Web Dev

The Web Dev module allows users to directly program against the web server inside the Ignition Gateway. Webpages can be built by hand using a combination of python programming and static web resources such as images, CSS files, JavaScript files, and HTML files. Likewise, this module allows you to build RESTful web service APIs that allow external systems to interact with the Ignition server. This module follows the normal installation process.

> **Disclaimer**
> The Web Dev module requires specialized web-programming knowledge. The Inductive Automation support team is unable to provide detailed advice about creating a particular site. Furthermore, they are unable to provide troubleshooting beyond the basic functionality of the module.

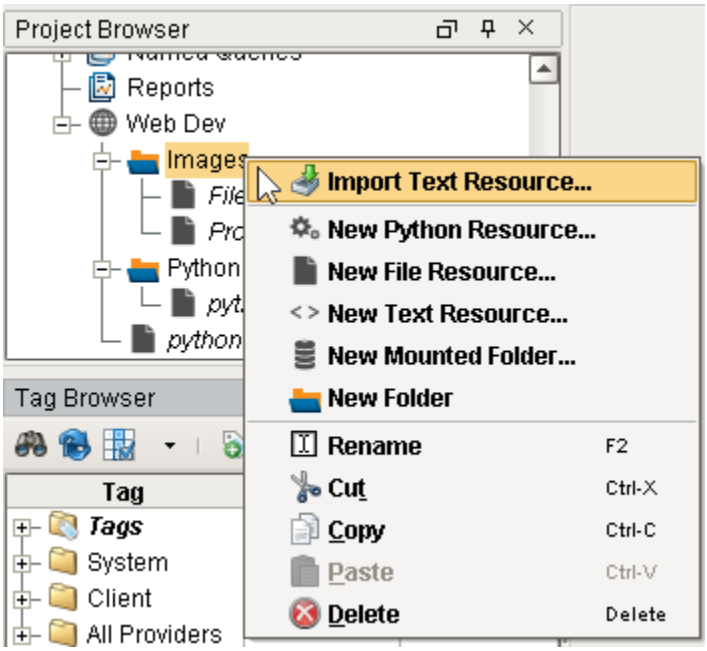
On this page

...

- Basic Usage
  - Python Resource
  - File Resource
  - Text Resource
  - Mounted Folder
- URL
- Python Resources
- Parameters
  - Request Parameter
  - Session Parameter
- Return Value
- Security Settings
  - Require HTTPS
  - Require Authentication
- httpPost Example

Basic Usage

Each type of resource may specify its *content type*. It is important to specify the correct content type for the contents of the resource. When the Web Dev module is installed, a new kind of project resource heading will appear in the Designer's project browser called "Web Dev". Right-clicking on this heading will allow the creation of several new types of project resources:



Python Resource

Python resources are dynamic web resources. Each time a user browses to the URL associated with a python resource, the script will run and generate a response in the form of a python dictionary. See Return Value for more details on formatting this response.

File Resource

A file resource is a static resource, usually a binary resource such as an image. Note that you will need to re-import a file resource if it has been changed since adding it to Web Dev.

Text Resource

A text resource is a static resource much like a file resource, except that its contents may be directly edited from within the Ignition Designer.

These are useful for static HTML, CSS, and JavaScript files.

Mounted Folder

Mounted Folders are a way to expose a folder from the gateway's hard drive as a resource endpoint. For example, if the Ignition server had some directories that look liked the following:

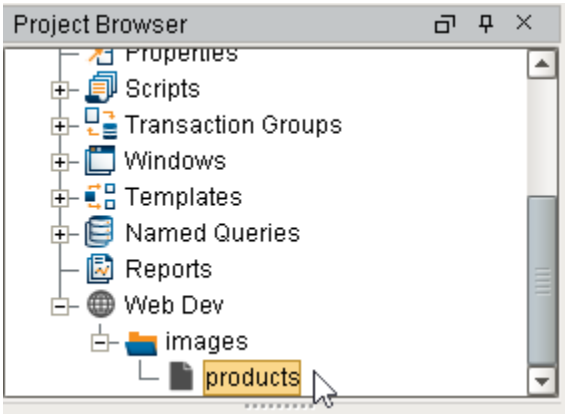
```
/opt/public/a.jpg
/opt/public/info.html
```

You can create a Mounted Folder named "assets", and point it to `/opt/public/`, and then you can access those assets via:

```
http://host:port/main/system/webdev/projectname/assets/a.jpg
```

Project Resources Folders

If a Mounted Folder is placed in a Project Browser folder, then the endpoint must also include the folder name. For example, if the Mounted Folder named "products" is located in the "images" folder:



The files would be accessible via:

```
http://host:port/main/system/webdev/projectname/images/products/a.jpg
```

URL

Each resource will be directly accessible over HTTP and mounted beneath the /main/system/webdev path.

