How to Use the Winflector API from JavaScript

This tutorial will guide you through the process of creating a custom web service backed by Winflector.

Full Winflector JavaScript API reference is available at: http://www.winflector.com/js_docs_out/index.html.

The most recent and online version of this manual in HTML format is available at: http://www.winflector.com/support/javascript-winflector-api.html.

> **Note:** You need an **activated** commercial or preview license of Winflector Server to be able to create a custom web interface.

Getting Started

Find a directory `[Winflector]\server\httproot` in your server machine. `[Winflector]` represents a directory where you installed the Winflector Server. The directory will contain some files, for now leave them unchanged.

Create a file `index.html` in this directory. Once you do it the directory becomes a root of your custom Winflector-backed web service. You can edit `index.html` directly or redirect your web users to another HTML file.

> **Important:** Never create a file `login.html` and never redirect the users to its URL. We use this URL internally for the default (non-customized) web interface. If you use this URL your users will see the default interface instead of your own.

First, you must include the JavaScript scripts into your web page. Put this tag in the head or in the body of your HTML page:

```
<script type="text/javascript" src="/virtual/wflogin.min.js">
</script>
```

Your minimum `index.html` file may look like this:

```
<html>
<head>
  <script type="text/javascript" src="/virtual/wflogin.min.js">
  </script>
</head>
<body>
</body>
</html>
```

Although more scripts are used by `wflogin.min.js` you don't have to include them because it will load them when needed.

Then you must obtain an access to the Winflector API. You will have to insert directly or include from an external file another script in your page and put this line inside:

```
var wfapi = WFAPI.getInstance();
```

The `WFAPI` class implements a *singleton pattern* which means:

- there should exist at most one object of `WFAPI` class,
- it's safe to call `WFAPI.getInstance()` multiple times, you will always get the same object as a result,
- you should not create the `WFAPI` object with `new WFAPI()` even if it is possible,
- actually, we have modified the construstor to ensure it will not create a new object even if you call `new WFAPI()`.

As an additional feature, the `WFAPI` object automatically saves its internal state. This means that you can move to another web page and call `WFAPI.getInstance()` again and it will restore the same looking `WFAPI` object as you had on a previous page. This works as long as the user keeps the browser tab open and does not move to another web domain.

Notes for Internet Explorer pre-10

Winflector API uses typed arrays. They are supported in most modern web browsers. However, they have not been provided in Internet Explorer 9.x and before. If you want to support users with these older browsers we have provided a custom *polyfill* implementation of typed arrays. The script URL is `/virtual/typedarray.min.js` so you have to include:

```
<script type="text/javascript" src="/virtual/typedarray.min.js">
</script>
<script type="text/javascript" src="/virtual/wflogin.min.js">
</script>
```

Note the correct order of including the files.

Logging In

Once you have a `WFAPI` object you must log in to the Winflector server. You may have a different way to obtain the credentials but here let's assume they must be entered by a user. Put these tags somewhere in the body of your custom web page:

```
<table border="0">
<tr>
  <td>Login:</td>
  <td><input type="text" size="40" maxlength="30" id="login"></td>
</tr>
<tr>
  <td>Password:</td>
  <td><input type="password" size="40" maxlength="30" id="password"></td>
</tr>
<tr>
  <td rowspan="2" align="center">
    <input type="button" value="Login" onclick="mystartlogin()">
  </td>
</tr>
</table>
```

When the user clicks the button a `mystartlogin` function will be called. It must obtain the credentials and use them to log in:

```
<script type="text/javascript">
  function mystartlogin() {
      var login_value = document.getElementById("login").value;
      var password_value = document.getElementById("password").value;

      var wfapi = WFAPI.getInstance();
      wfapi.sendLogin(login_value, password_value, null, onloggedin);
  }
</script>
```

As the fourth parameter, `onloggedin` in this example, a callback function should be passed. This may be a function identifier or an anonymous function. It will be called by `WFAPI` once the login is completed successfully. The `WFAPI` object will be passed as the only argument of this function. Here is a sample implementation of this function:

```
function onloggedin(wfapi) {
    alert("Hello " + wfapi.getLogin() +
          ", you have logged in successfully.");
}
```

However, usually you will want to obtain a list of applications in this callback.

Do We Need a Domain?

If Winflector Server is configured by an administrator to authenticate the users by Windows or Active Directory rather than Winflector itself then another parameter, the Windows domain, may be required. It should be delivered the same way as the user's login name and passed as the third argument of `WFAPI.sendLogin()` method. If the domain is not needed, `null` or an empty string should be passed.

