Scalable User Interfaces
Using HTML5

dr. ir. Natalia Stash
Project goals:
* Scalable user interfaces that can run on PC, tablet (iPad, Android), smartphone, custom devices (e.g. Bosch)
On all these devices we wish to create one “look and feel”.
However each platform has different processing power, display capabilities etc. Therefore we would like to create a scalable user interface that can work on all platforms.
* Idea: to create a concept and show this on at least two of the targeted hardware platforms.
* Also impact on Web server will need to be examined
* Usability/Performance of the concept

The most important application: a family of user interfaces for Public Address Systems (or broadcasting systems).
This is a request from Bosch company (www.boschsecurity.com).
Bosch makes Public Address (PA) and Conference audio systems (microphones, transmission, headsets, interpreter support facilities).

In this workshop we explain the basic concepts and look at specific options that HTML5 can offer for the design and implementation of scalable interfaces.
For providing audio streaming we look into WebRTC technology.
We provide hands-on experience with these technologies in some well-chosen examples.

Learning:
* Scalable UI design: “One design for all platforms”
* Cross platform development: abstraction from platform dependency
* Regression testing
* (Platform dependency: auto-detection of the platform and hardware properties, auto adjustment?)
Target Application: 
Public Address (PA) System

Usage of the PA System:
• For various types of announcement:
  − ad-hoc e.g. 
    *passengers lost, boarding starts, etc.*
  − scheduled/periodic e.g.
    *“Do not leave your luggage unattended”*
  − targeted (zones, buildings, ...) e.g.
    *selection of where you want to make an announcement - in one hall or in the whole building*
• Background music - *different music in different halls*
• Login procedures

Driving application: Public Address Systems - speaker systems at the stations, airports.
Assignment description from Bosch:

"The current PC Call Station Client is developed in C#, .NET and communicates with the PC Call Station server via .NET-remoting interface."

"The html5-based Call client can be developed in different iterations and needs to contain the following functionality

Functionality basic Call Station client:
* Select a predefined call (with predefined zones)
* Start/stop/abort the call and display call phase (start chime, messages, live speech, end chime)
* Optional : Transfer live speech from the client to the Praesideo system (via a dedicated audio server)"
BGM - background music.

* Functionality basic Bgm client:
  * Select a bgm channel and add/remove zones from the channel
  * Change the volume of a channel or zone (+ display bgm volumes)
  * Optional: Listen in on the selected bgm channel (via a monitor zone)

* Functionality advanced Call Station client:
  * Add/remove zones, zonegroups to a call
  * Zones should be presented/selected as polygons on a screen (for now this polygon information should be fixed in the client; polygon information is not provided by the server)
  * Provide zone(group) status feedback in polygon (using colors, edges, etc)
  * Navigate (or zoom in/out) to different layouts. Think of a complex site like a cruiseship with multiple decks, an airport with multiple terminals or a hotel with multiple floors. Some examples to show the complexity:
    * An addition is to combine the bgm channel selection and volumes (per zone) in the polygons

The provided test system will probably not contain enough zones (7) to properly develop/demonstrate a complex layout. But it is of course always possible to add ‘dummy’ zones in the client (that needs to define the polygon position information anyway)"
Organizational Matter

• October 17, afternoon, MF 6.132: HTML5/CSS3 tutorial
• October 29, afternoon, MF 12: WebRTC tutorial
• October 30, morning, MF 12: Hands-on experiments afternoon: Kick-off meeting with Bosch
• October 31, morning, MF 12: Hands-on experiments

Group formation by dr. Ad Aerts

• November-December: Work on the project, regular meetings with Bosch
Workshop Prerequisites

• (X)HTML (EXtensible HyperText Markup Language)
• CSS (Cascading Style Sheets)
• JavaScript

Check http://www.w3schools.com/
HTML5 is mostly about adding new programming features to the platform.

HTML5 has a major focus on facilitating use of a browser as a Web application platform (or Web application runtime environment).

HTML5 is also a potential candidate for cross-platform mobile applications.
Why HTML5?

- Forget about plug-ins
  - cost reduced
  - performance improved
  - better security

- Smarter local storage
  - better than cookies

- Better interactions
  - CSS3 effects, <canvas>, drag&drop, form validation

- Mobile, mobile, mobile
  - CSS3 media queries
  - mobile meta tags - viewport property
  - new input types in Web forms (we’ll see examples later)

http://tympanus.net/codrops/2011/11/24/top-10-reasons-to-use-html5-right-now/

* Audio and Video native HTML5 support

Prior to HTML5, there was no standards-based way to embed audio/video in a Web page. Virtually all the video you've ever watched "on the Web" has been funneled through a third-party plug-in — QuickTime, RealPlayer, Flash (YouTube e.g. uses Flash.)

Different browsers may have different plug-ins.

These plug-ins integrate with your browser well enough that you may not even be aware that you're using them. That is, until you try to watch a video on a platform that doesn't support that plug-in.

In HTML5 these are replaced with open standards.

Due to native support of features the cost of Web applications is reduced.

Apple claims Flash is enemy for batteries and mostly browsers crash due to Flash.

Thus, the HTML5-based Web applications are much better performance wise.

Besides, plug-ins also have security problems.

* Smarter storage

One of the coolest things about HTML5 is the new local storage feature. It's a little bit of a cross between regular old cookies and a client-side database. Could be useful e.g. for shopping card.

* Better interactions

We all want better interactions, we all want a more dynamic website that responds to the user and allows the user to enjoy/interact your content instead of just look at it.

Enter <canvas>, the drawing HTML5 tag that allows you to do most (if not more) interactive and animated possibilities than the previous rich internet application platforms like Flash.

Beyond <canvas>, HTML5 also comes with a slew of great APIs that allow you to build a better user experience and a beefier, more dynamic web application — here’s a quick list of native APIs:

- Drag and Drop (DnD)
- Offline storage database
- Browser history management
- Document editing
- Timed media playback

HTML5 allows for form validation for some new input types, before their validation was done with JavaScript. The validation happens before the inputs are submitted to the server, which makes it more reliable and speedy.

* Mobile, mobile, mobile

HTML5 is a potential candidate for cross-platform mobile applications.

Many features of HTML5 have been built with the consideration of being able to run on low-powered devices such as smartphones and tablets.

Mobile browsers have fully adopted HTML5 so creating mobile ready projects is as easy as designing and constructing for their smaller touch screen displays — hence the popularity of Responsive Design.

- CSS3 media queries allow to tweak the layout based on the screen size
- Mobile meta tags and viewport property: allow you to define viewport widths and zoom settings
- New input types in Web forms, e.g. the email input type looks just like a text box, but mobile browsers will customize their onscreen keyboard to make it easier to type email addresses.

For more information on how to mobilize your site via HTML5, check out "Mobifying" Your HTML5 Site (http://www.html5rocks.com/en/mobile/mobifying/)

Some resources worth checking out:

- http://www.mobilehtml5.com/
- Mobile HTML5
- Mobile Boilerplate
- HTML5 Mobile Web Applications
Why HTML5?
Mobile, Mobile, Mobile

- html5rocks.com on desktop & mobile
The first meta setting (and one that you'll use most often) is the viewport property. Setting a viewport tells the browser how content should fit on the device's screen and informs the browser that the site is optimized for mobile.

Viewport meta tag is generally used for responsive design to set the viewport width and initial-scale on mobile devices.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

The default viewport width on iPhone is 980px. But your design might not fit in that range. It could be wider or narrower. The slide shows two examples where you can utilize the viewport tag to improve the rendering of non-responsive design on mobile devices.

The screenshot on the left shows how the site would render without the viewport tag. As you can see, the page is touching on both sides. viewport tag was added to specify the viewport width to 1024px so it leaves some margin space on the left and right sides.

```
<meta name="viewport" content="width=1024">
```
HTML5 is still a work in progress. However, the major browsers support many of the new HTML5 elements and APIs.

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.

W3C specification:
http://www.w3.org/TR/html5/

WHATWG specification:
http://www.whatwg.org/specs/web-apps/current-work/multipage/index.html#contents
One of the principles of HTML5 - "Don't break the Web" which means that a standard shouldn't introduce changes that make other people's Web pages stop working. On the other hand, it would be nice if in one hundred years we can still read what we write today.
Does my browser support HTML5?
Detect support for individual features at http://html5test.com

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.

