# Developing Mobile-Compatible Web Pages

## Problems in Mobile Web Development

One common problem in web development is to ensure that your site is usable on both desktop computers and mobile devices, such as mobile phones. A related issue is that, even if you are aiming ***only*** at phones, you will probably want to make your site viewable across a wide range of different models running different operating systems, such as Android, iPhone OS and Symbian. There are a number of reasons why this might require extra effort.

- The screen size is much smaller on a typical mobile device, and a different orientation (portrait rather than landscape) compared to a desktop machine. Furthermore, different mobile devices have different screen resolutions, e.g. a tablet such as an iPad is much bigger than a typical smartphone, and a basic (non smart) phone will have a smaller screen than a smartphone.
- Different mobile devices run different browsers, or different versions of the same browser. Phone browsers include Chrome, Firefox, Safari. Each browser supports HTML, CSS and JavaScript to a different extent. Some of the more advanced features, such as AJAX or the Canvas tag, are only available on up-to-date phone browsers. Some older phones do not even support full XHTML, but instead, support a subset known as XHTML-MP; and some very old phones may not even support commonly-used image formats such as JPEG or PNG. ***Given that many smartphone browsers are comparable in features to desktop browsers these days, this is less of a problem than it used to be.***

## What can we do about it?

There are a number of techniques that we can use to maximise the chances of our website behaving in a user-friendly manner on smartphone browsers:

- Use ***CSS Media Queries*** to link in different stylesheets depending on the properties of the device.
- Use a framework such as ***Bootstrap***
- Use a JavaScript-based ***feature sniffing library***
- Use a server-side ***Device Description Repository*** (DDR) - a collection of data describing the properties of different devices and their capabilities, e.g. screen resolution, whether AJAX or HTML5 Canvas is supported, and so on.

## CSS Media Queries

***Reference:* *Mozilla developer page***

***CSS Media Queries*** allow us to define different stylesheets, or stylesheet rules, depending on the properties of the device. What do we mean by this? Here is an example:

```
<html>
<head>
<link rel="stylesheet" media="only screen and (max-widt
<link rel="stylesheet" media="only screen and (min-widt
</head>
<body>
...
```

Note the media query, specified in the ***media*** property of the ***link*** tag. This code is telling the browser to include the stylesheet ***mobile.css*** if it is a screen-based device (as opposed to a printer, for instance) and the width of the screen is 599 pixels or less, or to include the stylesheet ***desktop.css*** if it is a screen-based device and the width of the screen is 600 pixels or more. So for a typical smartphone the first stylesheet will be linked in, and for a typical desktop browser the second stylesheet will be included.

When we say ***width***, we mean the width of the ***browser viewport***, i.e. the currently visible area of the browser window in which the page content is displayed. We do not mean the width of the device itself. This has the advantage that the "mobile" layout will apply if we resize the window of a regular desktop browser below 600 pixels - which is probably what we want.

## Different media types

As well as ***screen***, there are other media types such as ***aural***, ***braille*** or ***print***, for different types of devices. These three are for screen readers (which speak the contents of a web page), braille-based browsers, and printers, respectively.

## Other CSS Media Query directives

Besides ***max-width*** and ***min-width***, a range of other useful CSS media query rules exist. They include:

- ***max-device-width*** and ***min-device-width***. These specify the maximum and minimum width of ***the entire device***, rather than the viewport, as described in ***max-width*** and ***min-width*** above. So, if you were to use ***max-device-width*** and ***min-device-width*** you could use one stylesheet for mobile devices and another for desktop browsers, the desktop stylesheet being applied even if you resized the browser window on a desktop machine to a narrow width.

