body>
<p><canvas id="myCanvas" width="100" height="100">Your browser does not have support for Canvas.</canvas></p>
</body>
</html>
```

Example 7 – Pixels

```html
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
window.addEventListener('load', function () {
    var elem = document.getElementById('myCanvas');
    var context = elem.getContext('2d');
    // Create a new image.
    var img = new Image();
    // Once it's loaded draw the image on canvas and
    // invert the colors.
    img.addEventListener('load', function () {
        var x = 0, y = 0;
        // Draw the image on canvas.
        context.drawImage(this, x, y);
        // Get the pixels.
        var imgd = context.getImageData(x, y, this.width, this.height);
        var pix = imgd.data;
        ...
    });
});
</script>
</head>
<body>
<p><canvas id="myCanvas" width="100" height="100">Your browser does not have support for Canvas.</canvas></p>
</body>
</html>
```
Example 7 – Pixels

```javascript
... // Loop over each pixel and invert the color.
for (var i = 0, n = pix.length; i < n; i += 4) {
    pix[i] = 255 - pix[i]; // red
    pix[i+1] = 255 - pix[i+1]; // green
    pix[i+2] = 255 - pix[i+2]; // blue
    // i+3 is alpha (the fourth element)
}

// Draw the ImageData object.
context.putImageData(imgd, x, y);
}, false);
```

Event listener

- `addEventListener(eventType, listener, useCapture)`
  - method that associates a function with a particular event and binds the event to the current node
- 3 parameters
  - `eventType`: a string representing the event to bind, without the “on” prefix (e.g. “click”, “mousedown”, …)
  - `listener`: the object or function to fire when the event fires. The actual parameter entered should be a reference to the function or object (i.e. “dothis” instead of “dothis()”)
  - `useCapture`: a Boolean value. If true, the node listens for the event type only while the event propagates toward the target node (in event capture node). If false, the node listens only when the event bubbles outward from the event target. If the current node is the target of the event, either Boolean value may be used
- The advantage of using the DOM to bind an event is that you can assign multiple functions to a node for the same event (i.e: `window.onload`) without running into event handler conflicts
Canvas and SVG: comparison

- Advantages

<table>
<thead>
<tr>
<th>Canvas</th>
<th>SVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance 2D surface for drawing anything you want.</td>
<td>Resolution independence – this makes SVG better suited for cross-platform user interfaces because it allows scaling for any screen resolution.</td>
</tr>
<tr>
<td>Constant performance — everything is a pixel. Performance only degrades when the image resolution increases.</td>
<td>SVG has very good support for animations. Elements can be animated using a declarative syntax, or via JavaScript.</td>
</tr>
<tr>
<td>You can save the resulting image as a .png or .jpg.</td>
<td>You have full control over each element using the SVG DOM API in JavaScript.</td>
</tr>
<tr>
<td>Best suited for generating raster graphics (for example in games, fractals, etc.), editing of images, and operations requiring pixel-level manipulation.</td>
<td>SVG is an XML file format, which means that depending on each Web browser implementation the accessibility of SVG documents can be much better than that of canvas elements. This makes SVG a better solution for Web application user interfaces. Even if SVG provides mostly presentational markup, the semantics of the user interface can be improved with ARIA attributes.</td>
</tr>
</tbody>
</table>

- Disadvantages

<table>
<thead>
<tr>
<th>Canvas</th>
<th>SVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are no DOM nodes for anything you draw. It is all pixels.</td>
<td>Slow rendering when document complexity increases – anything that uses the DOM a lot will be slow.</td>
</tr>
<tr>
<td>There's no API for animation. You have to resort to timers and other events to update the Canvas when needed.</td>
<td>SVG might not be suited by itself for applications like games. Perhaps the best choice would be a Canvas + SVG combination.</td>
</tr>
<tr>
<td>Poor text rendering capabilities.</td>
<td></td>
</tr>
<tr>
<td>Might not be the best choice for cases where accessibility is crucial. Canvas gives you a surface to draw onto with the API of the content you choose. Inherently, this means it is all pixels — unless some future API will define additional capabilities for accessibility. For now, you can provide fallback content inside the canvas element that is displayed by the Web browser when the element itself cannot be rendered. Additionally, you can perform checks with JavaScript to see if the desired Canvas API is available for use. Based on that you can provide different functionality for users of Web browsers that lack canvas support.</td>
<td></td>
</tr>
<tr>
<td>Canvas is not suited for Web site or application user Interfaces. This is because user interfaces typically need to be dynamic and interactive, and Canvas requires you to manually redraw each element in the Interface. Other reasons would be the lack of animation and accessibility support.</td>
<td></td>
</tr>
</tbody>
</table>
WebGL

- WebGL is a standard for programming in 3D with the browser as platform
  - It is an interface between JavaScript and OpenGL ES 2.0, a library in native code that directly accesses the hardware of graphics cards
- WebGL runs on a browser supporting Canvas (Firefox, Chrome and Safari) and with an appropriate plugin
- Web browsers that support WebGL
  - Mozilla Firefox 4 Beta, Google Chrome Beta or Chromium, WebKit nightly
- WebGL cheat sheet
- Example: Google body browser
  - http://bodybrowser.googlelabs.com/

Drag & drop

- HTML 5 includes a Drag and Drop API that brings native drag&drop support to the browser
- Drag & drop requires
  - Something to drag
  - A drop target
  - JavaScript event handlers
- See
  - https://developer.mozilla.org/En/DragDrop
Drag events

- A number of events are fired during various stages of the drag and drop operation
  - Only drag events are fired: mouse events such as mousemove are not fired during a drag operation
- The dataTransfer property of all drag events holds data about the drag and drop operation
- dragstart
  - Fired on an element when a drag is started
  - The user is requesting to drag the element the dragstart event is fired at
  - During this event, a listener would set information such the drag data and image to be associated with the drag
Drag events

- **dragenter**
  - Fired when the mouse is first moved over an element while a drag is occurring
  - A listener for this event should indicate whether a drop is allowed over this location
  - If there are no listeners, or the listeners perform no operations, then a drop is not allowed by default
  - This is also the event to listen to if you want to provide feedback that a drop is allowed such as displaying a highlight or insertion marker

- **dragover**
  - This event is fired as the mouse is moved over an element when a drag is occurring
  - Much of the time, the operation that occurs during a listener will be the same as the dragenter event

- **dragleave**
  - This event is fired when the mouse leaves an element while a drag is occurring
  - Listeners should remove any highlighting or insertion markers used for drop feedback

- **drag**
  - This event is fired at the source of the drag, that is, the element where dragstart was fired, during the drag operation
Drag events

- **drop**
  - This event is fired on the element where the drop occurred at the end of the drag operation
  - A listener would be responsible for retrieving the data being dragged and inserting it at the drop location
  - This event will only fire if a drop is desired
  - It will not fire if the user cancelled the drag operation, for example by pressing the Escape key, or if the mouse button was released while the mouse was not over a valid drop target

- **dragend**
  - The source of the drag will receive a dragend event when the drag operation is complete, whether it was successful or not

Something to drag

- In HTML the `<img>` elements and the `<a>` elements (with an href) are draggable by default
- In order to make another HTML element draggable, two things must be done
  - Set the draggable attribute to true on the element that you wish to make draggable
  - Add a listener for the dragstart event and set the drag data within this listener

