EXTEXP USER GUIDE

Table of Contents

Introduction

CAUTION	Extexp is an end-of-life product. As of 28th May I will no longer using it for website builds. So its no longer a priority for me. I am using JBake as a website builder and asciidoctor to build webpages, documents and books.

WARNING	This documentation was written for a previous version (1.0.0-SNAPSHOT). The latest release of Extexp is 1.1.0.

Extexp is a NetBeans plug-in which can process a command sequence which defines the required text processing. Whilst this task could be undertaken by a command script, Extexp provides an OS independent implementation, with the additional benefits of a simple definition of the command sequences, and use of in-memory storage for intermediate results (if required).

Text processing actions (executors) which are integral to the plug-in include:

- Markdown processing: using markdown sources to create HTML fragments
- Substitution: insert file content/parameter strings into a template file
- FreeMarker Template processing: inserting file context and other fragments
- XSLT processing: XML to XML as directed by an xslt stylesheet
- FOP processing: using a fo-xsl document to create a PDF
- HTML IMG tag creation: scanning image resources to create IMG tags suitable for responsive HTML page (using the srcset attribute)
- Ability to run any OS utility program: providing basic extendibility

In addition, actions (controls) are available to provide simple flow control functions within a command sequence. These controls include:

- If-defined: allows optional processing if a parameter or file is defined/present
- Run: run a command sequence
- Use: similar to run, with minor differences when handling in-memory file system access
- For: repeat an action on all files with the defined extension

Installing / Configuring / Extending Extexp

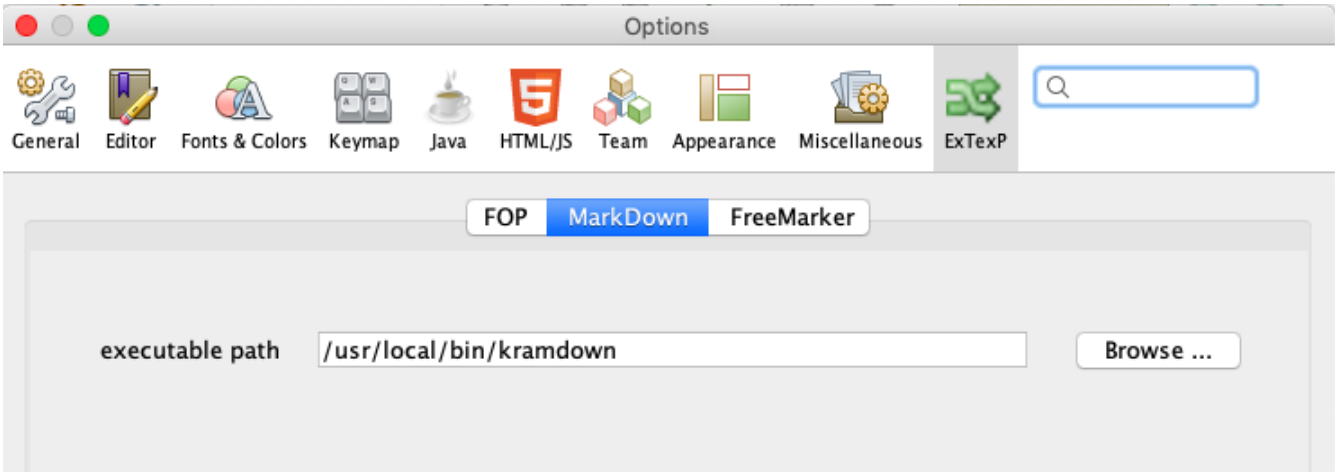
Installing

The NBM can be built from source if required.

Configuring

To enable all Extexp functionality, additional software needs to be installed:

- **Markdown**: it is recommended that the kramdown processor is used to process markdown. Kramdown can be downloaded from it's website. Just follow the instructions on the install tab. Once installed, Extexp needs to know about kramdown's location. By using the NetBeans preferences window and selecting the Extexp options panel and then the markdown tab, the path to the kramdown executable can be defined.