Does my browser support HTML5?
You cannot detect "HTML5 support" but you can detect support for individual features, like canvas, video, or geolocation.
http://html5test.com/
1. The **DOCTYPE**

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" 
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!doctype html>
```

The root element of an HTML page is always `<html>`.

In HTML5 there is no need for an ending tag.
2. Character encoding

The character encoding is the standard that tells a computer how to convert your text into a sequence of bytes when it's stored in a file (and how to convert it back again when the file is opened). For historical reasons, there are many different character encodings in the world. Today, virtually all English websites use an encoding called UTF-8, which is compact, fast, and supports all the non-English characters you'll ever need.

Often, the Web server that hosts your pages is configured to tell browsers that it's serving out pages with a certain kind of encoding. However, because you can't be sure that your Web browser will take this step (unless you own it), and because browsers can run into an obscure security issue when they attempt to guess a page's encoding, you should always add encoding information to your markup.

HTML5 makes it easy to do. All you need is add the meta element at the very beginning of your <head> section.

3. type="" is now optional

Because CSS is the only style sheet language around, there's no need to add the type="text/css" attribute that Web pages used to require.

There's no need to include type="text/javascript". The browser assumes you want JavaScript unless you specify otherwise.

4. Some deprecated elements from HTML 4.01 have been dropped, including purely presentational elements such as <font> and <center>, whose effects are achieved using CSS.
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <title>My Website</title>
  </head>
  <body>
    <p>Hello, World!</p>
  </body>
</html>
To better handle today's Internet use, HTML5 includes new elements for better structure, better form handling, drawing, and for media content.
A new feature being introduced in HTML 5 is the addition of custom data attributes. Simply, the specification for custom data attributes states that any attribute that starts with "data-" will be treated as a storage area for private data (private in the sense that the end user can't see it - it doesn't affect layout or presentation).

The advantage is that you can easily associate some scripting data with your elements without having to insert inline javascript all over the place, and it will be valid HTML5. To do the same thing in HTML4 would require specifying a custom namespace, and add some namespaced attributes.

http://wwwis.win.tue.nl/~nstash/html5/examples/example34.html

contenteditable
You can add 'contenteditable' attribute to any element to make the content of that element editable. Some browser support built-in commands for bold, italic, etc. formatting.
Editing an element does not change the in-memory content of the page. A more typical page would send this data to a Web server, probably using the XMLHttpRequest object.

HTML5: New Elements/Attributes in Web Forms

- Revamping a traditional HTML form
  - placeholder text *(also called watermark)*, displayed as long as input field is empty
  - autofocus attribute, specifies that an `<input>` or `<textarea>` element should automatically get focus when the page loads

- New input types:
  - email addresses, URLs, search boxes, telephone numbers, numbers, sliders, dates and times, colors

- New elements:
  - input suggestions with `<datalist>`

HTTP://WWWIS.WIN.TUE.NL/~NSTASH/HTML5/EXAMPLES/MUSEUMVISITORFORM.ORIGINAL.HTML
HTTP://WWWIS.WIN.TUE.NL/~NSTASH/HTML5/EXAMPLES/MUSEUMVISITORFORM.REVISED.HTML

HTML5 has several new input types for forms. These new features allow better input control and validation without requiring a tangle of JavaScript code or a JavaScript toolkit from another company.

Improvements that HTML5 brings to Web forms:
- placeholder text in an input field. Placeholder text is displayed inside the input field as long as the field is empty. When you click on (or tab to) the input field and start typing, the placeholder text disappears.
  * autofocus fields. After loading up a form, the first thing your visitors want to do is start typing. Unfortunately, they can’t - at least not until they tab over to the first control, or click it with the mouse, thereby giving it focus.

You can make this happen through the use of autofocus attribute which you can add to a single `<input>` or `<textarea>` element.

New types of input:
- email addresses.
- urls. The url type is used for input fields that should contain a URL address.
- search boxes. Search box looks and behaves almost exactly like a normal text box. On some browsers like Chrome or Safari, search boxes are styled slightly differently, with rounded corners. Also, as soon as you start typing in a search box in Chrome and Safari, a small ‘x’ icon appears on the right side that you can click to clear the box.
- telephone numbers. Right now, the only value in using the tel type is to get customized virtual keyboard on mobile browsers, which focuses on numbers and leaves out the letters.
- numbers. The number type is used for input fields that should contain a numeric value.
  You can also set restrictions on what numbers are accepted:
  Use the following attributes to specify restrictions:
  - max - specifies the maximum value allowed, min - the minimum value allowed, step - the legal number intervals, value - the default value.
  - sliders. 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.
  - date. HTML5 defines several date-related types. The date type allows the user to select a date.
  - color. The color type is used for input fields that should contain a color. The element lets you pick a color and returns the color's hexadecimal representation.

New elements:
- datalist. 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.
When people use a mobile device to fill out a form, they do not have the luxury of entering information on a full keyboard. Apple’s iPod and iPhone recognizes several of the new HTML5 input types, and dynamically changes the on-screen keyboard to optimize for that kind of input.

Default text keyboards
You can get these keyboards by just adding a normal `<input type="text" />` field to your web page. You can get the optional Go button if you put the `<input />` element in a `<form />` element that has an action attribute (value is not important). When the users presses to Go button, the form will be submitted. You can capture this event by observing the submit event on the form. It is not possible to get the Go button in a `<textarea />` element, you will always get the first keyboard with the return button.
An Overview of Different iPhone Keyboards in Mobile Safari

- Email and URL keyboards

These keyboards are a variation of the regular keyboards.

Virtually all email addresses contain the @ sign and at least one period (.), but they are unlikely to have spaces. So when you use an iPod or iPhone and focus an `<input type="email">` element, you get an on-screen keyboard that contains a smaller-than-usual space bar, plus dedicated keys for the @ and . characters.

The email keyboard also disables autocorrect and autocapitalization.

On the other hand, when you focus on Web addresses the virtual keyboard is being customized so that the space bar is being replaced with three virtual keys: a period, a forward slash and a " .com" button. (You can long-press the " .com" button to choose other common suffixes like " .org" or " .net".)
An Overview of Different iPhone Keyboards in Mobile Safari

- Numeric keyboards

Phone numbers
<input type="tel" />

Digits only
<input pattern="[0-9]*" />

These two variants of the same keyboard serve specialized functions. This first keyboard can be used to enter phone numbers: it has dedicated buttons for each digit and offers a button for special phone characters such as +, * and #. An iPhone user cannot switch back to a different keyboard layout if you use this one. It is triggered by creating an <input type="tel" /> element.

The second variant is the same but lacks the button for entering non-digit characters. You can trigger this keyboard layout by using the HTML5 pattern="[0-9]*" attribute. ‘pattern’ is an attribute that can be used on elements to specify a regular expression that the input value has to match.

This particular regular expression means “a character between 0 and 9, repeated 0 or more times”.

Perhaps the most exciting part of HTML5 forms is the automatic input validation. Consider the common problem of entering an email address into a Web form. You probably have some client-side validation in JavaScript, followed by server-side validation. HTML5 can never replace your server-side validation, but it might someday replace your client-side validation.

A surprising number of visitors won’t have JavaScript enabled. When the user tries to submit a form with an `<input type="email">` field, the browser automatically offers RFC-compliant (Request for Comments) email validation, even if scripting is disabled. HTML5 also offers validation of Web addresses entered into `<input type="url">` fields, and numbers in `<input type="number">` fields. The validation of numbers even takes into account the min and max attributes, so browsers will not let you submit the form if you enter a number that is too large.

There is no markup required to activate HTML5 form validation; it is on by default. To turn it off, use the `novalidate` attribute:

```html
<form novalidate>...
</form>
```

The most powerful (and complex) type of validation that HTML5 supports is based on regular expressions (as mentioned in the previous slide).

A regular expression is a pattern written using the regular expression language. Regular expressions are designed to match patterned text - for example, a regular expression can make sure that a postal code has the right sequence of letters and digits, or that an email address has an `@` symbol and a domain extension that’s at least two characters long. For example, consider this expression:

```
[A-Z]{3}-[0-9]{3}
```

The square brackets at the beginning define a range of allowed characters. The curly brackets that follow multiply this effect, so `{3}` means you need three uppercase letters.

The dash that follows does not have a special meaning, so it indicates that a dash must follow the three-letter sequence. Finally, `[0-9]` allows a digit from 0 to 9, and `{3}` requires 3 of them.

Once you have a regular expression, you can enforce it in any `<input>` or `<textarea>` element by adding the `pattern` attribute.

Note: Regular expressions seem like a perfect match for email addresses (and they are). However, hold off on using them this way, because HTML5 already has a dedicated input type for email addresses that has the right regular expression baked in.
• **Give required field certain background:**
  
  ```html
  input:required {
    background-color: lightyellow;
  }
  ```

• **or, highlight only required fields that hold invalid values**
  
  ```html
  input:required:invalid {
    background-color: lightyellow;
  }
  ```

Although you can’t style validation messages, you can change the appearance of the input fields based on their validation state. For example, you can give invalid values a different background color, which will appear in the text box as soon as their browser detects the problem. This can be done by adding a few new pseudoclasses:

* **required and optional** which apply styles to fields based on whether they use the required attribute.
* **valid and invalid** which apply styles to controls based on whether they contain mistakes. Most browsers won’t actually discover invalid values until the visitor tries to submit the form.
* **in-range and out-of-range**, which apply formatting controls that use the min and max attributes to limit numbers to a range.

With this setting, blank fields are automatically highlighted, because they break the required-field rule.
### HTML5: New Semantic/Structural Elements

- Better structure for documents based on author conventions e.g.

<table>
<thead>
<tr>
<th>Element</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;article&gt;</code></td>
<td>an article</td>
</tr>
<tr>
<td><code>&lt;section&gt;</code></td>
<td>a section in a document</td>
</tr>
<tr>
<td><code>&lt;header&gt;</code></td>
<td>a header for a document or section</td>
</tr>
<tr>
<td><code>&lt;footer&gt;</code></td>
<td>a footer for a document or section</td>
</tr>
<tr>
<td><code>&lt;nav&gt;</code></td>
<td>navigation links</td>
</tr>
<tr>
<td><code>&lt;aside&gt;</code></td>
<td>content aside from the page content</td>
</tr>
<tr>
<td><code>&lt;hgroup&gt;</code></td>
<td>used to group heading elements</td>
</tr>
</tbody>
</table>

Earlier people were using tables to organize info in columns/rows, later `<div>` and `<span>` tags. In HTML5 there are better possibilities to organize layout.

New content-specific elements are designed to enrich the semantic content of documents:

- `<article>` The article element represents a component of a page that consists of a self-contained composition in a document, page, application, or site and that is intended to be independently distributable or reusable, e.g. in syndication. This could be a forum post, a magazine or newspaper article, a Web log entry, a user-submitted comment, an interactive widget or gadget, or any other independent item of content.

- `<section>` The section element represents a generic document or application section. A section, in this context, is a thematic grouping of content, typically with a heading. Examples of sections would be chapters, the tabbed pages in a tabbed dialog box, or the numbered sections of a thesis. A Web site’s home page could be split into sections for an introduction, news items, contact information.

- `<header>` The header element represents a group of introductory or navigational aids. A header element is intended to usually contain the section’s heading (an h1–h6 element or an hgroup element), but this is not required. The header element can also be used to wrap a section’s table of contents, a search form, or any relevant logos.

- `<footer>` The footer element represents a footer for its nearest ancestor sectioning content or sectioning root element. A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like. Footers don’t necessarily have to appear at the end of a section, though they usually do. When the footer element contains entire sections, they represent appendices, indexes, long colophons, verbose license agreements, and other such content.

- `<aside>` The aside element represents a section of a page that consists of content that is tangentially related to the content around the aside element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography. The element can be used for typographical effects like pull quotes or sidebars, for advertising, for groups of nav elements, and for other content that is considered separate from the main content of the page.

- `<nav>` The nav element represents a section of a page that links to other pages or to parts within the page: a section with navigation links. The navigation bar which usually appears on top. Not all groups of links on a page need to be in a nav element — only sections that consist of major navigation blocks are appropriate for the nav element. In particular, it is common for footers to have a short list of links to common pages of a site, such as the terms of service, the home page, and a copyright page. The footer element alone is sufficient for such cases, without a nav element.

- `<hgroup>` The hgroup element represents the heading of a section. The element is used to group a set of h1–h6 elements when the heading has multiple levels, such as subheadings, alternative titles, or taglines.

Some elements such as `<a>`, `<cite>` and menu have been changed, redefined or standardized.

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:

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

### HTML5: New Semantic/Structural Elements

- **Progress bars and meters**

<table>
<thead>
<tr>
<th>Element</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;progress&gt;</code></td>
<td>how far a task has progressed</td>
</tr>
<tr>
<td><code>&lt;meter&gt;</code></td>
<td>a value within a known range</td>
</tr>
</tbody>
</table>

`<progress>` and `<meter>` element are two new graphical widgets that look similar but serve different purposes.

The `<progress>` element indicates how far a task has progressed.
The `<meter>` element indicates a value within a known range.

HTML5: New Elements for Multimedia and Graphical Content

- **Media playback**
  <audio>, <video>

  *No single audio/video format that works on every browser*

- **2D graphics**
  <canvas>

These features are designed to include and handle multimedia and graphical content on the Web without having to resort to proprietary plugins and APIs. These elements are used instead of previously used <object> element.

The <video> and <audio> elements are used for media playback.

The story isn’t however all rosy. The major browser companies are involved into an audio and video format war. The consequence is that there’s no single audio and video format that works on every browser, and you’ll need to encode and then re-encode your media files to get them working in HTML5.

```html
<audio>
  <audio controls="controls">
    <source src="horse.ogg" type="audio/ogg" />
    <source src="horse.mp3" type="audio/mpeg" />
  </audio>
</audio>