```html
<div draggable="true"
ondragstart="event.dataTransfer.setData('text/plain', 'This text may be dragged')">
  This text <strong>may</strong> be dragged. </div>
```

- Within the dragstart event, you can specify the drag data, the feedback image and the drag effects
  - Only drag data is required
Drag data

- All drag events have a property called `dataTransfer` which is used to hold the drag data
- Information to be provided
  - The data to be dragged
  - The drag feedback image which appears beside the mouse pointer during the drag operation; this image may be customized but most of the time it is not specified, and a default image is generated
  - Drag effects that are allowed

When a drag occurs, data must be associated with the drag which identifies what is being dragged
- Example: when dragging the selected text within a textbox, the data associated with the drag is the text itself; when dragging a link on a web page, the drag data is the URL of the link

The drag data contains two pieces of information
- the type (or format, or data): a type string (e.g. `text/plain` for text data): types are a MIME-type like string, such as `text/plain` or `image/jpeg`
- the data value: a string of text

When the drag begins, you add data by providing a type and the data
- During a drop event, a listener would retrieve the data being dragged and insert it at the drop location
- To set data within the `dataTransfer`: `setData` method

```javascript
event.dataTransfer.setData("text/plain", "Text to drag");
```
Drag feedback

- When a drag occurs, a translucent image is generated from the drag target and follows the mouse pointer during the drag
- This image is created automatically
- However, it is possible to use the setDragImage to specify a custom drag feedback image