- **FOP**: install the latest version of FOP and then use the same process as was used for markdown (Extexp options panel/FOP tab) in order to define the location of the executable script for FOP.
- **FreeMarker**: while FreeMarker software is already installed, it is necessary to define the root of the users file system where FreeMarker templates are stored. This defaults to a OS specific value ("/" for MacOS and LINUX, or "c:\" for Windows). If this needs to be changed for your workstation, it can be configured in the preferences window (Extexp options panel/FreeMarker tab).

An Extexp Project

Creating an Extexp project

Any Extexp project has the following minimum content within the project directory:

- The Source folder: A folder (src) is used for all input content resources etc. If required, all sources can be stored in this single folder or it can be subdivided into a deeper folder structure.
- The Build file: this file (build.json) defines the MAIN command sequence and any other command sequences. It defines the build execution for this project.

When processing the build file, an output directory will be created in the project directory into which all output will collect. A cache directory at the same level may be created to hold any necessary intermediate values created during the build process.

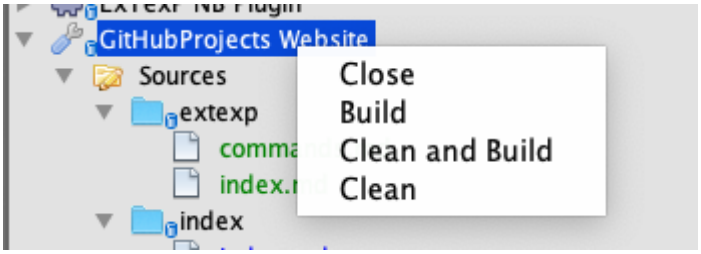
Creating the Extexp Build file

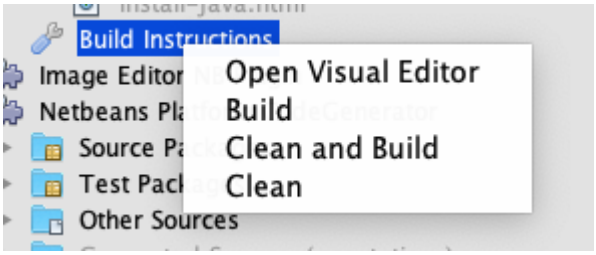
The build file is a Json formatted file which contains one or more command sequences. There must be one sequence named MAIN which is the top-level sequence, other named sequences can be defined which can be called as required.

More details and examples the build file content can be found here.

Executing the Build file

The pop-up menu at the project level and at the build file level have commands to execute the build file.





Both have the following commands:

- Clean: delete all content in the output and cache directories of the project
- Build: execute the build file
- Clean and Build: a clean action followed by build action

Structure of the Build file

The Command sequences are named JSON arrays. The first sequence is named `MAIN` (must always be defined in every build file). Additional sequences can be defined with other names, which can be called when required.

A sequence is a series of JSON objects, which describe the commands to be sequentially undertaken.

These Commands can be Controls or Executors. Controls manage the flow of control through sequences, while Executors are the commands which process textual objects, transforming them in someway.

The command format is:

```
    {
        "<key>": "<value>";     (repeat as necessary}
    }
```

The value can be a:

- Textual value
- In-memory filestore reference(i.e. filename)
- Filename, the execution environment will be used to determine the location context (input v. output)

All these forms of a value can include parameter substitution using ${<key>}, which is replaced with the value of the parameter. Substitutions can be embedded within text, and multiple substitutions are allowed. The substitution process is recursive so allowing complex generation of values (filenames or text).

Details of Controls (syntax and examples) can be found here.

Details of Executors (syntax and examples) can be found here.

The Execution Environment

The execution environment is the information available to an executor during its processing.

The key information in the environment are:

The Parameters

Parameters are set as part of the executors command definition but also inherited from any parent controls.

