



Contents

- [1 Mobile Web Server Tutorial for Custom Applications](#)
- [2 Getting started](#)
 - ◆ [2.1 Installing MWS to a mobile device](#)
 - ◆ [2.2 Installing MWS to an emulator](#)
 - ◆ [2.3 Getting the documentation](#)
 - ◆ [2.4 Development tools](#)
- [3 Implementing an MWS application](#)
 - ◆ [3.1 Reference applications](#)
 - ◆ [3.2 MWS page layout](#)
 - ◆ [3.3 MWS directory structure](#)
 - ◆ [3.4 Application entrypoint](#)
 - ◆ [3.5 Adding your application to the main menu](#)
 - ◆ [3.6 Localisation](#)
 - ◆ [3.7 User access control](#)
 - ◇ [3.7.1 How to add your application to the framework's user management](#)
 - ◇ [3.7.2 How to check a user's rights](#)
 - ◇ [3.7.3 Get users of MWS](#)
 - ◇ [3.7.4 Adding the "Change access rights" button to your application](#)
 - ◆ [3.8 Adding data to framework content](#)
 - ◆ [3.9 Call sequence](#)
- [4 Development tips](#)
 - ◆ [4.1 Debug mode](#)
 - ◆ [4.2 Logging](#)
 - ◆ [4.3 Reducing bandwidth usage](#)
 - ◆ [4.4 Faster development with PSP templates and Python modules](#)
 - ◆ [4.5 File browser HTTP API](#)
 - ◆ [4.6 Call sequence with JavaScript](#)
- [5 Conclusion](#)

Mobile Web Server Tutorial for Custom Applications

The framework in Mobile Web Server v1.5 has been improved to provide better support for custom applications. This document explains the process of developing a custom Web application using the MWS/mymobilesite.net application framework. As a reference application, we have developed a file browser capable of browsing your mobile device's file system; downloading, removing, and uploading files. This application also demonstrates user access control with the application framework and serves as a useful utility when developing your MWS application.

Getting started

This section describes the basic steps for getting started with Mobile Web Server development.

Installing MWS to a mobile device

If you are reading this tutorial, you probably already have MWS installed on your phone. If not, install and configure MWS to your device by following the instructions at mymobilesite.net.

Installing MWS to an emulator

Developing with a mobile device may not be as fast as you'd like because you have to restart the server each time you change the Python sources. Install the [emulator plug-in](#) to reduce delays caused by slow Internet connections or server restarts, and to avoid the need to update the files on the phone.

Getting the documentation

Here are a few useful links.

- [The MWS framework](#)
- The PyS60 documentation from [SourceForge](#)(the .pdf file)
- [Apache 2.0 documentation](#)
- [mod_python 3.3.1 documentation](#)
- [Python 2.2.2 documentation](#)

Development tools

Because we are mostly working with the Python programming language, a Python source code editor is needed. A simple text editor will suffice. Numerous editors with at least highlighting support are available, but [Eclipse](#) with [PyDev](#) plug-in is a very good choice.

For deploying Python applications as a installable Symbian SIS package, check out Ensymble:
<http://code.google.com/p/ensymble/wiki/Welcome?tm=6>

Implementing an MWS application

This section describes how to create an application for MWS. In this tutorial, the assumption is that MWS has been installed on the C: drive.

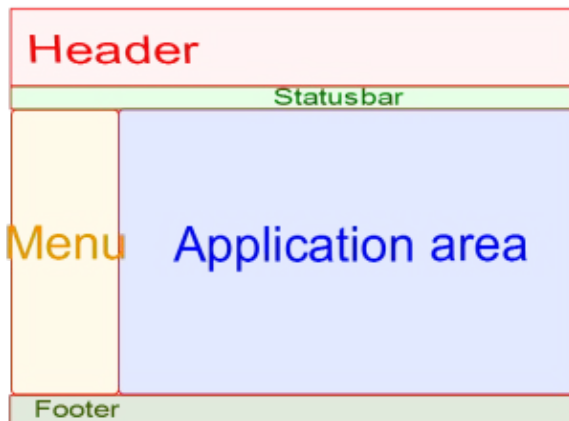
Reference applications

Download the [File browser](#). Install the application into MWS by copying the Filebrowser folder into device folder /data/Web server/htdocs/Web_Applications. Copy into the same drive (memory card, internal memory or phone memory) where you have installed the MWS.

There is also a simple application called [SkeletonApp](#) to make it easier for you to get started with your own application.