<video width="320" height="240" controls="controls">
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
</video>
```

Your browser does not support the audio element.

Your browser does not support the video tag.

The control attribute adds video controls, like play, pause, and volume. `loop` attribute specifies that the audio/video will start over again, every time it is finished.

`autoplay` attribute specifies that the audio/video will start playing as soon as you open the page.

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

The <audio>/<video> element allows multiple <source> elements. <source> elements can link to different video files. The browser will use the first recognized format. It is also a good idea to always include width and height attributes for <video> element. 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).

HTML5 has DOM* methods, properties, and events for the <video> and <audio> elements.

These methods, properties, and events allow you to manipulate <video> and <audio> elements using JavaScript.

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

* DOM - defines the event and document model the Web platform uses. The DOM is a language- and platform neutral interface that allows programs and scripts to dynamically access and update the content and structure of documents.

Examples:

Canvas
The <canvas> element for 2D drawing, scriptable image, a powerful API for drawing graphs and designing games, etc...
**HTML5 Graphics**

- **2D using**
  - *inline SVG* (Scalable Vector Graphics)
    - XML based, every element is available within SVG DOM
    - each drawn shape is remembered as an object
    - re-rendering the shape
  - `<canvas>` element
    - rendered pixel by pixel
    - once the graphic is drawn, it is forgotten by the browser

- **2D/3D using**
  - CSS3

---

* SVG is a language for describing 2D graphics in XML. SVG is XML based, which means that every element is available within the SVG DOM. You can attach JavaScript event handlers for an element. In SVG, each drawn shape is remembered as an object. If attributes of an SVG object are changed, the browser can automatically re-render the shape.

Currently in our CHIP demonstrator (http://www.chip-project.org/demo) we use SVG for displaying the tours on the museum map.

* Canvas draws 2D graphics, on the fly (with a JavaScript).

Canvas is rendered pixel by pixel. In canvas, once the graphic is drawn, it is forgotten by the browser. If its position should be changed, the entire scene needs to be redrawn, including any objects that might have been covered by the graphic. The `<canvas>` element is only a container for graphics. You must use a script to actually draw the graphics.

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

A canvas is a rectangular area on an HTML page, and it is specified with the `<canvas>` element.

**Note:** By default, the `<canvas>` element has no border and no content. The markup looks like this:

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

**Note:** Always specify an id attribute (to be referred to in a script), and a width and height attribute to define the size of the canvas.

**Tip:** You can have multiple `<canvas>` elements on one HTML page.

The `getContext("2d")` object is a built-in HTML5 object, with many properties and methods for drawing paths, boxes, circles, text, images, and more. It is a powerful feature that allows dragging graphics on our web page.

**Examples**

Through

* CSS3

IE and Opera only support 2D transforms and do not support 3D transforms yet. Chrome and Safari requires the prefix `-webkit-`. (webkit - a layout engine)

Firefox requires the prefix `-moz-`.
HTML5: Extending HTML

- New Attributes
- New Elements
- Full CSS3 Support
- Video and Audio
- 2D/3D Graphics
- Local Storage
- Local SQL Database
- Web Applications
CSS is used to control the style and layout of Web pages. CSS3 is the latest standard for CSS.

CSS3 is completely backwards compatible, so you will not have to change existing designs. Browsers will always support CSS2.

CSS3 is completely backwards compatible, so you will not have to change existing designs. Browsers will always support CSS2.

CSS3 is split up into "modules" — more than 30. Each module has levels. The old specification has been split into smaller pieces, and new ones are also added. Some of the most important CSS3 modules are:

* Selectors
* Box Model
* Backgrounds and Borders
* Text Effects
* 2D/3D Transformations
* Animations
* Multiple Column Layout
* User Interface

Attribute Selectors - New Way of Referencing Elements

You can reference an element by an attribute, whether attribute alone or with given value.

Pseudo-classes

Parent-child elements, more control over children

If you have a table you could e.g. show every other row or any specific row in a different color.

This will make some important items a lot easier to see.

The :nth-child() and :nth-of-type() pseudo-classes allow you to select elements with a formula.

The syntax is :nth-child(an+b), where you replace a and b by numbers of your choice. For instance, :nth-child(1) selects the 1st, 4th, 7th etc. child of type elements the same, except that it only considers elements of the given type in the examples.

Pseudo-classes example

With CSS3, you can create multiple columns for laying out text - like in newspapers!

CSS3 Borders, Text Shadows, Gradients, and Alpha

People tend to prefer rounded corners, see e.g. the browser tabs.

Adding shade was difficult. Now using CSS3 everything is very simple.

2D/3D transformations

A transformation is an effect that changes an element change shape, size and position.

You can transform your elements using 2D or 3D transformations.

With CSS3, you can move, scale, turn, spin, and stretch elements.

Internet Explorer 9 requires the prefix -ms-. Firefox requires the prefix -moz-. Chrome and Safari requires the prefix -webkit-. Opera requires the prefix -o-.

2D transformations example

CSS transitions are effects that let an element gradually change from one style to another.


Flexible Box Model

Layout for graphical user interfaces (GUI)

HTML5 & CSS3 came up with this model.

It is the future of Web design.

Flexbox – status

Experimental implementations in various browsers.

Check the difference in look & feel when changing the screen width for


and


With flexbox you do not need to apply any difficult math or css, everything changes dynamically.

You can have fluid columns and information changing automatically.

Flexing is the ability of the container to stretch or shrink to fill the available space.

We can have a media query where the first is 800px and the other is 1024px.

We do not want to make everything flexible.

A feature like this enables things inside a function. It is not going to have how big to make sure and how small to make the other.

A media query, e.g. an image in a table fixed and the other will change dynamically depending on how big the browser is.

We can use media queries to manipulate these things.

If the browser width is 800 and the fixed column width is 100 then the column width will be 100.

Reversing the Box Order

Could be useful e.g. for lining sections of a document.


About handoff

Handheld unresponsive, but there is a trick to fix that.

http://www.w3.org/Talks/2012/0913-CSS-Amsterdam/slide-media-queries.html
CSS3 Media Queries: Adapting to Different Devices

- Easiest way to use media queries
  - start with the standard version of a website
  - override it selectively

@media (media-query-property-name: value) {
  .  .  .
}

Examples of media-query-property-name:
- max-width
- max-device-width

Today it is not unusual to find developers creating custom versions of the same website for specific devices, like iPhones and iPads. These sites are often hosted on different Web domains (like http://m.nytimes.com for the mobile version of the New York Times). However as mobile browsing becomes more popular, and mobile devices become increasingly numerous and varied, Web developers can end up with big headaches managing all those device-specific sites.

Of course, separate sites aren’t the only way to deal with different devices. You can also write Web server code that checks every request, figures out what Web browser is on the other end, and sends the appropriate type of content. This costs times and requires skills.

Media queries - this CSS3 feature gives you a simple way to vary styles for different devices and viewing settings. Used carefully, they can help you serve everything from an ultra-widescreen desktop computer to an iPhone - without altering a single line of HTML.

History of CSS Media Types

*Interestingly, the creators of CSS took a crack at the multiple-device problem in CSS 2.1, using a feature called media types. You might already be using this standard to supply a separate style sheet for printouts:

```html
<head>
...<link rel="stylesheet" media="screen" href="styles.css">
<!-- Use this stylesheet to display the page on-screen. -->
<link rel="stylesheet" media="print" href="print_styles.css">
<!-- Use this stylesheet to print the page. -->
</head>
</html>

The media attribute also accepts the value handheld, which is meant for low-bandwidth, small-screen mobile devices. Most mobile devices make some attempt to pay attention to the media attribute and use the handheld style sheet, if it exists.

But there are quirks are plenty, and the media attribute is woefully inadequate for dealing with the wide range Web-connected devices that exists today. *

"To use media queries, you must first choose the property you want to examine, e.g. max-width which gets the current size of the page, in the browser window. Even more useful is max-device-width, which checks the maximum screen width. If this value is small, it's clear that you're working with a webphone or a similarly tiny device. The easiest way to use media queries is to start with the standard version of your website, and then override it selectively.

When trying to identify mobile devices like webphones, you need to use the max-device-width property, not max-width. That's because the max-width property uses the size of the phone's viewport - the segment of the Web page that the phone user can scroll around. A typical viewport is twice as wide as the actual device width.

In the following example the rules implement standard two-column layout, with a fixed 330-pixel column on the left, and a side bar on the right that expands to fill up the remaining space.

http://wwwis.win.tue.nl/~nstash/html5/examples/example32.html

The magic happens when you define a separate part of your style sheet that comes into effect for a given media-query value. In the following example the new set of styles comes into effect when the width of the browser window is 480 pixels or less.

http://wwwis.win.tue.nl/~nstash/html5/examples/example33.html
• More advanced media queries

```css
@media (min-width: 600px) and (max-width: 700px) {
  . . .
}
@media (min-width: 400px) and (max-width: 599.99px) {
  . . .
}
@media (max-width: 399.99px) {
  . . .
}
```

Sometimes you might want your styles even more specific, so that they depend on multiple conditions. This comes in handy if you want to apply several sets of mutually exclusive styles.

Now if the browser window is 380 pixels, exactly two sets of style will apply: the standard styles and the styles in the final @media block. The solution to use fractional values like 399.99 pixel measurement is used to avoid rules overlap.
• More advanced media queries

```css
@media (not max-width: 600px) and (max-width: 700px) {

}

@media (not max-width: 400px) and (max-width: 600px) {

}

@media (max-width: 400px) {

}
```

Another option is to use the ‘not’ keyword. There is no functional difference.