The input file locations

Input files are selected from one of two locations (folders):

- The first is the content file location, it is initially the source level folder, but can be changed to sub folder (as directed by the `path` or `inputpath` parameters of the `Run` or `Use` command).
- The second is the shared input content file location which is always the source level folder.

The input file selector will initially attempt a filename match in the content file location and, if missing, will then attempt the match in the shared content file location.

The output file location

The output file location (folder) is initially the output folder but can be changed to a child folder (as directed by the `path` parameter of the `Run` or `Use` command).

The in-memory file store

The initial in-memory file store is created as an empty store for the main command sequence. When a `Run` control is executed a new in-memory file system is created for use by the new command sequence. If it is required to switch to a command sequence but to continue to use the caller's in-memory file system then the `Use` control should be used rather than the `Run` control.

Potential action on the in-memory filestore include:

- create new files (prepend the filename with !)
- append to the existing files, creating a file if it does not exist (prepend the filename with +)
- read existing files

Control Commands

Control commands manage the flow of control through the command sequences. Commands exist which can:

- switch to new command sequences
- conditionally execute a command
- repetitively execute a command

Use of these commands may alter the state of the executor environment in which executors will run.

All controls have a parameter with the key set to the control name and a value which is the associated option.

An example of this part of the control is:

```
{
    "Run": "PAGEBUILDER";
    ...
}
```

If-defined

```
{
    "If-defined": "<entity to test>";
    ....
}
```

`If-defined` tests for the existence of:

- a defined parameter
- the existence of a file (found is either normal or in-memory filestore).

It can have up to two optional parameters `then` and `else`, each of which can have a single command as its parameter value. The `then` command will be executed if the parameter is defined, the `else` command being executed if the parameter is not defined.

The `then` and `else` commands are executed using the current Execution Environment i.e. the command makes no changes to environment in preparation for execution,

If you want to execute a sequence of commands for either the `then` or `else` condition, then the required sequence should be defined as a named sequence and either a `Use` or `Run` command inserted as the `then` or `else` parameter value.

Example:

```
    {
        "If-defined": "topinsert.md";
        "then": {
                    "Do": "markdown";
                    "from": "topinsert.md";
                    "to": "!topinsert";
                };
    }
```

Run

```
    {
        "Run": "<named command sequence>";
        ...
    }
```

`Run` executes a named command sequence. It updates the execution environment prior to executing the command sequence. Parameters `path` or `inputpath` can be used to update the input and output file contexts, and a new empty in-memory filestore is created.

Parameter `path` will cause both the input and output file context to be updated, while parameter `inputpath` will only update the input file context.

Additional parameters can be defined as needed. These parameters are available to the called command sequence, so provide a method of passing information.

Example:

```
    {
        "Run": "PAGEBUILDER",
        "title": "Java Installation Notes",
        "inputpath": "index",
        "page": "install-java",
        "navmenu": "Home>index"
    }
```

Use

```
    {
        "Use": "<named command sequence>";
```

```
    ...
}
```

`Use` executes a named command sequence. It is very similar to the `Run` control, the only difference being that the `Use` control does not create a new in-memory file store. This allows a `Use` command sequence to return information via the in-memory file store.

Example:

```
{
    "Use": "PAGEBUILDER",
    "title": "Java Installation Notes",
    "inputpath": "index",
    "page": "install-java",
    "navmenu": "Home>index"
}chapter7/
```

For

```
{
    "For": "<ext>",
    "do": <command>;
    .....
}
```

`For` executes a command repetitively for each file in the current input context with the defined file extension. The `do` parameter defines the command to be executed. During execution of the command a special parameter `FILENAME` will be defined with a value which is the selected filename.

Example:

```
{
    "For": "md",
    "do": {
        "Run": "PAGEBUILDER",
        "page": "${__FILENAME__}"
    }
}
```

Executor Commands

Executors are the set of commands which undertake the text transformations.

Whilst Extexp is designed as an extendable text processor and can have additional executors added, there are a standard set of executor commands present in the extexp plug-in.