MWS page layout

The MWS page layout consists of header, footer, status bar, menu, and application areas. The header contains the image set by the current theme, some information about the phone, and links to administrator tools. The status bar contains information set by the administrator. The menu area contains buttons to launch the MWS application. The application area is filled with the content of the current application. The header and status bar can be used by applications to provide constant content. Normally you'll only need to be concerned about the contents of the application area and the limited width of the area.



MWS directory structure

The applications are located under c:\data\web server\htdocs\web_applications\. Each Web application has its own folder. The basic directory structure looks like this:

Mobile_Web_Server_Tutorial_for_custom_applications

```
Web_Applications
|- Filebrowser
    |- FilebrowserController.py
    |- webapp_conf.py
    |- viewer
        |- index.psp
```

Each application has a controller module, **webapp_conf.py** configuration file, and **viewer** folder for PSP templates. When the server is started, the framework scans the application folder looking for the **webapp_conf.py** file. This file defines the basic entry points for your application and is mandatory. For the file browser it looks like this:

```
#: The id of the webapp and is used in the url when
#: using the application
url_name = "filebrowser"
#: The name of the main module and class of the application
controller_name = "FilebrowserController"
```

The FilebrowserController.py module must exist in the application's folder containing a class with the same name in the application's folder. There cannot be more than one MWS application using the same url_name.

The url_name is used to refer to your application in the URL. This makes it possible for you to access your application directly by adding an "application=myapp" GET parameter to the URL of your mymobilesite.net page.

Application entrypoint

The main class of the application must be inherited from **WebAppController**. By default, the WebAppController uses PSP templates to generate page content. Each page is defined with an **action** GET parameter, for example, ".py?application=myapp&action=mypage". Each action must have an action handler in the controller with the same name. Actions must be defined in the available_actions list. If the action defined in the URI is not found from the list, the action is set to 'index'. Each action must also have a PSP template in the 'viewer' folder. The name of the PSP template can be controlled with a **self.viewer** attribute or by overriding the **_get_viewer_filename()** method. By default, self.viewer is the same as the action, and the file name is self.viewer + ".psp". It is also possible to implement your own pages directly by overloading the **get_content(self, req)** method. The skeleton application demonstrates both approaches:

```
...
class SkeletonAppController(WebAppController):
    """Controller class of webapp"""
    ...

    def get_content( self, req ):
        """You can override the default PSP templating here"""
        if "action" in req.form:
            if req.form["action"] == "nopsp":
                return "<h1>Hello World without PSP!<h1>"

        return WebAppController.get_content( self, req )

#: The actions using PSP. Called by giving action=<actionname> as GET parameter.
available_actions = ['index']

#: Action functions
def index(self):
```

```
# Hello world by PSP
return ""
```

...

Adding your application to the main menu

The main module must have a `show_in_menu` function, which controls who can see your application in the main menu. If `True` is returned, the folder of the Web application is used on the menu item. You can customise the item's label by returning a string, which also allows you to localise the application.

```
def show_in_menu(user_id, lang="en"):
    return True
```

Localisation

The framework does not require any special localisation files; use whatever method you feel necessary. The framework gives you the ID of the current language, and it is up to you how to use the ID. On Python, there is no need for any special localisation files such as XML. It is very easy to provide localisation by simply creating a Python module for each language. The modules are named so that they differ only by the language ID, for example, `mws_filebrowser_en_loc.py` and `mws_filebrowser_fr_loc.py`. Each module defines different values for the variables that are to be localised. If the import of the localization module fails, you can catch `ImportError` and revert to the default language of your choice. Keeping the localised strings away from the code modules makes it easier to ensure that the file encoding is set correctly. The IDs of the supported languages are located in the `WebFrameworkConstants.py` file. Currently, they are 'en', 'fr', 'de', 'it', and 'es'.

The following code demonstrates how to import module dynamically:

```
#: Template for getting correct localization module for language
TEMPLATE_LOCALIZATION_MODULE = "mws_filebrowser_%s_loc"

def get_localization( langid ):
    """@return: The localization module for current language"""
    loc = None
    try:
        loc = __import__( TEMPLATE_LOCALIZATION_MODULE % ( langid ) )
    except ImportError:
        # Use the default localization then
        import mws_filebrowser_en_loc
        loc = mws_filebrowser_en_loc

    return loc
```

User access control

MWS uses the mobile device's contacts to centrally manage the user accounts and permissions for each Web application. This removes the basic need to implement your own user management in your own application. If you have additional user management requirements, such as file access permissions of the file browser, you need to come up with additional access control on top of the framework's own. Nevertheless, the framework gives you the users to manage.

