scientiamobile

# Using WURFL in ASP.NET Applications

An application that uses WURFL is simply an application that needs to check a number of effective browser capabilities in order to decide what to serve. The WURFL database contains information about a huge number of devices and mobile browsers. This information essentially consists in a list of over 600 capabilities. Each device, uniquely identified by its user-agent string (UAS), falls into a group and inherits the capabilities defined for the group, plus the delta of base capabilities that make it unique.

Programming a WURFL-based application is therefore a fairly easy task. All you need to do is loading WURFL data in memory and query for specific capabilities you're interested in for the currently requesting browser/device. Let's step into some more details for an ASP.NET application. Note that the WURFL API makes no difference between ASP.NET Web Forms and ASP.NET MVC. What you need to in both cases is exactly the same.

## Required Binaries

To enable WURFL on your application you need to download the WURFL binaries and data. Binaries include **wurfl.dll** and **wurfl.aspnet.extensions** that you The **wurfl.dll** file must be added as a reference to any WURFL project. The **wurfl.aspnet.extensions** file must be referenced only in ASP.NET projects where you plan to use WURFL. For example, you don't strictly need to reference **wurfl.aspnet.extensions** if you're using WURFL from within a console application.

## Loading WURFL Data

WURFL data changes over time but it's not certainly real time data. So you can blissfully load all of it at the startup of the application and keep it cached for as long as you feel comfortable. When an update is released, you just replace the data and restart the application. Both programming and deployment aspects of the operation are under your total control.

In an ASP.NET application, the **Application_Start** method is the place where you perform all one-off initialization work. Here's how you can instruct the method to load and cache WURFL data.

```
public class Global : HttpApplication
{
    public const String WurflDataFilePath = "~/App_Data/wurfl.zip";
    public const String WurflPatchFilePath = "~/App_Data/web_browsers_patch.xml";

    private void Application_Start(Object sender, EventArgs e)
    {
        var wurflDataFile = HttpContext.Current.Server.MapPath(WurflDataFilePath);
        var wurflPatchFile = HttpContext.Current.Server.MapPath(WurflPatchFilePath);

        var configurer = new InMemoryConfigurer()
                .MainFile(wurflDataFile)
                .PatchFile(wurflPatchFile);
        WURFLManagerBuilder.Build(configurer);
    }
}
```

You can specify multiple patch files by simply calling the **PatchFile** method multiple times in the same chained expression.

```
var configurer = new InMemoryConfigurer()
                .MainFile(wurflDataFile)
                .PatchFile(yourWurflPatchFile1)
                .PatchFile(yourWurflPatchFile2);
```

It is important to note that in this version of WURFL you only to add your own patch files as there are no longer system patch files to add to the list.

As you can see, both file names and cache details are under your control. You might want to maintain a copy of the WURFL data on your Web server. The API doesn't currently support reading from other than local files.

In ASP.NET applications, you can also specify the WURFL data files in the **web.config** file. In this case, you replace the call to **InMemoryConfigurer** with **ApplicationConfigurer**.

```
var configurer = new ApplicationConfigurer();
```

The **web.config** section has to look like below:

```
<wurfl>
   <mainFile path="~/..." />
   <patches>
      <patch path="~/..." />
      :
   </patches>
</wurfl>
```

Note that the <wurfl> section is a user-defined section and needs be registered before use with the .NET infrastructure. For this reason, you also need to add the following at the top of the web.config file:

```
<configuration>
  <configSections>
     <section name="wurfl"
              type="WURFL.Aspnet.Extensions.Config.WURFLConfigurationSection,
                    Wurfl.Aspnet.Extensions, Version=1.5.0.0, Culture=neutral" />
  </configSections>
  :
</configuration>
```

You can use the ASP.NET tilde (~) operator to reference the root of your site when specifying file paths. Note that the **ApplicationConfigurer** class is defined in the **wurfl.aspnet.extensions** assembly so if you plan to use configuration files in, say, a console application that does batch processing, then you need to reference the **wurfl.aspnet.extensions** assembly as well.

## Querying for Capabilities

Once you hold a WURFL manager object in your hands, you're pretty much done. The WURFL manager has its own cache holding the entire database of device information. You can reference the WURFL manager using the following code:

```
var device = WURFLManagerBuilder.Instance.GetDeviceForRequest(userAgent);
```

A WURFL manager is an object that implements the **IWURFLManager** interface. The WURFL Manager offers a few methods for you to gain access to the in-memory representation of the detected device model. You can query for a device in a variety of ways: passing the user agent string, the device ID, or a WURFL specific request object. The WURFL specific request object —the class **WURFLRequest**—is merely an aggregate of data that includes the user agent string and profile.

All of the methods on the WURFL manager class return an **IDevice** object which represents the matched device model. The interface has the following structure:

```
public interface IDevice
{
    String Id { get; }
    String UserAgent { get; }
    String NormalizedUserAgent { get; }
    String FallbackId { get; }
    Boolean IsActualDeviceRoot { get; }
    String GetCapability(String name);
    IDictionary<String, String> GetCapabilities();
    String GetVirtualCapability(String name);
    IDictionary<String, String> GetVirtualCapabilities();
    IDevice GetDeviceRoot();
    String GetDeviceRootId();
    IDevice GetFallbackDevice();
    String GetMatcher();
    String GetMethod();

}
```

You can check the value of a specific capability through the following code:

```
var is_tablet = device.GetCapability("is_tablet");
```

If you call the **GetCapabilities** method you get a dictionary of name/value pairs. The value associated with a capability is always expressed as a string even when it logically represents a number or a Boolean.