All executors commands have a parameter with the key `Do` and a value which is the executor name.

An example of this part of the executor command is:

```
{
    "Do": "copy";
    ...
}
```

Copy

The `Copy` command copies text to the target file.

Example:

```
{
    "Do": "copy";
    "from": "example.md";
    "to": "!example";
}
```

Copy resources

The `Copy Resources` command copies contents of resource folder to the target folder.

It has two optional parameters `to` and `foldername`.

- `to` defines the target folder and is one of two values `output` or `cache`. The parameter is optional, with its default being `output`.
- "foldername" describes the name of the resources folder. The parameter is optional with its default being "resources".

The folder structure for resource input will allow optional child folder structures within the `resources` folder, but the resulting created `resources` folder will be flat structure.

The folder structure between the project source folder and the input resources folder will be

replicated between the source folder and the output resources folder.

Examples:

```
    {
        "Do": "copyresources";
    }
```

```
or
```

```
    {
        "Do": "copyresources";
        "to": "cache";
        "foldername": "images";
    }
```

List

The `List` command copies the source to the output window. The `title` parameter defines the displayed title and `from` parameter defines the source.

Example:

```
    {
        "Do": "list";
        "from": "example.md";
        "title": "Example ONE";
    }
```

which displays:

```
====================================
Example ONE
====================================
file content
....
====================================
```

Message

The `Message` command writes a message to the output window. The `text` parameter defines the displayed message.

Example:

```
    {
        "Do": "message";
        "text": "Processing page ${page}";
    }
```

displays (where parameter `page` has the value "index"):

```
Processing page index
```

Substitute

The substitute command processes a template (usually a file) performing parameter substitution on all ${<key>} markers. The substitutions can be parameter values, in-memory file content, files content (from the input and shared input contexts). All insertions have parameter substitution performed on their content (i.e. the substitution process is fully recursive).

The command has two parameters:

- `from` - the template file
- `to` - the results of the substitution process

Additional parameters may be defined in the command which will be available for use in the substitute process (as well as all parameters defined in parent controls).

Example:

```
    {
        "Do": "substitute";
        "from": "simple-template.txt",
        "to": "+generated-text";
    }
```

FOP

XSL-FO is a W3C standard describing a XML vocabulary which defined printed pages. Tools exist

which can process these documents and create PDF output. There is an Apache open source tool (FOP) which can be used for this purpose. This commands runs the FOP tool.

The command has two parameters:

- `xsl-fo` - the xsl-fo document
- `pdf` - the resulting pdf document

This commands requires that the FOP software is installed on your machine and the any necessary configuration is undertaken via the NetBeans Preference screen. For more details about this please see Configuring Extexp.

Example:

```
{
    "Do": "fop";
    "xsl-fo": "book",
    "to": "Book.pdf";
}
```

External

This command will run any simple text processing program that is available on your workstation. It required addition parameters:

- `command` - the command name to be executed
- `parameters` - any additional command parameters that are required (optional)
- `from` - the input text for the command
- `to` - the output file to collect the output from the command

The error stream from the external program will be written to the netbeans output window.

If using Linux or MacOS, programs such as grep, sed and awk are text processing programs which could be executed by this command to access additional specialist text processing capabilities.

Example:

```
{
    "Do": "external";
    "command": "wc",
    "parameters": "-w",
    "from": "chapter2.md",
    "to": "wc-param"
}
```

Markdown

Markdown is a lightweight markup language with plain text formatting syntax. This command processes the markdown document and process html output (which is probably a html fragment for later insertion into a full html document).

There are a range of markdown processors available, and whilst it is technically possible to use any such software, the extexp project recommends use of the kramdown processor.

The command requires up to three parameters:

- `from` - the markdown source file
- `to` - the generated html fragment
- `template` - The name of an ERB template file (optional)

At its simplest, the ERB template file provides a wrapper around the generated source. The point in the template where the processed markdown is to be inserted is marked with a tag `<%=@body%>`.