The framework's access control requires an ID for each access right level. On File browser, the shared files and folders are configured by typing the shared paths. If the default access control was used, the shared resources would have to be identified first and then they could be assigned to users. The user interface of File browser would be stricter and more complicated this way. Because there is no share-specific user access controlling, you can still use the framework to set read and write permissions for users and groups.

How to add your application to the framework's user management

For an example of how to add your application to the "My settings->Access rights" configuration, see the following code from File browser:

```
def __init__(self, app_config):
    ...
    self.accessrights = {
        "name"      : loc.APPLICATION_TITLE,
        "id"        : self.appConfig['id'],
        "rights"    : (
            ( self.right_read,      loc.RIGHT_READ,      0 ),
            ( self.right_readwrite, loc.RIGHT_READWRITE, 1 ),
        ),
        "exclusive" : 1,
    }
    ...
```

The "name" is the localised name of your application, shown in the "Edit access rights" view when editing users' permissions. "id" is the ID of your application, which you can retrieve from the **self.appConfig** dictionary.

"Rights" is a tuple of 3-tuples (id, name, weight). 'Id' is a numeric value of your choosing for the permission. 'Name' is the localised name shown in the "Edit access rights" view, where you assign the rights for the user. If "exclusive" is 0, only ID and name are required (2-tuple). If "exclusive" is enabled, a user can have only one of the rights enabled. 'Weight' is used to avoid situation where you are actually removing rights instead of adding them and is used in exclusive mode. In our case, if a user has right 'self.right_readwrite', he is able to read and write files. It is not possible to also give him right 'self.right_read', because it has lower weight. Only one access right setting can be active at a time.

How to check a user's rights

In addition to configuring the access rights, you must also implement the **check_access** method for your controller class. The method simply returns True if the user has suitable permissions and False otherwise. The framework provides a few functions to check permissions. Function **has_access** is the most common. It requires three parameters: the ID of the user to check, the application's ID, and the ID of the permission to check. See the file browser's `access_control.py` module, where there are `user_can_read` and `user_can_write` functions to check whether the user has read-only access or read-write access.

```
from Framework.Authentication import has_access

def user_can_read(fbctrl):
    """@return: True if user has permission to use file browser."""
    readwrite = user_can_write(fbctrl)
```

Mobile_Web_Server_Tutorial_for_custom_applications

```
read      = has_access( fbctrl.current_user,
                       get_webapp_id(webapp_conf.url_name),
                       fbctrl.right_read )

return ( read or readwrite )
```

In the case of File browser, the user can also read if s/he has write access. And if the user can read, s/he can use the application. Thus, the implementation of `check_access` of the FilebrowserController is as follows:

```
def check_access( self, user_id):
    """ Does the user have rights to this application?
    Framework calls.
    """
    self.current_user = user_id
    return access_control.user_can_read(self)
```

If the administrator is the only user of your application, you don't need the access control configuration. Simply use **Authentication.is_admin(id=userid)** in the **check_access** method of the controller class and in the module's **show_in_menu** function.

Get users of MWS

The framework provides a utility function for retrieving all user IDs: **Authentication.get_all_users()**. For accessing the MWS database directly, use the Database class from the Framework.Database module and supply WebFrameworkConstants.NATIVE_DB_PATH and the UserEntry class from Framework.UserEntry to get more information about the users.

Adding the "Change access rights" button to your application

First, define the button to open the settings view. The button is located in the same heading as the application's title. Put the following HTML before your application's title, but make sure it is shown only when the current user is the administrator. By setting div's class to "accessright", the button is positioned the same way as it is used in the default applications. Replace the application's name in the href attribute and the application's title with your application. The localization handling code is omitted for clarity.

```
<h1>
  <div class="accessright">
    <a href=".py?application=myapp&action=accessrights" class="button">
      Change access rights
    </a>
  </div>
  MyApplicationTitle
</h1>
<div class="line"></div>
```

Next, define the accessrights view itself. The framework provides a PSP template for creating the view semi-automatically. See the "accessright_*" files in the file browser's viewer folder for an example and make modifications to suit your needs. While editing, remember that you need to create a page for each access ID, for example, read and read&write access have their own page. Of course you can choose to not use the framework's PSP template and create the page on your own, as we did with the file sharing settings.