```javascript
event.dataTransfer.setDragImage(image, xOffset, yOffset);
```

Drag effects

- The copy operation is used to indicate that the data being dragged will be copied from its present location to the drop location
- The move operation is used to indicate that the data being dragged will be moved
- The link operation is used to indicate that some form of relationship or connection will be created between the source and drop locations
- To specify which of the three operations are allowed for a drag source: set the effectAllowed property within a dragstart event listener

```javascript
event.dataTransfer.effectAllowed = "copy";
```

- Legal values are: “none”, “copy”, “move”, “link”, “copyMove”, “copyLink”, “linkMove”, “all" (default)
Specifying drop targets

- A listener for the dragenter and dragover events are used to indicate valid drop targets, that is, places where dragged items may be dropped
- Most areas of a web page or application are not valid places to drop data
  - The default handling for these events is to not allow a drop
- If you want to allow a drop, you must prevent the default handling by cancelling the event
- You can do this either by returning false from an attribute-defined event listener, or by calling the event’s event.preventDefault method
  - The latter may be more feasible in a function defined in a separate script

```html
<div ondragover="return false">
  <div ondragover="event.preventDefault()">
```

Specifying drop targets

- It is most common to accept or reject a drop based on the type of drag data in the data transfer
  - For instance, allowing images or links or both
- To do this, you can check the types of the dataTransfer object

```javascript
function doDragOver(event) {
  var isLink = event.dataTransfer.types.contains("text/uri-list");
  if (isLink)
    event.preventDefault();
}
```

- The contains method checks if the type text/uri-list is present in the list of types
  - If it is, we will cancel the event so that a drop may be allowed
  - If the drag data does not contain a link, the event will not be cancelled and a drop cannot occur at that location
Drop feedback

- There are several ways to indicate to the user that a drop is allowed at a certain location
  - The mouse pointer will update as necessary depending on the value of the dropEffect property (The actual effect that will be used, should always be one of the possible values of effectAllowed)
  - Although the exact appearance depends on the user’s platform, typically a plus sign icon will appear for a 'copy' for example, and a 'cannot drop here' icon will appear when a drop is not allowed
  - This mouse pointer feedback is sufficient in many cases
- You can also update the user interface with an insertion point or highlight as needed
  
  ```
  .droparea:~moz-drag-over {
    border: 1px solid black;
  }
  ```

- The element with the class droparea will receive a 1 pixel black border while it is a valid drop target, that is, if the event.preventDefault method was called during the dragenter event
- Note that you must cancel the dragenter event for this pseudoclass to apply, as this state is not checked for the dragover event

For more complex visual effects, you can also perform other operations during the dragenter event, e.g. by inserting an element at the location where the drop will occur
- This might be an insertion marker or an element that represents the dragged element in its new location
- You could create an image or separator element for example, and simply insert it into the document during the dragenter event
- The dragover event will fire at the element the mouse is pointing
  - You may need to move the insertion marker around a dragover event as well
  - You can use the event's clientX and clientY properties as with other mouse events to determine the location of the mouse pointer
- The dragleave event will fire at an element when the drag leaves the element
  - This is the time when you should remove any insertion markers or highlighting
Performing a drop

- When the user releases the mouse, the drag and drop operation ends
  - If the mouse is released over an element that is a valid drop target (i.e. one that cancelled the last dragenter or dragover event) then the drop will be successful, and a drop event will fire at the target
  - Otherwise, the drag operation is cancelled and no drop event is fired
- During the drop event, you should retrieve that data that was dropped from the event and insert it at the drop location
- The getData method may be used to retrieve the data from dataTransfer property
- The getData method takes one argument, the type of data to retrieve and returns the string value that was set when the setData was called at the beginning of the drag operation
  - An empty string will be returned if data of that type does not exist

Here, once we have retrieved the data, we insert the string as the textual content of the target
- This has the effect of inserting the dragged text where it was dropped, assuming that the drop target is an area of text such as a p or div element
- In a web page, you should call the preventDefault method of the event if you have accepted the drop so that the default browser handling does not handle the dropped data as well
  - For example, when a link is dragged to a web page, Firefox will open the link
  - By cancelling the event, this behaviour will be prevented

```javascript
function onDrop(event) {
  var data = event.dataTransfer.getData("text/plain");
  event.target.textContent = data;
  event.preventDefault(); }
```
Performing a drop