Whether the server uses the Windows authentication and needs a Windows domain is determined by the `showdomain` cookie which should be present even before you obtain the `WFAPI` object. A nonzero numerical value (usually 1) indicates that you need the domain, a value 0 or a missing cookie means that you don't need a domain. You can read this cookie value and customize your Windows domain login field e.g. in your `body.onload` event:

```
<head>
<script type="text/javascript">

  function setup_domain() {
      var cookies = document.cookie;
      var index = cookies.indexOf("showdomain=");
      if (index >= 0) {
          var sd_value = parseInt(
              cookies.substr(index + "showdomain=".length));
          if (!isNaN(sd_value) && sd_value != 0) {
              // Create or show the domain entry
          }
      }
      // Delete or hide the domain entry or just do nothing
      // if it does not exist or is hidden by default
  }

</script>
</head>

<body onload="setup_domain()">
...
```

Obtaining the List of Applications

Once you are logged in you can obtain a list of available applications. Logging in and obtaining the list of applications are two separate operations because you may want to move to another web page to display the list of applications but for simplicity we'll put all the code in one page. In order to obtain this list you have to make sure you are logged in and then call `WFAPI.getApps()`:

```
function onloggedin(wfapi) {
    wfapi.getApps(onappsloaded);
}
```

The `WFAPI.getApps()` method accepts a callback function as its single argument. The callback will be called once the applications are obtained and the `WFAPI` object will be passed as its only argument:

```
function onappsloaded(wfapi) {
  var table, i;

  document.body.appendChild(document.createTextNode("We have " +
          wfapi.Applications.length + " applications available:"));
  table = document.createElement("table");

  for (i in wfapi.Applications) {
      var tr, td, img;
      var app = wfapi.Applications[i];

      tr = document.createElement("tr");
      td = document.createElement("td");
      img = document.createElement("img");
      img.src = app.IconURL;
      td.appendChild(img);
      tr.appendChild(td);

      td = document.createElement("td");
      td.appendChild(document.createTextNode(app.Name));
      tr.appendChild(td);
      table.appendChild(tr);
  }
  document.body.appendChild(table);
}

function onloggedin(wfapi) {
  wfapi.getApps(onappsloaded);
}
```

This simple implementation presents all available applications (their icons and human-readable names) in a table.

The applications are provided in the `Applications` member of the `WFAPI` object which is an array of objects of `WFAPI.Application` class.

You can also provide all your callback functions as the anonymous functions, if you prefer:

```
<script type="text/javascript">
  function mystartlogin() {
      // ...

      var wfapi = WFAPI.getInstance();
      wfapi.sendLogin(login_value, password_value, null,
          function (wfapi) {
              wfapi.getApps(function (wfapi) {
                  document.body.appendChild(document.createTextNode(
                      "We have " + wfapi.Applications.length +
                      " applications available:"));
              // Here goes the rest of this function...
          });
      });
  }
</script>
```

If you decide to move to another web page to display the list of applications you must call `WFAPI.getInstance()` again to obtain the `WFAPI` object but the object you will obtain will be logged in already. So you must only call `getApps()` immediately:

```
function onappsloaded(wfapi) {
  // ...
}

var wfapi = WFAPI.getInstance();
wfapi.getApps(onappsloaded);
```

Launching an Application

All you need is to call the `run()` method of a `WFAPI.Application` instance. In the example below instead of defining an `onloggedin` function we pass an inline anonymous function which just launches the first application immediately, if it exists:

```
// ...
var wfapi = WFAPI.getInstance();
```

```
// Log in here or make sure you are logged in already
wfapi.getApps(function (wfapi) {
  if (wfapi.Applications[0])
      wfapi.Applications[0].run(WFAPI.Application.RUN_JAVASCRIPT);
});
```

In the following example our modified `onappsloaded` function searches for an application named `"Notepad"` and launches it. You must pass this function identifier as the argument of `WFAPI.getApps()`:

```
function onappsloaded (wfapi) {
  var i;

  for (i in wfapi.Applications) {
      if (wfapi.Applications[i].Name == "Notepad") {
          wfapi.Applications[i].run(WFAPI.Application.RUN_NATIVE);
          return;
      }
  }
}
```

The first argument of `WFAPI.Application.run()` method determines which client will be launched:

- `WFAPI.Application.RUN_JAVASCRIPT` - will open a new browser window and launch a JavaScript Winflector client. This client should work on most of the modern browsers, however some of them are unsupported.
- `WFAPI.Application.RUN_NATIVE` - will launch a platform-native Winflector client, if it is installed. Note that a native client may not be installed or may not be available for the current platform.
- `WFAPI.Application.RUN_UPDATE_NATIVE` - will not actually run a native Winflector client but will update it. This is necessary in case if a native client is installed but its version is too old and does not recognize the runtime parameters. Usually even if the client is older but is able to start and connect to the server it will update itself automatically.

`WFAPI.Application.run()` can also accept more arguments:

- The second argument: working directory of the application (on the remote server).
- The third argument: runtime parameters of the application.