Mobile_Web_Server_Tutorial_for_custom_applications

Next, modify the application's controller class to use the template when the action is "accessrights". Again, remember to check if the user is the administrator and add "accessrights" to your **available_actions** list. See the following from File browser.

```
class FilebrowserController( WebAppController ):
...
    def accessrights_readwrite(self):
        """ Create application access right editing view """
        if is_admin( id = self.req.session['user_id'] ):
            self.data["loc"] = self.get_localization()
            return None

        return access_control.html_no_access(self)

    def accessrights_read(self):
        """ Create application access right editing view """
        if is_admin( id = self.req.session['user_id'] ):
            self.data["loc"] = self.get_localization()
            return None

        return access_control.html_no_access(self)
...
```

Adding data to framework content

Applications can register themselves as constant content providers, which means that you can add content to the main application header or to the status bar. To do that, you must register a callback function to the framework. The callback takes a request object as a parameter. The content to add is simply returned as a string. See the following code from File browser about how to register your callback function as a constant data provider.

```
# Register constant content
from Framework.ConstantContentRegistry import register, Position

#: Localization module cache
LOCALIZATION = None

def constant_content_callback(req):
    """ Shows free space in the statusbar """
    if LOCALIZATION is None: return ""

    # Check user permissions first
    user_id = req.session["user_id"]
    if not has_access( user_id, get_webapp_id( webapp_conf.url_name ) ):
        return ""

    # Code left out for clarity
    ...

    return result.encode( "utf-8" )

register( Position.status, constant_content_callback )
```

To add content to the header, supply `Position.header` for the `register()` function.

Call sequence

As a summary, see the following sequence diagram for what and when the framework calls the application functions.

-
- F = Framework
 - A = Web Application
-

- F: System initialisation, looking for applications.
 - ◆ F->A: Found **webapp_conf.py**.
 - ◇ F: Read **url_name** and **controller_name**.
 - ◆ F->A: Importing controller module.
- F: Handling request.
 - ◆ F->A: Calls **show_in_menu()** for each application module.
 - ◇ Returns True, False, or localised application title.
 - ◆ F->A: Create instance of the application controller class (**WebAppController**).
 - ◆ F->A: Call **WebAppController.check_access()**.
 - ◇ Returns True if user has access, False otherwise.
 - ◆ F->A: Call **WebAppController.get_content()**.
 - ◇ A: Default behaviour: Call handler of the current action.
 - ◇ A: Call **self.get_bypass_template()**.
 - A: If False, call **self._build_view()** to handle PSP template.
 - A: Call **WebAppController._get_viewer_filename()** to get PSP filename.
 - ◆ F->A: Call **WebAppController.get_bypass_template()**.
 - ◇ If False: framework content added.
 - F->A: Call **WebAppController.get_stylesheet()**
 - ◆ F->A: Call all the registered **constant content callbacks**.

Development tips

This section presents a few methods that should simplify development.

Debug mode

To enable debug mode, set `DEBUG=1` in `htdocs\Framework\WebFrameworkConstants.py`. This shows the exception traces on the browser if an exception is raised in the application. You should also enable Apache's debug mode by setting "PythonDebug On" in "Web server\conf\htdocs.inc".

Logging

Another place to look for exceptions is the data\web server\logs\error.log file. You can write to the error.log yourself with the `mod_python.apache.log_error()` function or use the more preferred `request.log_error()`.

Unfortunately, the files in the logs folder are problematic because they cannot be read on the device if Apache is running. The files are always kept open, and Symbian does not let you share the files. You cannot use PC Suite or File browser to read them while the Apache server is running.

See the file browser's `fb_utils.py` for an example of how to create an easy-to-use logging function of your own. The `log()` function writes logging data to the "log.txt" located in the file browser's installation folder, and it takes any type of argument, removing the need to convert the data, such as integers, into string type. As a bonus, it also has logs of the calling line.

Reducing bandwidth usage

The 3G connection is slow when compared to landline broadband connections or WLAN. Always try to reduce the amount of transferred data. This can be achieved by using style sheets and embedding JavaScript? to the HTML page. Style sheets can be used to create styles for different elements, removing duplication and allowing a more efficient way to define the visual style of a page. For example, Mobile Web Server's overall theme can be modified by simply changing the style sheet. JavaScript can be used to create requests to a Web server and update only parts of the page. This, of course, requires cooperation from the server side by providing a suitable HTTP interface to allow requesting data for different parts of the page. This technique is called "Asynchronous JavaScript and XML" aka [AJAX](#).