```javascript
function doDrop(event) {
    var links =
        event.dataTransfer.getData("text/uri-list").split("\n");
    for each (var link in links) {
        if (link.indexOf("#") == 0) continue;
        var newlink = document.createElement("a");
        newlink.href = link;
        newlink.textContent = link;
        event.target.appendChild(newlink);
    }
    event.preventDefault(); }
```

- The `text/uri-list` type actually may contain a list of URLs, each on a separate line
  - In this code, we use the `split` to split the string into lines, then iterate over the list of lines, inserting each as a link into the document.
  - Note also that we skip links starting with a number sign (`#`) as these are comments

Performing a drop

- For simple cases, the special type `URL` to just retrieve the first valid URL in the list

```javascript
var link = event.dataTransfer.getData("URL");
```
Finishing a drag

- Once the drag is complete, a dragend is fired at the source of the drag (the same element that received the dragstart event)
  - This event will fire if the drag was successful or if it was cancelled
- You can use the dropEffect to determine what drop operation occurred
  - If the dropEffect property has the value none during a dragend, then the drag was cancelled
  - Otherwise, the effect specifies which operation was performed
- A drop can occur inside the same window or over another application
- After the dragend event has finished propagating, the drag and drop operation is complete

The example: dragstart

```html
<div draggable="true" id="paper"
dragstart="drag(this, event)"></div>
<div draggable="true" id="coke_can"
dragstart="drag(this, event)"></div>
<div draggable="true" id="empty_wrapper"
dragstart="drag(this, event)"></div>
<div draggable="true" id="bottle"
dragstart="drag(this, event)"></div>
<div draggable="true" id="pencil_shavings"
dragstart="drag(this, event)"></div>
```

```javascript
function drag(drop_target, e) {
  e.dataTransfer.setData('Text', drop_target.id);
  document.getElementById('thanks').style.display="none";
}
```
The example: dragenter, dragover, drop

```javascript
function drop(drop_target, e) {
    count++;  
    var id = e.dataTransfer.getData('Text');
    document.getElementById('thanks').style.display = 'block';
    drop_target.appendChild(document.getElementById(id));
    document.getElementById(id).style.display = 'none';

    if(count==5) {
        document.getElementById('thanks').style.display = 'none';
        document.getElementById('done').style.display = 'block';
    }

    e.preventDefault();
}
```

Another example
History API

- The HTML5 history API is a standardized way to manipulate the browser history via script
- The session history is the sequence of documents in a browsing context
- The DOM window object provides access to the browser’s history through the window.history object
  - It exposes useful methods and properties that allow to move back and forth through the user’s history, as well as manipulate the contents of the history stack

Reasons to manipulate history

- What does a URL do? It identifies a unique resource
- Browsers have always had a fundamental limitation: if you change the URL, even through script, it triggers a roundtrip to the remote web server and a full page refresh
  - This takes time and resources, and it seems especially wasteful when you are navigating to a page that is substantially similar to the current page
  - Everything on the new page gets downloaded, even the parts that are exactly the same as the current page
  - There is no way tell a browser to change the URL but only download half a page
Reasons to manipulate history

- Example: you have two pages, page A and page B; the two pages are 90% identical
- The user navigates to page A, then tries to navigate to page B
- Instead of triggering a full page refresh, you interrupt this navigation and do the following steps manually:
  - Load the 10% of the page from page B that is different from page A (probably using XMLHttpRequest); this will require some server-side changes to your web application
  - Swap in the changed content (using innerHTML or other DOM methods)
  - Update the browser location bar with the URL of page B, using a method from the HTML5 history API

Example

- “Dive into dogs”
  - [http://diveintohtml5.org/examples/history/fer.html](http://diveintohtml5.org/examples/history/fer.html)
- Demonstrates a common pattern: a long article with an associated inline photo gallery
- In a supported browser, navigating the Next and Previous links in the photo gallery will update the photo in place and update the URL in the browser location bar, without triggering a full page refresh
- In unsupported browsers the links simply function as regular links, taking you to a new page with a full page refresh
Example

- The links are just regular `<a>` elements

```html
<aside id="gallery">
  <p class="photonav">
    <a id="photonext" href="casey.html">Next &gt;</a>
    <a id="photoprev" href="adagio.html">&lt; Previous</a>
  </p>
  <figure id="photo">
    <img id="photoimg" src="gallery/1972-fer-500.jpg" alt="Fer" width="500" height="375">
    <figcaption>Fer, 1972</figcaption>
  </figure>
</aside>
```

- The rest of the code that follows is behind a detection script: if the user is using an unsupported browser, no history API code will be executed

Modernizr is an open source, MIT-licensed JavaScript library that detects support for many HTML5 & CSS3 features