Armed with capabilities—WURFL supports more than 600 different capabilities—you are ready to apply the *multi-serving* pattern to your next ASP.NET mobile Web site.

## Match Mode: High-Performance (HP) vs. High-Accuracy (HA)

Starting with version 1.4, the WURFL API introduces a new concept—the match mode. This is an additional parameter you can use to drive the behavior of the API. In High-Performance mode (the default), the API will adopt heuristics to detect the majority of desktop web browsers without involving the matcher logic nor allocating memory space for them. This is useful for installations in which WURFL manages mixed web/mobile HTTP traffic. In High-Accuracy mode, the user agent is processed as in previous versions and ensures the most accurate answer. Setting the match mode is important only if your WURFL installation deals with both web and mobile traffic.
You can set the match mode through the configurer object of your choice. If you use the **InMemoryConfigurer**, you can pick a match mode with the following method:

```
public InMemoryConfigurer SetTarget(MatchMode target)
```

The **MatchMode** enumeration contains the following values:

| Value | Description |
|---|---|
| **Accuracy** | This algorithm ensures the highest accuracy but may take a bit more time to execute. |
| **Performance** | This algorithm ensures a much faster answer on desktop browsers.<br>*For non-desktop devices, there's nearly no difference between the two match modes.* |

Likewise, if you plan to use the **ApplicationConfigurer** then you have a brand new mode XML attribute on the <wurfl> node to set to either **Accuracy** or **Performance**.

```
<wurfl mode="Performance|Accuracy">
    <mainFile path="~/App_Data/wurfl.zip" />
</wurfl>
```

The **mode** attribute is optional; if not specified it defaults to Performance.

Note that you can also set the match mode on a per-request basis. In this case, the match mode you indicate works only for the specific request. Successive requests will default to whatever option you set through the configurer object. You indicate the match mode for a request using one of the overloads for the **GetDeviceForRequest** method:

```
var deviceInfo = WURFLManagerBuilder.Instance.GetDeviceForRequest(
                                                    userAgent, MatchMode.Accuracy);
```

Note also that a request that indicates explicitly a match mode will not be served through the internal cache and will not have the response cached for further references. For this reason, you might want to configure the system for using the match mode that most suits you and use the overload of **GetDeviceForRequest** for special requests.

## Virtual Capabilities

Version 1.5 of the API introduces the concept of virtual capabilities. A virtual capability doesn't represent a physical capability of the device; it is rather a capability that is recognized to the device based on the values of a few physical capabilities. The table below lists the supported virtual capabilities:

| Virtual Capability | Description |
|---|---|
| **is_smartphone** | Indicates whether the device is reckoned to be a smartphone. Internally, the matcher checks the operating system, screen width, touch and a few other capabilities. |
| **is_ios** | Indicates whether the device runs iOS (any version). |
| **is_android** | Indicates whether the device runs Android (any version). |
| **is_app** | Indicates whether you're getting the request through the browser embedded in a native app. This typically happens if |

| | |
|---|---|
| | the web page is hosted in a WebView component. |
| **is_robot** | Indicates whether the device is reckoned to be a robot. |
| **advertised_device_os** | Returns the common name of the operating system mounted on the device. |
| **advertised_device_os_version** | Returns the version of the operating system mounted on the device. |
| **advertised_browser** | Returns the common name of the default browser mounted on the device. |
| **advertised_ browser _version** | Returns the version of the default browser mounted on the device. |

All virtual capabilities return a string—either "true" or "false". You query for a virtual capability using the following code:

```
var gotSmartphone = device.GetVirtualCapability("is_smartphone");
```

You can override the default returned value for a virtual capability using a matching **control_cap** property in your patch file. For example, let's say that for a given device WURFL sets the **is_smartphone** virtual capability to true. Let's also say that for your application it would be better to treat that device as a plain phone. All you do is adding a **controlcap_is_smartphone** property to the patch file for the given device and assign it any of the following values: **force_true** or **force_false**.

Note that properties such as **is_wireless_device**, **is_tablet** and **is_smarttv** already exist in the WURFL database as regular properties.

## Capability Filters

With WURFL 1.5, you get the possibility of loading and processing only a segment of the properties available in the database with subsequent gain in terms of memory occupation. You can filter capabilities by group name or capability name. If you indicate a group name then all capabilities in that group will be loaded. Otherwise, you can just list the capabilities you're interested in regardless of the group they're defined in. You can set capability filters in two ways: via configuration or programmatically.

If you opt for configuration, you do as follows:

```
<wurfl mode="Accuracy">
    <mainFile path="~/App_Data/wurfl-latest.zip" />
    <filter groups="product_info" caps="resolution_width,is_smarttv" />
</wurfl>
```

In this case, you get all the capabilities in the **product_info** group plus the **resolution_width** and is_smarttv capabilities. Multiple groups can be expressed as comma-separated names.

You can achieve the same programmatically by working on the new methods added to the **InMemoryConfigurer** class:

```
var configurer = new InMemoryConfigurer()
                .MainFile(wurflDataFile)
                .SelectGroups("product_info")
                .SelectCapabilities("resolution_width", "is_smarttv");
```

For more information about the WURFL capabilities and groups, check out: http://wurfl.sourceforge.net/help_doc.php.

## Summary

The WURFL API currently available for ASP.NET is a rather generic one that works very closely to the metal and doesn't provide much abstraction. This is good news as it allows ASP.NET developers to fully customize every single aspect of their code interaction with WURFL. In particular, you can control the caching logic and can easily define your helper classes taking into account testability and dependency injection.