In these examples, there is still one level of style overriding to think about. That’s because every `@media` section starts off with the standard, no-media-query style rules. Depending on this situation, you might prefer to separate your style logic completely (for example, so a mobile device gets its own, completely independent set of styles).
To do so, you need to use media queries with external style sheets, as described next.
CSS3 Media Queries: Adapting to Different Devices

• Replacing an entire style sheet

```html
<head>
    <link rel="stylesheet" href="standard_styles">
    <link rel="stylesheet" media="(max-width:480px)"
        href="small_styles.css" >
</head>
```

Separate style sheets

```html
<head>
    <link rel="stylesheet" media="(min-width:480.01px)"
        href="standard_styles">
    <link rel="stylesheet" media="(max-width:480px)"
        href="small_styles.css" >
</head>
```

If you have simple tweaks to make, the @media block is handy, because it lets you keep all your styles together in one file. But if the changes are more significant, you may decide that it’s just easier to create a whole separate style sheet. You can then use a media query to create a link to that style sheet. The browser will download the second style sheet (small_styles.css) with the page but won’t actually apply it unless the browser width falls under the maximum. As in the previous example, the new styles will override the styles you already have in this place. In some cases, what you really want is completely separate, independent style sheets. First, you need to add a media query to your standard style sheet, to make sure it kicks in only for large sizes.
CSS3 Media Queries: Adapting to Different Devices

• Replacing an entire style sheet

Separate style sheets

```html
<link rel="stylesheet" media="(min-width:480.01px)" href="standard_styles">
<link rel="stylesheet" media="(max-width:480px)" href="small_styles.css">
<!--[if lt IE 9]>
<link rel="stylesheet" href="standard_styles">
<![endif]-->
```

The problem with this approach is that browsers that don’t understand media queries will ignore both style sheets.
You can fix this up for old versions of Internet explorer by adding your main style sheet again, but with conditional comments.
• Replacing an entire style sheet

Separate style sheets

<link rel="stylesheet" media="print and (min-width:25cm)" href="NormalPrintStyles.css">
<link rel="stylesheet" media="print and (not min-width:25cm)" href="NarrowPrintStyles.css">

Incidentally, you can combine media queries with media types. When doing this always start with the media type, and don't put it in parentheses.
If you are looking for mobile phones, check for a max-device-width of 480 pixels. This is the best, more general rule. It catches the iPhone and the Android phones that exist today.

The iPad poses a special challenge: users can turn it around to show content vertically or horizontally. And although this changes the max-width, it doesn’t alter the max-device-width. In both portrait and landscape orientation, the iPad reports a device width of 768 pixels. Fortunately, you can combine the max-device-width property with the orientation property if you want to vary styles based on the iPad’s orientation.

Of course this rule applies to other devices with the similar screen sizes.

Note: On their own, media queries probably aren’t enough to turn a normal website into a mobile-friendly one. You’ll also need to think about bandwidth and the user experience. On the bandwidth side, you’ll want to use smaller, lightweight images. (You can do this by giving elements background images, and setting these images in your styles. However, this approach is a nightmare for websites with lots of pictures.) On the user experience side, you need to think about breaking content down into smaller pieces (so less scrolling is required) and avoiding effects and interactions that are difficult to navigate with a touch interface (like pop-up menus).
One obvious difference between desktop websites and mobile websites is the way they use video. A mobile website may still include video, but it will typically use a smaller video window and a smaller media file. The reasons are obvious - not only do mobile browsers have slower, more expensive network connections to download video, they also have less powerful hardware to play it back.

Using the media query techniques you can easily change the size of a `<video>` element to suit a mobile user. However, it’s not as easy to take care of the crucial second step, and link to a slimmed-down video file.

HTML5 has a solution: it adds a media attribute directly to the `<source>` element. It specifies the media file a `<video>` element should play. By adding the media attribute, you can limit certain media files to certain device types.

An example on the slide hands butterfly_mobile.mp4 file out to small-screened devices. Other devices get butterfly.mp4 or butterfly.ogv depending on which video format they support.

Of course it is up to you to encode a separate copy of your video for mobile users. Encoding tools usually have device-specific profiles that can help you out. For example, they might have an option for encoding “iPad video”. It’s also still up to you to make sure that you use the right media format for your device and supply video formats for every other browser.

See more at http://www.w3.org/Talks/2012/0913-CSS-Amsterdam/slide-media-queries.html
Setting a viewport tells the browser how content should fit on the device's screen and informs the browser that the site is optimized for mobile.

For example:

```html
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=yes">
```

tells the browser to set the viewport to the width of the device with an initial scale of 1. This example also allows zooming, something that may be desirable for a web site but not a web app. We could prevent zooming with user-scalable=no or cap the scaling to a certain level:

```html
<meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=0.5 maximum-scale=1.0">
```

Note: width can also take a pixel value. Setting width=320 would achieve the same result as width=device-width on the iPhone and a few other smart phones. Keep in mind not all phones have this exact width, so it's best to stick with device-width and let the browser figure out the rest.

Android extends the viewport meta tag by allowing developers to to specify which screen resolution the site has been developed for:

```html
<meta name="viewport" content="target-densitydpi=device-dpi">
```

Possible values for target-densitydpi are device-dpi, high-dpi, medium-dpi, low-dpi.

If you want to modify your web page for different screen densities, use the -webkit-device-pixel-ratio CSS media query and/or the window.devicePixelRatio property in JavaScript, then set the target-densitydpi meta property to device-dpi. This stops Android from performing scaling in your web page and allows you to make the necessary adjustments for each density, via CSS and JavaScript.

See Android's WebView documentation for more information on targeting device resolutions.
APIs for client-side data storage per-session (sessionStorage) and persistently across sessions (localStorage and client-side SQL database storage).

Many more features that make Web application development easier than ever.
• With HTML5, Web application development is easier than ever:
  - Local data storage
  - Local file access
  - Local SQL database
  - Application cache
  - Javascript workers
  - XHTMLHttpRequest 2
On the Web there are two places to store information: on the Web server or on the Web client.

Local data storage or Web storage (new, better local storage than cookies***)

Local storage makes sense for user preferences (e.g. settings that influence how the Web page tailors its display) and application state (a snapshot of where the Web application is right now, so the Web visitor can pick up at the same spot later on.

With HTML5, web pages can store data locally within the user's browser - for simple things. Before you were limited to sessions and 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.

Unlike cookies, this data is never transmitted to the remote web server (unless you go out of your way to send it manually). Unlike all previous attempts at providing persistent local storage, it is implemented natively in web browsers, so it is available even when third-party browser plugins are not.

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

In the examples we fill in the form on the left and display the results on the right. When we go to a different site and then come back the data will still be there.

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

* `localStorage` - stores 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.

* `sessionStorage` - stores data for one session. The data is deleted when the user closes the browser window.

Although the HTML5 specification doesn’t lay down any hard rules about maximum storage space, most browsers limit local storage to 5MB. (From wikipedia: Web storage provides far greater storage capacity (2.5 MB per origin in Google Chrome; 5 MB per origin in Mozilla Firefox, and Opera; 10 MB per storage area in Internet Explorer) compared to 4 kB (around 1000 times less space) available to cookies.)

For larger data sets you need a database.  
http://www.webgranth.com/html5-web-storage-advantages-over-cookies

Cookies were invented early in the web’s history, and indeed they can be used for persistent local storage of small amounts of data. But they have three potentially dealbreaking downsides:

- Cookies are included with every HTTP request, thereby slowing down your web application by needlessly transmitting the same data over and over
- Cookies are included with every HTTP request, thereby sending data unencrypted over the internet (unless your entire web application is served over SSL)
- Cookies are limited to about 4 KB of data — enough to slow down your application (see above), but not enough to be terribly useful

more info on Web Storage at http://diveintohtml5.info/storage.html
**HTML5: Web Applications**

- With HTML5, Web application development is easier than ever:
  - Local data storage
  - Local file access
  - Local SQL database
  - Application cache
  - Javascript workers
  - XMLHttpRequest 2

**Application Cache**

API for offline Web applications

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.


**Javascript workers or Web workers**

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.


Some people compare Web workers to threads, that's the same use case but they are not. They are more like processes. They do not have shared memory (state), message passing.
In addition to specifying markup, HTML5 specifies scripting application programming interfaces (APIs) that can be used with JavaScript.

Drag-and-drop
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.

Geolocation
User Geolocation is not a new idea; this was done before using IP address. HTML5 Geolocation provides more detailed information about user. It is done by the getting the information from WiFi towers and GPS.
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.

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.
A Web page can lose its connection to the Web server for a variety of reasons. If possible the browser will attempt to reopen the connection automatically, after waiting a default 3 seconds.
Examples: Facebook/Twitter updates, stock price updates, news feeds, sport results, etc.
http://wwwis.win.tue.nl/~nstash/html5/examples/example27.html
Adaptive streaming and DRM:
* Media Source Exts.: Allows JavaScript to generate media streams for playback; facilitates adaptive streaming and time shifting live streams.
* Encrypted Media Exts: Enables playback of protected content; that is, decryption using license/key exchange.