- ***Aspect ratio*** rules. These relate to the aspect ratio of the viewport or the device, in other words the width divided by the height. This allows certain layouts to be applied to landscape screens (those where the width is greater than the height) and others to portrait screens (where the height is greater than the width). The aspect ratio rules include:
  - ***aspect-ratio and device-aspect-ratio***: the aspect ratio of the viewport and the device itself (use above for the difference between the viewport and the device). Use these if you want a stylesheet for a ***specific*** aspect ratio. These are expressed as fractions, e.g.:

```
<link rel="stylesheet" type="text/css" media="or
```

  - ***max-aspect-ratio***, ***min-aspect-ratio***, ***max-device-aspect-ratio*** and ***min-device-aspect-ratio***. These allow you to specify the maximum or minimum aspect ratio for a particular style. What do "maximum" and "minimum" mean? Since aspect ratio is width divided by height, it follows that the greater the width is relative to the height, the higher the aspect ratio. So a ***max-aspect-ratio*** of ***3/2***. e.g:

```
<link rel="stylesheet" type="text/css" media="or
```

    , would apply the stylesheet if the width is 1.5 times the height or less.
- ***orientation***: a shortcut for the aspect ratio rules above. This can have the values "portrait" or "landscape" and will link a particular stylesheet if the device is in landscape or portrait orientation respectively. For example:

```
<link rel="stylesheet" type="text/css" media="only
```

- ***color***, ***min-color*** and ***max-color*** - used to specify different stylesheets for different colour depths. ***color*** on its own will apply the stylesheet if it is a colour device while ***min-color*** and ***max-color*** will apply the stylesheet if the colour depth is at least, or at most, a given number of bits of colour. So for example:

```
<link rel="stylesheet" type="text/css" media="only
```

  would only apply the stylesheet ***highcolour.css*** if the device had at least 24-bit colour.

## Combining conditions

You can combine conditions with the use of ***and*** and ***not*** operators, for example, this will only link in the stylesheet if the width of the viewport is at least 800 pixels ***and*** there are at least 24 bits of colour:

```
<link rel="stylesheet" type="text/css" media="only scre
 href="desktop.css" />
```

or this will link in the stylesheet if the viewport is ***not*** at least 800 pixels (i.e. less than 800 pixels) but the colour depth is at least 24:

```
<link rel="stylesheet" type="text/css" media="only scre
 href="mobile.css" />
```

There is not an "or" operator but you can separate criteria with a comma and the stylesheet will be applied if any of the media are met, e.g:

```
<link rel="stylesheet" type="text/css" media="only scre
only screen and (min-color:24)" href="style.css" />
```

would apply the stylesheet if ***either*** the width is at least 800 pixels ***or*** the colour is at least 24-bit.

## Media Queries and individual CSS rules

The examples so far have shown how to link in whole stylesheets depending on the properties of the device. However in certain cases we might just want to incorporate ***certain rules (selectors)*** within the CSS depending on the device properties, for example the dimensions of a specific ***div***. Here is how to do this in a CSS stylesheet:

```
@media only screen and (min-width: 800px)
{
   #div1
   {
       width: 800px;
```

```
        height: 600px;
    }

    #div2
    {
        width: 800px;
        height: 400px;
    }
}

@media only screen and (max-width: 480px)
{
    #div1
    {
        width: 320px;
        height: 200px;
    }

    #div2
    {
        width: 320px;
        height:200px;
    }
}
```

The first pair of rules will be applied if the width is at least 800 pixels, and the second, if the width is no more than 480 pixels. Note how we specify the media rule and then use curly brackets { and } to define the block of CSS which applies to this media rule. Within this outer block we list one or more CSS selectors (rules) which will be applied if the media rule holds true.

## Recommendations for Mobile Design

Some sources (e.g. this article here) recommend the use of so-called "fluid" layouts when designing for desktop and for mobile devices. A "fluid" design adapts easily to changing device resolution. How can one create a "fluid" layout? The steps include:

- Avoid using pixel widths and use percentages instead where possible (images, being of fixed size, will demand pixel widths but elements filled with text do not)
- Or even better: use "ems" for dimensions. An "em" is the height of the current font. So using "ems" will help the layout scale to different font sizes. Some mobile browsers use a smaller font size by default.

# Dynamic media query testing with JavaScript: matchMedia

Sometimes we might want our pages to be even more dynamic in response to different devices than CSS media queries alone allow. The media queries you have seen so far allow you to position different elements at different places depending on the properties of the device. However, we might want more control than that, for example we might want the layout to be ***entirely different*** in landscape and portrait orientation. We might want certain s to be ***completely hidden*** in a portrait mobile layout. We could always set ***visibility:none*** in the CSS for all the s that we want to hide, however this is inefficient as we are loading all these elements into memory and not showing them. An example is Freemap; on a mobile device in portrait layout we might not want to show the sidebar and login form ***at all*** and instead just have a top bar above the map with the website title and a menu icon which loads up menu options, including an option for logging in - rather like a typical native Android app layout.