```javascript
if (Modernizr.history) {
  // history management works!
  // Code HERE...
} else {
  // no history support
}
```

Example

- The main driver function gets the links and passes it to a function, `addClicker()`

```javascript
function setupHistoryClicks() {
  addClicker(document.getElementById("photonext"));
  addClicker(document.getElementById("photoprev"));
}
```

- The `addClicker()` function takes an `<a>` element and adds a click handler

```javascript
function addClicker(link) {
  link.addEventListener("click", function(e) {
    swapPhoto(link.href);
    history.pushState(null, null, link.href);
    e.preventDefault();
  }, false);
}
```
Example

- The first half of the `swapPhoto()` function takes part of the URL of the navigation link itself (casey.html, adagio.html, ...) and constructs a URL to a hidden page that contains the markup required by the next photo.

```javascript
function swapPhoto(href) {
    var req = new XMLHttpRequest();
    req.open("GET", "http://diveintohtml5.org/examples/history/gallery/" + href.split("/").pop(), false);
    req.send(null);
    if (req.status == 200) {
        document.getElementById("gallery").innerHTML = req.responseText;
        setupHistoryClicks();
        return true;
    } return false;
}
```

- The second half inserts this newly downloaded markup into the current page.

Example

- The `history.pushState()` function takes three parameters:
  - State can be any JSON data structure; we don’t need to track any state in this demo, so I’ve left it as null.
  - Title can be any string.
  - URL can be, well, any URL: this is the URL you want to appear in the browser’s location bar.

```javascript
history.pushState(null, null, link.href);
```
Other history methods

- `window.history.back();`
  - to move backward through history
  - act exactly like the user clicked on the Back button in the browser toolbar
- `window.history.forward();`
  - to move forward
- `window.history.go(-2);`
  - move to a specific point in history
- `var numberOfEntries = window.history.length;`
  - to determine the number of pages in the history stack
- `history.replaceState()`
  - allows to modify history entries
  - operates exactly like `history.pushState()`

History

- Not widely supported yet

Mozilla Firefox 3.6.13

<table>
<thead>
<tr>
<th>User interaction</th>
<th>15/25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag and drop</td>
<td>Yes</td>
</tr>
<tr>
<td>Undo history</td>
<td>No</td>
</tr>
<tr>
<td>Session history</td>
<td>No</td>
</tr>
<tr>
<td>Text selection</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Opera 11.0

<table>
<thead>
<tr>
<th>User interaction</th>
<th>5/25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag and drop</td>
<td>No</td>
</tr>
<tr>
<td>Undo history</td>
<td>No</td>
</tr>
<tr>
<td>Session history</td>
<td>No</td>
</tr>
<tr>
<td>Text selection</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Inline editing

- Client-side in-browser “rich text” editing
  - Currently supported by all major browsers
- The “contenteditable” attribute allows for the editing of content within any valid HTML5 element
  - By default, elements implicitly inherit editability from their parent element unless explicitly defined
- Two attributes involved
  - The designMode attribute governs the entire document (i.e. it makes the entire document editable, like a dedicated HTML editor)
  - The contentEditable attribute governs just the element on which it appears, and that element’s children

Example:

```javascript
<script>
  document.getElementsByTagName('p')[0].contentEditable = true;
</script>
```
Inline editing

- The contentEditable attribute is an enumerated attribute with three states which map to four keywords:
  - **true**: it indicates that the element is editable; it is specified by the ("" empty string) or the true keywords
  - **false**: it indicates that the element is not editable; it is specified by the false keyword
  - **inherit**: it indicates that the element is editable if its immediate parent element is editable; it is specified by the inherit keyword

Example

```html
<head>
<style>
[contenteditable=true] {  
  outline: 5px dotted #cccccc;
}
[contenteditable=true]:hover {  
  outline: 5px dotted #ff0000;
}
</style>
</head>
<body>
<p contenteditable="true">This content is editable!</p>
<ul contenteditable="true">
  <li>By default, this content is editable via its inherited parent elements value.</li>
  <li contenteditable="false">This content is <b>not</b> editable via its explicit contenteditable value.</li>
</ul>
</body>
```
Cross-document messaging

- Cross-document messaging, or web messaging, is an API introduced in HTML5 draft specification, allowing documents to communicate with one another across different origins, or source domains.
- Prior to HTML5, web browsers disallowed cross-site scripting, to protect against security attacks.
- This practice barred communication between non-hostile pages as well, making document interaction of any kind difficult.
- Cross-Document messaging allows scripts to interact across these boundaries, while providing a rudimentary level of security.

Example

- Registers an event handler for incoming messages from other domains
  ```javascript
  window.addEventListener('message', function(e) {
    if (e.origin == 'http://example.com') {
      // you've verified the domain the message is coming from,
      // now do something
    }, false);
  }
  ```

- Sends a message to another domain
  ```javascript
  var frame = document.getElementById("someIframe").contentWindow;
  frame.postMessage("Some message", "http://www.example.com");
  ```
Example

- Document A