Responsive images:
The Responsive Images Community Group has been exploring proposals to make it possible to load and display images in HTML that are best adapted to the resolution of the device, and is now looking into incorporating a `<picture>` element in HTML5.
* `<img srcset...>` vs `<picture>`/`<source>`
* `<picture>` proposal came from community
* Microsoft supporting `<picture>` proposal
* Other browser vendors like srcset better
* Convergence?
Useful Links: HTML5 & CSS3

- CSS3 tutorial (use Opera):
  http://www.w3.org/Talks/2012/0913-CSS-Amsterdam/
- “Dive Into HTML5” by Mark Pilgrim
  http://diveintohtml5.info/
- http://www.w3schools.com/
- w3.org/Mobile/mobile-web-app-state/
- Device orientation demo
  http://goo.gl/UVNEJ
- w3.org/brief/Mjkw
- html5please.com
- caniuse.com
- html5accessibility.com
- http://platform.html5.org/
- HTML5-aware validator:
  html5.validator.nu
- HTML5 test for feature support:
  http://html5test.com

Anne van Kesteren - local Dutch expert on HTML5
http://annevankesteren.nl/presentations/html5.html
## Use of HTML5 Features for Implementing Public Address System

<table>
<thead>
<tr>
<th>PA System Feature</th>
<th>HTML5 Features for Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>New form elements and attributes, form validation based on input types or with regular expressions, Web storage API</td>
</tr>
<tr>
<td>Arranging information on the screen, splitting up screen in sections</td>
<td>Flexible box model</td>
</tr>
<tr>
<td>2D, 3D animation Map of the floors</td>
<td>Canvas use, CSS3</td>
</tr>
<tr>
<td>Adapting to different devices</td>
<td>Media Queries</td>
</tr>
<tr>
<td>Playing out predefined announcement</td>
<td>Audio element</td>
</tr>
</tbody>
</table>
## Use of HTML5 Features for Implementing Public Address System

- **Students proposals**

<table>
<thead>
<tr>
<th>PA System Feature</th>
<th>HTML5 Features for Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc messages that have to be repeated e.g. “Passenger X, you are delaying the flight”, but later can be deleted</td>
<td>Application cache</td>
</tr>
<tr>
<td>Grouping zones/sub-zones</td>
<td>Hierarchy with CSS3</td>
</tr>
<tr>
<td>Status screen</td>
<td>CSS3 transitions</td>
</tr>
<tr>
<td>Zoom, drag&amp;drop zones to some features instead of applying features to the zones</td>
<td>Drag &amp; drop</td>
</tr>
<tr>
<td>Priorities of the employees - trying to broadcast messages in the same hall</td>
<td>Has to be considered or left out?</td>
</tr>
</tbody>
</table>
Recommendations for Developing Your HTML5 Project

1. Use your favorite HTML editor
2. For consulting about HTML5 tags/attributes use http://www.w3schools.com/
3. Check whether your browser(s) support(s) the desired features at http://html5test.com
4. Validate your HTML5 page at html5.validator.nu
End of Part 1
Assignment Description

Assignment description from Bosch


We work in groups of 4 people

HTML5: Design and implementation of a number of screen types for desktop PC, tablet, mobile device:

1. Login screen for employee
2. Configuration screen - airport map (clickable) with an option to choose zone(s) for an announcement, background music, setting volume
3. Status screen - airport map with an indication in which zones an announcement is being made, etc.

WebRTC: Audio streaming

1. Playing out prerecorded audio
2. Recording an announcement from the device on the server and playing it out, an option to delete the recorded audio
What is WebRTC?

- <w3.org/html, whatwg.org>
- Real Time Communication
- A state-of-the-art audio/video communication stack in your browser
- A cross-industry effort to create a new

Part of Google Chrome project.

WebRTC is an HTML5 standard that's being drafted by two groups, W3C and IETF (The Internet Engineering Task Force), with the goal of allowing Web browsers to conduct real-time communication. The mission of the IETF is to make the Internet work better by producing high quality, relevant technical documents that influence the way people design, use, and manage the Internet.

WebRTC is a free, open project that enables Web browsers with Real-Time Communications (RTC) capabilities via simple Javascript APIs.

Imagine a world where your phone, TV and computer could all communicate on a common platform. Imagine it was easy to add video chat to your web application. That's the vision of WebRTC. WebRTC implements open standards for real-time, plugin-free video, audio and data communication.

The guiding principles of the WebRTC project are that its APIs should be open source, free, standardized, and more efficient than existing technologies.
Why WebRTC?

• No need for downloads, native apps or plug-ins

That means that the standard will eventually enable voice calling, video chat and P2P file-sharing without requiring plug-ins or extraneous software.

The need is real:
* A lot of web services already use Real-time Communication (RTC), but need downloads, native apps or plugins. These includes Skype, Facebook (which uses Skype) and Google Hangouts (which use the Google Talk plugin).
* For end users, plugin download, installation and update can be complex, error prone and annoying.
* For developers, plugins can be difficult to deploy, debug, troubleshoot, test and maintain—and may require licensing and integration of complex, expensive technology. It can be hard to persuade people to install plugins in the first place!
WebRTC Support

• Coming to almost all desktop browsers by EOY 2012

• Mobile browser support in progress
• Native C++ implementations also available

The first WebRTC support is shipping to Chrome 21, expecting Firefox to join by the end of the year, brought support to IE via ChromeFrame.

WebRTC coming to almost all desktop browsers by EOY 2012. As this technology standardizes we will see it start or appear in various mobile browsers. For those building native apps there is native version on desktop or mobile.
WebRTC Architecture

There are two distinct layers.

1. Browser developers will be interested in the WebRTC C++ API and the capture / render hooks at their disposal.
2. Web App developers will be interested in the Web API.

* Your Web App: A third party developer web based application with video and audio chat capabilities powered by the web API for real time communications.
* WebRTC Native C++ API: An API layer that enables browser makers to easily implement the Web API proposal.
* Transport / Session: The session components are built by re-using components from libjingle, without using or requiring the xmpp/jingle protocol.
* STUN/ICE: A component allowing calls to use the STUN and ICE mechanisms to establish connections across various types of networks.
* RTP Stack: A network stack for RTP, the Real Time Protocol.
* Voice and Media Components: A collection of media processing components for voice and video:
  - ISAC: A wideband and super wideband audio codec for VoIP and streaming audio. ISAC uses 16 kHz or 32 kHz sampling frequency with an adaptive and variable bit rate of 12 to 52 kbps.
  - ISAC: A narrowband speech codec for VoIP and streaming audio. Uses 8 kHz sampling frequency with a bitrate of 15.2 kbps for 20ms frames and 13.33 kbps for 30ms frames. Defined by IETF RFCs 3951 and 3952.
  - NetEQ for Voice: A dynamic jitter buffer and error concealment algorithm used for concealing the negative effects of network jitter and packet loss. Keeps latency as low as possible while maintaining the highest voice quality.
  - Acoustic Echo Canceller (AEC): The Acoustic Echo Canceller is a software based signal processing component that removes, in real time, the acoustic echo resulting from the voice being played out coming into the active microphone.
  - Noise Reduction (NR): The Noise Reduction component is a software based signal processing component that removes certain types of background noise usually associated with VoIP. (Hiss, fan noise, etc...)
  - VideoEngine: VideoEngine is a framework video media chain for video, from camera to the network, and from network to the screen.
  - VP8: Video codec from the WebM Project. Well suited for RTC as it is designed for low latency.
  - Image enhancements: For example, removes video noise from the image capture by the webcam.

From a JavaScript perspective, the main thing to understand from this diagram is that PeerConnection shields web developers from myriad complexities that lurk beneath. The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks:

* packet loss concealment
* echo cancellation
* bandwidth adaptivity
* dynamic jitter buffering
* automatic gain control
* noise reduction and suppression
* image 'cleaning'.

http://www.webrtc.org/reference/architecture
WebRTC client applications need to do several things:

• Get streaming audio, video or data
• Communicate streaming audio, video or data
• Exchange control messages to initiate or close sessions and report errors
• Exchange information about media such as resolution and format
WebRTC

Key Features:

- MediaStreams - access to the user's camera and mic
- PeerConnection - easy audio/video calls
- DataChannels - p2p application data transfer
WebRTC: MediaStreams

- Represent a media source containing one or more synchronized MediaStreamTracks.
- Can be converted to an object URL and passed to a <video/> element.
- Use the getUserMedia API to get a MediaStream for the webcam/mic (prompts user for consent).