Here is an example of another `onappsloaded` function which searches for an application named `"Java"` and launches it in a specified working directory and with the specified runtime parameters:

```
function onappsloaded (wfapi) {
  var i;

  for (i in wfapi.Applications) {
      if (wfapi.Applications[i].Name == "Java") {
          wfapi.Applications[i].run(WFAPI.Application.RUN_JAVASCRIPT,
                    "C:\\Java\\examples",
                    "-cp example1\\example.jar com.example.Example1");
          return;
      }
  }
}
```

These arguments are optional and may not be provided if not needed. Note that there are also situations where these parameters are predefined by the server administrator. In this case they should not be provided by the client; if provided they will be ignored.

Real World Examples

The examples above were simple and useful only if you as a developer know which application you want to run. In the real web services you will probably list the applications and let the user decide which one to run. Let's return back to our example which lists the applications in a table and add one more button:

```
function onappsloaded(wfapi) {
  // ...

  for (i in wfapi.Applications) {
      var tr, td, img, button;
      var app = wfapi.Applications[i];

      tr = document.createElement("tr");
      td = document.createElement("td");
      img = document.createElement("img");
      img.src = app.IconURL;
      td.appendChild(img);
      tr.appendChild(td);

      td = document.createElement("td");
      td.appendChild(document.createTextNode(app.Name));
      tr.appendChild(td);

      td = document.createElement("td");
      button = document.createElement("input");
```

```
            button.type = "button";
            button.value = "Run";
            button.onclick = ...        // Please see below for an explanation
            td.appendChild(button);
            tr.appendChild(td);

            table.appendChild(tr);
        }
        // ...
    }
```

Please try to figure out what should be assigned to `button.onclick`. The simplest answer `button.onclick = app.run(...);` is definitely wrong: you must assign a *function reference* which will be called later, not *call the function* now. You may think about this solution:

```
        // ...
            button.onclick = function() {
                app.run(WFAPI.Application.RUN_JAVASCRIPT);
            };
        // ...
```

but this is also wrong: this really does assign a function but this function refers to its outer variable `app` which changes its value later. The result is that whichever button a user clicks always the last application from the list is launched. This is the common loop problem in JavaScript and it must be solved with a *closure*:

```
    function create_onclick(app) {
        return function() {
            app.run(WFAPI.Application.RUN_JAVASCRIPT);
        };
    }

    function onappsloaded(wfapi) {
      // ...
            button.onclick = create_onclick(app);
      // ...
```

The same solution written with an anonymous self-invoking function:

```
    function onappsloaded(wfapi) {
      // ...
            button.onclick = function(a) {
```

```
            return function() {
                a.run(WFAPI.Application.RUN_JAVASCRIPT);
            };
        }(app);
  // ...
```

Some modern implementations of JavaScript also provide a `Function.prototype.bind()` method which can be useful:

```
function onappsloaded(wfapi) {
  // ...
      button.onclick = function(a) {
          a.run(WFAPI.Application.RUN_JAVASCRIPT);
      }.bind(this, app);
  // ...
```

Remember to make sure that the users' browsers support this method before using it.

Handling Errors

Runtime errors happen, the simplest example is a bad login name or a password provided by a user. Usually you will want to display a custom feedback to the user. In that case you should provide a callback function and set it as the `onerror` member of the `WFAPI` object. Here is an example:

```
function error_handler (number, title, text) {
  var msg;
  if (title && text)
      msg = title + "\n\n" + text;
  else if (title)
      msg = title;
  else
      msg = text;

  alert(msg);
}

// ...

var wfapi = WFAPI.getInstance();
wfapi.onerror = error_handler;
```

You may define your callback as a named or anonymous function.

The callback function accepts three parameters:

- `number` - the error number, usually you may ignore it but it may be useful if you want to provide a specific way to handle some errors and don't want to parse the error message text; for example the error number `-99` indicates a bad login or password and you will want to ask the user to retype the credentials again;
- `title` - the suggested localized title of the message box;
- `text` - the suggested localized main error message.

The `title` and `text` parameters may be missing or empty, you will have to be ready to provide your custom title (e.g., just the `"Error"` string) or construct the error message from the number (e.g., `"Error " + number`); also the `number` may be missing or set to 0 but it will never happen that both `text` and `number` are missing or empty or set to 0. If it happens this means there is no error and you should return from your function immediately:

```
function error_handler (number, title, text) {
  if (!number && !text)
      return;
  // ...
```

You must set your error handling function as the `wfapi.onerror` member *before* you call a method which may cause an error:

```
var wfapi = WFAPI.getInstance();
wfapi.onerror = error_handler;
wfapi.sendLogin(login_value, password_value, null, onloggedin);
```

If you don't provide your custom error handler a default one will be called which will provide some simple feedback.