```javascript
var o = document.getElementsByTagName('iframe')[0];
o.contentWindow.postMessage('Hello B','http://documentB.com/');
```

- Document B

```javascript
window.addEventListener('message', receiver, false);

function receiver(event) {
  if (event.origin == 'http://documentA.com') {
    if (event.data == 'Hello B') {
      event.source.postMessage('Hello A, how are you?', event.origin);
    }
    else { alert(event.data); }
  }
}
```

Offline Web applications

- “In order to enable users to continue interacting with Web applications and documents even when their network connection is unavailable (...) authors can provide a manifest which lists the files that are needed for the Web application to work offline and which causes the user’s browser to keep a copy of the files for use offline.”
Offline Web applications

- At its simplest, an offline web application is a list of URLs (HTML, CSS, JavaScript, images, or any other kind of resource)
- The home page of the offline web application points to this list, called a manifest file, which is just a text file located elsewhere on the web server
- A web browser that implements HTML5 offline applications will read the list of URLs from the manifest file, download the resources, cache them locally, and automatically keep the local copies up to date as they change
- When the time comes that you try to access the web application without a network connection, your web browser will automatically switch over to the local copies instead

The cache manifest

- It is a list of all of the resources that your web application might need to access while it's disconnected from the network

```
CACHE MANIFEST
/clock.css
/clock.js
/clock-face.jpg
```

- The manifest must be referenced by every HTML file

```
<!DOCTYPE HTML>
<html manifest="/cache.manifest">
  <body>...
</body>
</html>
```
Cache manifest sections

- The cache manifest file can include section headers
- **CACHE:** is the beginning of the “explicit” section
  - Resources in the “explicit” section will get downloaded and cached locally, and will be used in place of their online counterparts whenever you are disconnected from the network
  - Can be omitted if there is only the explicit section
- **NETWORK:** is the beginning of the “online whitelist” section
  - Resources in this section are never cached and are not available offline

---

**CACHE MANIFEST**

```text
NETWORK:
/tracking.cgi
CACHE:
/clock.css
/clock.js
/clock-face.jpg
```

---

Cache manifest sections

- **FALLBACK:** is the beginning of the fallback section
  - In a fallback section, you can define substitutions for online resources that, for whatever reason, can’t be cached or weren’t cached successfully

---

**CACHE MANIFEST**

```text
index.html
stylesheet.css
images/masthead.png
scripts/misc.js

NETWORK:
search.php
login.php
/api

FALLBACK:
images/dynamic.php static_image.png
```
The flow of DOM events

- When your browser visits a page that points to a cache manifest, it fires off a series of events on the window.applicationCache object.
- As soon as it notices a manifest attribute on the <html> element, your browser fires a checking event.
  - The checking event is always fired, regardless of whether you have previously visited this page or any other page that points to the same cache manifest.
- If your browser has never seen this cache manifest before...
  - It will fire a downloading event, then start to download the resources listed in the cache manifest.
  - While it's downloading, your browser will periodically fire progress events, which contain information on how many files have been downloaded already and how many files are still queued to be downloaded.
  - After all resources listed in the cache manifest have been downloaded successfully, the browser fires one final event, cached; this is your signal that the offline web application is fully cached and ready to be used offline.

On the other hand, if you have previously visited this page or any other page that points to the same cache manifest, then your browser already knows about this cache manifest: it may already have some resources in the appcache or even the entire working offline web application.

- Has the cache manifest changed since the last time your browser checked it?
  - If the answer is no, the cache manifest has not changed, your browser will immediately fire a noupdate event.
  - If the answer is yes, the cache manifest has changed, your browser will fire a downloading event and start re-downloading every single resource listed in the cache manifest.
  - While it's downloading, your browser will periodically fire progress events, which contain information on how many files have been downloaded already and how many files are still queued to be downloaded.
  - After all resources listed in the cache manifest have been re-downloaded successfully, the browser fires one final event, updateready; this is your signal that the new version of your offline web application is fully cached and ready to be used offline.