For example, if you created a Text Resource directly beneath the "Web Dev", it would be mounted at:

Pseudocode - Project Resource

```
http://host:port/main/system/webdev/project/resource_name
```

Notice that the project name and resource name are part of the path. If your resource is nested inside a folder, it will be part of the path too. For example:

Pseudocode - Folder Resource

```
http://host:port/main/system/webdev/project/folder_name/resource_name
```

Web Dev resources may have periods in their name. This means that if you upload an image file, you may include its extension directly in its name so that its path is more natural. For example, you might name an image resource "my_image.png" so that its URL is:

Pseudocode - Image Resource

```
http://host:port/main/system/webdev/project/my_image.png
```

The Web Dev module respects the published vs. staging system of Ignition. If your project is set up to have separate staging and published verions, you may access your staging version resources via the URL:

Pseudocode - Staging Version Resource

```
http://host:port/main/system/webdev_staging/project/resource_name
```

Requests to the root of your project will attempt to load a resource named "index.html". If no such resource exists, a 404 response code will be returned instead.

Pseudocode - Request to Root Project

```
http://host:port/main/system/project
```

Python Resources

Python resources are the heart of the functionality of the Web Dev module. These resources are completely dynamic, and can handle all parts of the HTTP protocol, formulating any type of response.

Each time an incoming HTTP request arrives at a URL where a python resource is mounted, the corresponding python script will be run. Each python resource may have up to seven scripts, one for each HTTP method (GET, POST, HEAD, TRACE, etc.). In practice, most python resources will probably only implement one or two of these, usually doGet or doPost at a minimum.

Parameters

Each do* method receives the same two parameters: 'request' and 'session'.

Request Parameter

The request parameter is a dictionary with a bunch of information about the incoming web request.

request['params']

Any URL parameters such as the following:

Pseudocode - Request Parameter

```
/main/system/webdev/project/foo?param1=value&param2=other_value
```

This will be contained in a dictionary accessible via request['params']. For the given example, request['params'] = {'param1':'value', 'param2':'other_value'}

request['remainingPath']

This provides the remaining text after a file resource. If you have a resource called 'foo', and a request is made to:

Pseudocode - Remaining Path

```
/main/system/webdev/project/foo/bar
```

request['remainingPath'] will be "/bar". Remaining path will be "None" if nothing is found after the resource name.

request['postData']

This parameter is only present for the doPost method, and its value is different based upon the value of the incoming HTTP request's Content-Type header. If the content type is 'application/json', then the request['postData'] will be a python dictionary structure which is the result of parsing the json document that was posted. If the content type starts with 'text/', then the value of request['postData'] will be the text which was posted, as a string.

request['headers']

The HTTP request headers will be in a dictionary under request['headers']. For example, you could read the User-Agent string with request['headers']['User-Agent'].

request['scheme']

The scheme is available via request['scheme']. The value will be 'http' or 'https'.

request['remoteAddr'] / request['remoteHost']

These two parameters give the remote IP address and host of the entity that made the web request. Note, that these are from the perspective of the web server, and so may not be what you expect due to the effects of things like NAT-ing routers.

request['servletRequest'] / request['servletResponse']

These two parameters give you direct access to the underlying Java servlet request and response objects.request['context']. This provides direct access to the Ignition GatewayContext. More information about this object may be found in the Ignition SDK developer guide and associated JavaDocs.

request['context']

This provides direct access to the Ignition GatewayContext. More information about this object may be found in the Ignition SDK Programmer's Guide and associated JavaDocs.

Session Parameter

The session parameter is a Python dictionary which may be used to hold information which persists across multiple different calls to your Web Dev Python Resource, or across multiple Python Resources. Session management is handled automatically, and this dictionary is created for each new client session. (Tthe client must have cookies enabled for sessions to work). You may place any key-value pairs in the session dictionary you'd like, just make sure that the values are things that can be serialized. All normal python types (scalar, dictionary, lists, and tuples) are serializable.