Progress Feedback

Most operations in Winflector API for JavaScript are performed fast and don't need a progress feedback. However, `getApps()` is an exception: although it provides the results in about one second on most modern browsers it may take up to several seconds if encryption is turned on by the Winflector Server administrator and the client runs in an old browser, like Internet Explorer 9 and before, which does not provide a native implementation of typed arrays. In these cases the progress feedback (usually a progress bar) would be useful to ensure the user that the system is not deadlocked and still processes the results.

You can provide your custom callback function for progress feedback and set it as the `wfapi.onprogress` member. It will be called multiple times with a single argument *percentage* which is a number between 0 and 100 expressing the progress position in percents. The progress callback will never be called with a value 100, instead the callback provided as the argument of `getApps()` will be called to indicate this is the end of the operation. Also note that an operation may end with an error, in that case the `onerror` callback will be called. Here is an example how the progress may be provided to the user:

```
function progress_handler (percent) {
  var nice_percent = percent.toString().substring(0, 4);
  var div = getElementById("progress");
  if (!div) {
      div = document.createElement("div");
      div.setAttribute("id", "progress");
      document.body.appendChild(div);
  }
  div.innerHTML = "Downloading applications list: " +
      nice_percent + "%";
}

function error_handler (number, title, text) {
  if (!number && !text)
      return;

  var div = getElementById("progress");
  if (div)
      document.body.removeChild(div);

  // ...
}

// ...

var wfapi = WFAPI.getInstance();
wfapi.onerror = error_handler;
wfapi.onprogress = progress_handler;

wfapi.getApps(function (wfapi) {
  var div = getElementById("progress");
  if (div)
      document.body.removeChild(div);

  // ...
});
```

Information about the Server Configuration

The server sends you some useful information about its configuration along with the list of applications. The data are set by the `WFAPI.getApps()` method. Some of the members of the `WFAPI` object:

- `AppWorkingDirVisible` and `AppParametersVisible`: whether the user should be able to provide the custom working directory and custom runtime parameters for all applications. The value may be 1 (`true`) or 0 (`false`). These flags are global and apply to all applications. Note that even if a value of one of these global flags is `true` some applications may disallow providing working directory and/or runtime parameters individually.
- `ButtonRunVisible` and `ButtonRunInBrowserVisible`: whether the user should be able to run applications in a native client or in a web-based client, respectively. These flags may be enabled or disabled by the server administrator. Additionally, the server detects the client browser and platform and turns these parameters off if any of the clients is not available for the current client's platform.
- `LoginPageTimeout` - time in milliseconds from now after which the server session will expire. It is useful to set the timer in the browser and provide a feedback as soon as the specified time passes instead of letting the server provide the error message when the user tries to use the session after some time of inactivity.

Also the `WFAPI.Application` object may contain some useful configuration information:

- `WDirDisabled -` whether the user should not be allowed to enter the working directory for this particular application (even if the `WFAPI.AppWorkingDirVisible` says it should be possible).
- `ParsDisabled -` whether the user should not be allowed to enter the rutime parameters for this particular application (even if the `WFAPI.AppParametersVisible` says it should be possible).

Note that all these informations are provided for you by the server and you should not change it. Any change will have no effect except confusing yourself.

Please read the reference documentation for more detailed information about `WFAPI`.

User's Options

The users may want to customize some options of their Winflector Client, for example the remote desktop size or whether they want ClearType^{TM} fonts (more smooth look but more network bandwidth used). In order to let them display the options dialog box you must:

1. Include these scripts along with other scripts:

   ```
   <script type="text/javascript" src="/virtual/commongui.min.js">
   </script>
   <script type="text/javascript" src="/virtual/wfclopts.min.js">
   </script>
   <script type="text/javascript" src="/virtual/wfsow.js">
   </script>
   ```

2. Call the function `wfl_ShowClientOptions()`, for example:

   ```
   <input type="button" value="Set client options"
      onclick="wfl_ShowClientOptions(); return false;">
   ```

Note that you should take the `WFAPI.ButtonSetClientOptionsVisible` member into account. You should allow the users display the options box only if this member value is `true` (or 1 or any nonzero number).

Language Options

Winflector API for JavaScript provides some limited localization options. They apply only to the texts generated by this library, for example the error messages. They do not intend to provide the complete localization solution for your custom web service. In order to select the language you should use `l=`*number* attribute in your URL, where *number* is one of:

- `0` - English,
- `1` - Polish,
- `2` - Italian,
- `5` - German,
- `9` - Dutch,

for example:

```
http://www.example.com:8080/index.html?l=9
```

If you use this value at least once in any URL it's stored in a cookie and reused the next time. The default value (if you never use the `l=` attribute) is 0 (English).