To allow this, we can ***test whether a particular media query is true in JavaScript***. Then, we can dynamically create different HTML content depending on whether the media query is true or not. This is done via the ***window.matchMedia()*** method (supported in Firefox, Chrome and IE10+, see caniuse.com). Here is an example:

```
var mq = window.matchMedia("(orientation: portrait)");
if (mq.matches)
{
    alert("Portrait orientation");
}
else
{
    alert("Landscape orientation");
}
```

As you can see we use ***matchMedia()*** with a given media query (orientation: portrait here). This returns a media query object which includes a boolean property ***matches***, which will be true if the media query matches. In this example it simply alerts "Portrait orientation" or "Landscape orientation" depending on whether the query matches. In a real-world example, we would dynamically create different HTML for portrait and landscape layouts.

## Adding an orientation change listener

The above example will test the current orientation of the device ***at the point the code is run***. However it will ***not*** dynamically detect live orientation changes and change the layout as the orientation changes. To do this we

need to ***listen for an orientation change event***. Here is an example of how we can do this:

```
function init()
{
    var mq = window.matchMedia("(orientation: portrai
    mq.addListener(onOrientationChange);
    onOrientationChange(mq);
}

function onOrientationChange(mq)
{
    if (mq.matches)
    {
        alert("Portrait orientation");
        // setup layout for portrait
    }
    else
    {
        alert("Landscape orientation");
        // setup layout for landscape
    }
}
```

In this example, in our ***init()*** function (which might run when the page first loads), we call the ***matchMedia()*** method with our media query. This returns a media query object, as before. However, rather than doing the test instantly, we attach a ***listener*** to the media query object with the line:

```
mq.addListener(onOrientationChange);
```

This will set up a callback function, ***onOrientationChange()***, to run whenever the device's orientation changes, e.g. the user rotates the device. If we look at this function, we can see that the media query object is passed to it automatically as a parameter, rather like the XMLHttpRequest object is passed to an AJAX callback automatically as a parameter. In the callback function we then use the ***matches*** property of this object to see if the media query currently matches, and display a different message depending on whether the query does match. So every time the orientation changes, an alert will appear displaying the current orientation. (Again in a "real world" example this would setup an appropriate layout for landscape or portrait).

Lastly in the ***init()*** function we call our callback directly, with the line:

```
onOrientationChange(mq);
```

This is so that we can set up the appropriate layout for the current orientation when the page first loads. If this was missed out, the layout would be uninitialised when the page first loads; we would have to wait until the user rotates the device for the layout to be initialised.

## Exercises

1. Develop your web mapping example to use a sidebar/main content area layout. The sidebar should contain links to OpenStreetMap (www.openstreetmap.org), OpenCycleMap (www.opencyclemap.org) and Freemap (www.free-map.org.uk). Use: sidebar width 20%, main content width 80%,e.g:

```
#sidebar
{
    position:absolute;
    top:0%;
    left:0%;
    width:20%;
    height:100%
}

#main
{
    position:absolute;
    top:0%;
    left:20%;
    width:80%;
    height:100%
}
```

2. Use CSS media queries so that a different layout is produced when the screen width is 480 pixels or less. This alternative layout should have a map of width 480 pixels, height 200 pixels and should show the copyright message and the three links immediately above the map, so that everything is in one column, not two. The width of the sidebar (now top bar) should be 100%. This "narrower" layout is more mobile-friendly.
3. Change your answer so that the first layout appears if the aspect ratio is greater than 2/3, and the second layout if the aspect ratio is less than 2/3.
4. Change your answer so that, in portrait mode, the top bar just contains the heading "Mapping". Do not include the links (the intention is to save space; the idea would be the user would click on a "menu" link which brings up an Android-style menu of the links but you do not need to implement this). Use JavaScript's window.matchMedia() to do this and use innerHTML to populate the contents of the sidebar/top bar with appropriate HTML.

5. Extend the previous question to dynamically change the layout when the user resizes the window