MediaStreams - represents a media source, containing one or more synchronized MediaStreamTracks that can be of various types.

E.g. if we get a media stream for the user’s webcam and mic, we have a single stream but a separate track for video and a separate track for audio (as shown in the diagram).

In the videoconference we can have multiple media streams and in each separate stream for each participant we have an audio and a video track.

Use the getUserMedia API to get a MediaStream for the webcam/mic (prompts user for consent).

Once we have a media stream we need a way to play it out.

A MediaStream can be converted to an object URL and passed to a <video/> element.

http://dev.w3.org/2011/webrtc/editor/getusermedia.html
WebRTC: getUserMedia Sample

```javascript
navigator.webkitGetUserMedia({video: true},
onGotStream, onFailedStream);

onGotStream = function(stream) {
  var url = webkitURL.createObjectURL(stream);
  video.src=url;
}
</script>

<html>
  <video id="video" autoplay="autoplay" />
</html>


Implementing WebRTC on a single page.

FEATURE DETECTION
Feature detecting is a simple check for the existence of navigator.webkitGetUserMedia.

GAINING ACCESS TO AN INPUT DEVICE
To use the webcam or microphone, we need to request permission. The first parameter to
webkitGetUserMedia is an object specifying the type of media you want to access. For example, if you want
to access the webcam, the first parameter should be {video: true}. To use both the microphone and camera,
pass {video: true, audio: true}.
Media capture works in conjunction with <audio> and <video> elements. Notice that we're not setting a src
attribute or including <source> elements on the <video> element.
To plug a media stream into a <video> element
we first need a way to get a URL that references the media stream.
There is a method called createObjectURL.
Plug the URL into a tag and a media will start to play.
The <video> is told to autoplay, otherwise it would be frozen on the first frame. Adding controls also works
as you'd expected.
*****
Unfortunately, currently no browser allows audio data from getUserMedia to be passed to an audio or video
element, or to other APIs such as Web Audio.
The WebRTC PeerConnection API handles audio as well as video, but audio from getUserMedia is not yet
supported in other contexts.
Therefore, further we show video examples.
WebRTC: getUserMedia + <canvas/>

<script>
function onClick() {
    snapshot.getContext("2d").drawImage(video, 0,0,canvas.width, canvas.height);
}
</script>

<video id="video" autoplay="autoplay" />
<canvas id="canvas"></canvas>
<button onclick="onClick()">Snapshot</button>

A photo booth app

Implementing WebRTC on a single page.
Adding a button for drawing video frame into a canvas, just like a camera.

The <canvas> API's ctx.drawImage(video, 0, 0) method makes it trivial to draw <video> frames to <canvas>.
An API, navigator.getUserMedia() allows web apps to access a user's camera and microphone.

ENABLING
The getUserMedia() API is still very new. In Chrome < 21, you need to enable the feature by visiting about:flags. If you're using Chrome 21, you can skip this section. Opera and Firefox do not need a flag. The feature is enabled by default.

FEATURE DETECTION
Feature detecting is a simple check for the existence of navigator.getUserMedia.
3. Gaining access to an input device

```
<video autoplay></video>

<script>
    var onFailSoHard = function(e) {
        console.log('Rejected!', e);
    }

    // Not showing vendor prefixes.
    navigator.getUserMedia({video: true, audio: true}, function(localMediaStream) {
        var video = document.querySelector('video');
        video.src = window.URL.createObjectURL(localMediaStream);

        // Note: onloadedmetadata doesn't fire in Chrome when using it with getUserMedia.
        // See crbug.com/110938.
        video.onloadedmetadata = function(e) {
            // Ready to go. Do some stuff.
        }
    }, onFailSoHard);
</script>
```

See comments to slide 58.
3. Gaining access to an input device

• If you want something that works cross-browser, try:

```javascript
window.URL = window.URL || window.webkitURL;
navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia || navigator.msGetUserMedia;

var video = document.querySelector('video');

if (navigator.getUserMedia) {
  navigator.getUserMedia({audio: true, video: true}, function(stream) {
    video.src = window.URL.createObjectURL(stream);
  }, onFailSoHard);
} else {
  video.src = 'somevideo.webm'; // fallback.
}
```
Some browsers throw up an infobar upon calling getUserMedia(), which gives users the option to grant or deny access to their camera/mic. The spec unfortunately is very quiet when it comes to security.
You call "getUserMedia" API, it prompts the user for permission to access the devices and it gives back a media stream if the callback was successful.
For users that don't have support for webkitGetUserMedia, one option is to fallback to an existing video file if the API isn't supported and/or the call fails for some reason.

```javascript
function fallback(e) {
    video.src = 'fallbackvideo.webm';
}

function success(stream) {
    video.src = window.URL.createObjectURL(stream);
}

if (!navigator.getUserMedia) {
    fallback();
} else {
    navigator.getUserMedia({video: true}, success, fallback);
}
```
Some amazing things are possible when we combine WebRTC with Web platform. That's how we get access to devices.
WebRTC: PeerConnection Basics

- API for establishing audio/video calls (“sessions”)
- Built-in:
  - Peer-to-peer
  - codec control
  - encryption
  - bandwidth management

Peer connection takes media streams and sends them across the Internet peer to peer. As the name indicates PeerConnection is the API that uses several peer to peer sessions, establishes the connection & runs audio/video over it.
- establishing peer-to-peer link
- management of various audio and video codec
- encryption
- tuning audio and video stream to make the best use of bandwidth
Check that PeerConnection is enabled in chrome://flags.

PEERCONNECTION SANS SERVERS
WebRTC from the PeerConnection point of view is described in the example below. The code is taken from the 'single page' WebRTC demo at webrtc-demos.appspot.com, which has local and remote PeerConnection (and local and remote video) on one Web page. This doesn't constitute anything very useful—caller and callee are on the same page—but it does make the workings of the PeerConnection API a little clearer, since the PeerConnection objects on the page can exchange data and messages directly without having to use intermediary servers.
Typical voice/video app, e.g. GoogleTalk. App (caller) wants to be able to send a media directly to callee. The way it does this is through the cloud by sending signaling messages over its connection using a protocol like XMPP (Extensible Messaging and Presence Protocol) or SIP (Session Initiation Protocol).
Signaling

- **Signaling** - a mechanism to send control messages between peers
- Signaling methods and protocols are not specified by WebRTC - signaling is not part of PeerConnection API
- WebRTC app developers can choose
  - preferred protocol such as:
    - SIP (Session Initiation Protocol) or
    - XMPP (Extensible Messaging and Presence Protocol)
  - appropriate duplex communication channel such as:
    - WebSocket or
    - XMLHttpRequest
  - Google Channel API

WebRTC uses PeerConnection to communicate streams of data, but also needs a mechanism to send control messages between peers, a process known as signalling. Signalling methods and protocols are not specified by WebRTC: signalling is not part of the PeerConnection API. Instead, WebRTC app developers can choose whatever messaging protocol they prefer, such as SIP or XMPP, and any appropriate duplex (two-way) communication channel such as WebSocket, or XMLHttpRequest (XHR) in tandem with the Google Channel API.
Puts minimum needed into the browser (p2p codec and encryption) and app does the rest. The app will give the parameters to the browser he wants to use for the call - session descriptions.
E.g. we could use app engine & XMLHttpRequest to post the session descriptions in JSON encodings to app engine, app engine would deliver them to the remote site.

https://apprtc.appspot.com
Or we could implement SIP protocol in JavaScript, convert session descriptions to SIP messages and send them to SIP equipment.

PeerConnection API has to handle both cases.
To start a session, a client needs:
- Local Session Description *(describes the configuration of the local side)*
- Remote Session Description *(describes the configuration of the remote side)*
- Remote Transport Candidates *(describes how to connect to the remote side)*

These parameters are exchanged via signaling, and communicated to the browser via the PeerConnection API.

The initial session description sent by the caller is called an *offer*, and the response from the callee is called an *answer*.

A peer connection needs:
* Local session description which holds the local parameters for the call.
* Remote session description indicates the remote parameters.
* Remote transport candidates - IP addresses and ports the remote side is reachable at. Sometimes these are included within session description.

These parameters are exchanged via signaling, and communicated to the browser via the PeerConnection API.

The initial session description sent by the caller is called an *offer*.
The initial parameters sent by the caller, specify everything the caller is capable of.
The response from the callee which indicates the negotiated or selected parameters is called an *answer*.

Let's walk through call set up and see how these parameters are exchanged.
The app creates the local session description - the offer, passes it to peer connection API and sends it to the remote site using whatever mechanism it wants.
The caller gets it & gives it to the peer connection API as a remote description.
Then assuming that the callee accepts the call, generates its own session description, passes it into peer connection and sends it back to the caller as an answer.
The caller gets that answer and gives it to peer connection as a received session description.
At this point the browser has everything it needs to establish the call: the local/remote session description, candidates, the peer to peer link is established and media flows.
CALLER

1. Create a new PeerConnection and add a stream (e.g. from a webcam):
   ```javascript
   var servers = null;
   pc1 = new webkitRTCPeerConnection(servers);
   //
   pc1.addStream(localstream);
   ```

2. Create a local SessionDescription, apply it and initiate a session:
   ```javascript
   pc1.createOffer(gotDescription1);
   function gotDescription1(desc){
      pc1.setLocalDescription(desc);
   //
   }
   ```

http://www.html5rocks.com/en/tutorials/webrtc/basics/
WebRTC from the PeerConnection point of view is described in the example below. The code is taken from the 'single page' WebRTC demo at webrtc-demos.appspot.com, which has local and remote PeerConnection (and local and remote video) on one web page. This doesn't constitute anything very useful—caller and callee are on the same page—but it does make the workings of the PeerConnection API a little clearer, since the PeerConnection objects on the page can exchange data and messages directly without having to use intermediary servers.

ZZZ
CALLER

3. (Wait for a response from the callee.)
4. Receive remote SessionDescription and use it:
   pc2.createAnswer(gotDescription2);
   function gotDescription2(desc){
      //
      pc1.setRemoteDescription(desc);
   }

CALLEE

1. (Receive call from caller.)

2. Create PeerConnection and set remote session description:
   pc1 = new webkitRTCPeerConnection(servers);
   pc2.onaddstream = gotRemoteStream;
   //
   pc2.setRemoteDescription(desc);

3. Create local SessionDescription, apply it, and kick off response:
3. Create local SessionDescription, apply it, and kick off response:

```
pc2.createAnswer(gotDescription2);
//
pc2.setLocalDescription(desc);
```
Peer Connection Sans Servers:
The Whole Process (Sans Logging)

// create the 'sending' PeerConnection
var servers = null;
pc1 = new webkitRTCPeerConnection(servers);

// create the 'receiving' PeerConnection
pc2 = new webkitRTCPeerConnection(servers);

// set the callback for the receiving PeerConnection to display video
pc2.onaddstream = gotRemoteStream;

// add the local stream for the sending PeerConnection
pc1.addStream(localstream);

// create an offer, with the local stream
pc1.createOffer(gotDescription1);

// set the offer for the sending and receiving PeerConnection
pc1.setLocalDescription(desc);
pc2.setRemoteDescription(desc);

// create an answer
pc2.createAnswer(gotDescription2);

// set it on the sending and receiving PeerConnection
pc2.setLocalDescription(desc);
p1.setRemoteDescription(desc);

// start the connection process
pc1.startIce();
p2.startIce();
The caller creates a peer connection, plugs a media stream into it which it got from the getUserMedia API via addStream, it then creates an offer, plugs it in and sends it to the callee.

When the callee gets it (on the right), it creates a peer connection, stuffs the offer via setRemoteDescription, and then creates its own session description as an answer that it can send back to the caller. The caller then gets its answer (on the left), sets remote description to the received answer and the set up is complete.
Backend Service

• Connecting WebRTC endpoints
WebRTC: Signaling Channel

- XMLHttpRequest works great for sending requests but receiving them isn’t as easy
- This is a real issue when trying to set up a video call
- App Engine’s Channel API provides this server->client messaging path

It is very easy to send the session description to the cloud. We make it a string and ship it over the XMLHttpRequest. But how do we receive it? Especially if this is a real-time application. The caller does not sit forever waiting for the callee.

App Engine gives a tool for this – Channel API provides an easy to use server-> client signaling path for pushing the session descriptions from the server to the client. And App Engine takes care of all the details. No matter how many users you have, where they are in the world we can use the same simple API to send our messages.
Establishing a channel with the Channel API works like this:
1. Client A generates a unique ID.
2. Client A requests a Channel token from the App Engine app, passing its ID.
3. App Engine app requests a channel and a token for the client's ID from the Channel API.
4. App sends the token to Client A.
5. Client A opens a socket and listens on the channel set up on the server.
Sending a message works like this:
1. Client B makes a POST request to the App Engine app with an update.
2. The App Engine app passes a request to the channel.
3. The channel carries a message to Client A.
4. Client A's onmessage callback is called.
WebRTC: DataChannels

• Peer-to-peer exchange of arbitrary application data:
  − Low latency
  − High message rate/throughput
  − Optional unreliable semantics

• Many real-world use cases:
  − Gaming
  − Remote desktop applications
  − Real-time text
  − File transfer
  − Decentralized networks
DataChannel Key Features

- Leverages PeerConnection session setup
- Multiple simultaneous channels, with prioritization
- Reliable and unreliable delivery semantics
- Built-in security (DTLS) and congestion control
- Can be used with or without audio/video
- Similar API to WebSockets
  - `send()`, `onMessage()`
Simple usage of data channel.
On each side you create data channel...
Works like WebSockets.
Web sockets - full duplex, bidirectional. Not exactly a tcp socket but close to it. Can be used for implementing chat.
More complicated topologies...
In order to distribute data among multiple peers we can create a mesh where each endpoint is connected to each other endpoint.
You can also create a graph. And unicast data to all your clients.
WebRTC Availability

- Chrome+ChromeFrame
  - Chrome21: MediaStreams
  - Chrome22 (target): PeerConnections
  - Q4 2012: Data Channels

- Opera
  - Opera12: MediaStreams

- Firefox
  - Early Q4 2012: MediaStreams
  - Late Q4 2012: PeerConnections, DataChannels

- Other
  - Internet Explorer support via GoogleFrame
  - Mobile browser support in progress
  - Native C++ implementations are also available
WebRTC Part of the Assignment

WebRTC: Audio streaming
1. Playing out prerecorded audio
2. Recording an announcement from the device on the server and playing it out, an option to delete the recorded audio
Q: Can the microphone input from GetUserMedia be used to pipe it to the <audio> for local playback?
A: Unfortunately, currently no browser allows audio data from getUserMedia to be passed to an <audio> or <video> element, or to other APIs such as Web Audio.
A: No, the <audio> tag does not support MediaStreams yet but it is currently being worked on at:
http://code.google.com/p/chromium/issues/detail?id=112367
Work Around With Audio

http://webaudiodeemos.appspot.com/input/index.html

Testing WebRTC on Chrome (Web Audio Input flag must be enabled):
- Canary build
- Dev Channel build

http://www.webrtc.org/running-the-demos
latest Chrome Canary (24.0.1288.0)
Chrome 24 (Stable version)
WebRTC: Status of Implementation

Recording
Q: What is the status of the recording functionality?
A: Recording does not have a stable specification yet and the current focus is on PeerConnection.

A: Recording an audio is possible through:

```html
<h2>capture=.microphone</h2>
<input type="file" accept="audio/*;capture= microphone">
```

Support:
• Android 3.0 browser
• Chrome for Android (0.16)
• Firefox Mobile 10.0
• iOS6 Safari and Chrome (partial support)
HTML Media Capture:
The Device API in Android 3.0 Honeycomb
Useful Links: WebRTC

- http://www.webrtc.org/
- http://www.youtube.com/watch?v=E8C8ouiXHHk
- http://www.w3.org/TR/2012/WD-webRTC-20120821/