However it is more normal practise within Extexp to use the markdown command to generate a html fragment, and then combine all the fragments into a final html document using either the Substitute or FreeMarker commands (i.e. not using the markdown template parameter), but the software can support either styles.

This commands requires that the kramdown software is installed on your machine and the any necessary configuration is undertaken via the NetBeans Preference screen. For more details about this please see Configuring Extexp.

Example:

```
{
    "Do": "markdown";
    "from": "chapter2.md",
    "to": "!chapter2"
}
```

Markdown and Substitute

The Markdown And Substitute command (`markdown-substitute`) is a convenience command combining the Markdown and Substitute commands. It has the same parameters as the markdown command, first processing the markdown text, after which it runs the substitution process on the generated output before saving it (the `to` parameter).

Additional parameters may be defined in the command which will be available foe use in the substitute process (as well as all parameters defined in parent controls).

Example:

```
    {
        "Do": "markdown-substitute";
        "from": "chapter2.md",
        "to": "!chapter2",
        "title": "ABC ... XYZ"
    }
```

FreeMarker

FreeMarker provides a more fully featured templating language for building complex documents. All parameters and in-memory filestore is made available to the FreeMarker engine, and files can be made available, by declaration as a parameter value.

Two parameters are required:

- `template` - the Freemarker template
- `to` - the output document generated by FreeMarker

Additional parameters may be defined in the command which will be available for use by FreeMarker (as well as all parameters defined in parent controls).

While the FreeMarker software is already installed on your workstation, there is a small amount of necessary configuration to be undertaken via the NetBeans Preference screen. For more details about this please see Configuring Extexp.

Example:

```
    {
        "Do": "freemarker";
        "template": "main.ftl",
        "to": "!chapter2",
        "title": "Chapter 2 - Getting to know your Software",
        "content": "content-fragment-ch2"
    }
```

Imageset

The Imageset command (`create-imageset`) collects together information about a set of like images (each being a different image size) and creates a suitable \<IMG> html tag. Such a tag would be used to display an image in a responsive design and would be inserted into an html document or fragment (using a templating or substitution method provided within Extexp).

The command has 5 required parameters:

- `image` - the image name for the 'core image'
- `width` - the width of the core image
- `height` - the height of the core image
- `class` - the css class name to be added to the imageset tag
- `to` - the html output created by the command.

Example:

```
{
    "Do": "create-imageset",
    "to": "!swimming-3",
    "class": "img-responsive",
    "image": "Swimming-3.png",
    "width": "500",
    "height": "300"
}
```

which creates content:

```
<img
    width="500"
    height="300"
    src="resources/Swimming-3.jpg"
    class="img-responsive" alt=""
    srcset="resources/Swimming-3.jpg 500w,
        resources/Swimming-3-300x180.jpg 300w,
        resources/Swimming-3-200x120.jpg 200w,
        resources/Swimming-3-60x36.jpg 60w"
        sizes="(max-width: 500px) 100vw, 500px"
/>
```

XSLT

XLST is a W3C standard describing a language for defining transforms that can be applied to a XML document to produce an output (normally another XML document). As HTML5 is an XML subset, it can be used to transform html documents or fragments.

The command requires 3 parameters:

- `from` - the source XML (html) document
- `stylesheet` - the file defining the transformations to be applied (xslt language)
- `to` - the output document created by the xslt

Example of use: to transform a html document (probably generated from markdown sources) into an xsl-fo document for passing to FOP to build a pdf document.

Example:

```
    {
        "Do": "xslt",
        "from": "fullbook",
        "stylesheet": "bookbuilder.xsl",
        "to": "!fo"
    }
```

Developer Notes

The source code is available on GitHub.

The build/dependency manangement system is Maven.

The development environment uses the Netbeans IDE, but this is not mandatory, any development environment which can work with a Maven project can be used.

Source Code

The current development version is 1.0.0-SNAPSHOT.

The project can be forked or cloned from here to obtain the whole source tree.