File browser uses JavaScript to update the page whenever the user removes a file or directory from the phone or creates a new folder. The use of JavaScript in the file browser example is very basic to keep the implementation simple. The interaction could be more efficient by using more compact JSON data instead of sending complete HTML markup. Use of JSON would require more logic to the JavaScript code to generate suitable HTML markup in the browser. An example of how to use JSON can be found from File browser's `html_tab_browser.py` source. The user's file share settings are received as JSON.

Normally the framework adds its own HTML code to the response. To be able to use the AJAX technique, disable the framework's default template. To do that, call the `_set_bypass_template()` method inherited from the `WebAppController`, before returning from `get_content()`. Another way is to override the `get_bypass_template()` method to return True. `_set_bypass_template()` disables all PSP template processing, including your own. If you use PSP templates, you must process them yourself; see this page: [A Brief Introduction to Apache's mod_python Module](#).

To process your template manually, see the following code:

```
# Name of the psp template in viewer folder.
self.viewer = "mytemplate"
# Build PSP
self._build_view(req)

return ""
```

Send your own data directly by calling the `write()` method of the request object.

```
self.req.write( "<h1>Hello world!</h1>" )
```

To provide a custom style sheet for your application, override **WebAppController.get_stylesheet()** and return the URI of the style sheet. For example:

```
def get_stylesheet(self):
    return self.link( action="stylesheet")

# Action handler of stylesheet
def stylesheet(self):
    self._set_bypass_template()
    self.req.write( (CSS contents) )
```

Faster development with PSP templates and Python modules

PSP templates are useful when developing a page, since, unlike Python sources, the PSP templates are always reloaded for each request. This allows you to change page contents without restarting the server. It may also be tempting to put all your code in PSP templates, but this leads to situations where the presentation is not cleanly separated from the logic, as stated in the article [A Brief Introduction to Apache's mod_python Module](#).

Another way to handle the refresh issue is to simply put your page's code into a separate module and reload it for each request. However, this reduces the server's performance. You can improve performance by reloading the module only when it has been changed, or implement a "development" mode for your application that, when enabled, reloads the Python modules. Another challenge with developing good PSP templates is that you often need to edit the Python source to be able to provide the data required by the PSP template. Changing the Python source requires you to restart the server, which is a slow process even on the emulator (sometimes, especially on the emulator). By keeping the code in a different module and reloading it for each request, you can be more efficient in developing good PSP templates and Python code.

File browser employs the separate modules method to allow faster development of the application by using a development mode flag located in **fb_utils.py**. The reloading happens in the controller's `get_content` method for each new request allowing rapid changes in Python code.

File browser HTTP API

File browser has the following interfaces to be used with AJAX. You can use these directly with the browser by writing the URL to the address bar.

Get directory contents:

```
user.mymobilesite.net/.py?application=filebrowser&action=dirdata&path=<path to directory>
```

Returns: HTML contents.

Get free space:

```
user.mymobilesite.net/.py?application=filebrowser&action=drivespace
```

Returns: HTML contents.

Remove file or empty directory from phone:

```
user.mymobilesite.net/.py?application=filebrowser&action=fsremove&path=<path to file>
```

Returns: "true" or error as HTML.

Create a new folder to device:

```
user.mymobilesite.net/.py?application=filebrowser&action=mkdir&path=<path to root directory>&mkdi
```

Returns: "true" or error as HTML.

Call sequence with JavaScript

Below is the interaction sequence between the browser and server when the user deletes a file.

- U = User
 - B = Browser
 - S = Mobile Web Server
-

- U->B: User clicks delete.
 - B->U: Ask user confirmation.
 - U->B: Confirm.
 - ◆ B->S Send delete request.
 - ◆ S->S completed, success or failure.
 - ◇ Success
 - B->S Request: give me updated free space.
 - B->S Request: give me updated directory contents.
 - S->B Free space as HTML.
 - B: Update page.
 - S->B Directory contents as HTML.
 - B: Update page.
 - ◇ Failure
 - B: Update error field contents with the error message.
 - U->B: Cancel.
 - ◆ B: Do nothing.
-

Conclusion

This tutorial explained the most important features of Mobile Web Server's application framework and offered several examples of how to accomplish tasks. It is intended to help developers start writing their own applications.