Return Value

Each do* function on a python resource must generate some sort of HTTP response. This is done by returning a value from the function. The return value should always be a dictionary, in which the following keys are recognized. The keys are listed here in the order they are evaluated. For example, if you have both file and bytes, only file will take effect. The exception is the contentType key, which may be included with any of the other keys to override the default content type.

File: The value should be a string, which should be the path to a file on the server's filesystem. If no contentType is specified, then the system will attempt to probe the content type from the operating system using java.nio.Files.probeContentType. If the file key is present, but the value points to a file that doesn't exist, an HTTP 404 will be returned.

Bytes: The value should be a byte array. The default content type is application/octet-stream, but you probably want to specify your own.

HTML: The value should be a string, which should be the source of an HTML document. Content type is assumed to be text/html.

JSON: The value is assumed to either be a string (which should be valid json) or to be a python object, which will then be encoded into a json string. Content type will be application/json

Response: If none of the other keys are present, the system will look for the key response which will be stringified and then returned with the content type text/plain.

If your implementation of the do* function returns a dictionary with none of the above keys, an HTTP 500 error will be returned. However, if

you return None, no HTTP 500 error will be returned. In this case, it is assumed that you used therequest['servletResponse'] parameter to directly formulate your HTTP response.

Security Settings

Each python resource can configure its own optional security settings.

Require HTTPS

If this is checked, then the resource will only be accessible via an SSL connection. If a non-secure HTTP transport is used, the browser will be sent a redirect to the the Gateway's SSL port. The Gateway must have SSL enabled, of course.

Require Authentication

If this is checked, the resource will require authentication before it executes. This uses HTTP BASIC auth, and so should really be combined with the Require HTTPS option so that the credentials are encrypted. The username/password combination sent through the HTTP BASIC authentication headers will then be passed through the chosen User Source. If roles are specified, the user must have at least one of the roles. Specify multiple acceptible roles using a comma separated list. If the credentials are missing, an HTTP 401 will be returned with the WWW-Authenticateheader. If the credentials are present but incorrect, an HTTP 403 will be returned.

If the credentials succeed, the python resource will execute. In addition, the authenticated user object returned by the User Source will be accessible inside the session object as session['user']. Since the user is stored in session, if the client has cookies enabled, then further requests against the same session will use the stored user object and will not require additional authentication.

httpPost Example

This example is to allow Ignition to receive data from an external source. It uses a button to send JSON data through an httpPost command and a Python Resource in the Web Dev module to receive the post and do something with the data. This button example is for testing purposes only, the common use-case for posting data is to use another program to post data.

1. Open the Designer and right-click on the Web Dev object in the Project Browser. Create a Python Resource named **postjson.**
2. Select the **doPost** HTTP method from the dropdown in the upper left. Copy this code into the doPost function.

"postjson" Python Resource (Web Dev Section)

```
# take in some JSON data and print it
# expecting 'names' and 'values' that are of the same length

# get the incoming parameters
data = request['postData']
names = data['names']
values = data['values']
# this will print to the wrapper.log file
print names, values

# format the string into HTML
formattedString = "<html><body>"
# loop through and add names and values
for i in range(len(names)):
 formattedString += "%s: %s, " %(names[i], values[i])
# remove the last ', ' and add closing html
formattedString = formattedString[:-2]+"</body></html>"
# this will print to the wrapper.log file
print formattedString

# return the value string
return {'html': formattedString}
```

3. Create a button on a window to test the above code. Copy the code below into your button. Make sure to change the **ProjectName** variable to the name of your project. If you used any name other than "postjson" for step 1, change the **doPostName** variable as well.

Call Web Service (Button component on a window)

```
# post data to the web service in a json format
# this allows you to use the 'postData' object in the Python Resource

# create url to post to
projectName = "MyProject"
doPostName = "postjson"
url = "http://localhost:8088/main/system/webdev/%s/%s" %(projectName, doPostName)
# create the dictionary of parameters to pass in
params = {}
params['names'] = ['String','Integer']
params['values'] = ['Hello World', 42]
# encode dictionary to JSON
jsonParams = system.util.jsonEncode(params)

# post to Ignition
postReturn = system.net.httpPost(url,'application/json',jsonParams)
# print return value
print postReturn
```

4. Now test your button. Make sure to open the console to see the print out, or the wrapper.log file to see any errors caused by the doPost function.

Related Topics ...

- Web Services, SUDS, and REST
- HTTP Methods
- Installing or Upgrading a Module
- Managing Users and Roles