Rendering content

Rendering is the process of merging templates with content. It generates a meaningful stream of HTML that can be displayed in a browser. This is the step where content becomes visible to the end user in page context.

Steps involved in rendering content

- **Steps involved in rendering content**
- **Renderers**
- **Rendering order**
- **Early execution of components**
- **Direct component rendering**
- **Direct area rendering**
- **Populating the rendering model with request parameters**
  - Enabling the population of request parameters
  - Processing the parameter in a model class

Rendering is performed by the RenderingEngine. The engine generates HTML output based on templates. The RenderingFilter that is part of the filter chain submits a content node to the engine.

Since submission of content is handled through a filter, an independent servlet or Java application can also submit content to the Magnolia rendering engine (FreeMarker only). You can trigger the rendering of any content node from Java code or template script. This is useful if you want to render a node that is not part of the page structure.

Advanced topic

It is not necessary to use a rendering model in Magnolia templating, but it is useful. Before you create your own model class, see if Magnolia's templating functions have the logic you need. They may save you some effort.

Steps involved in rendering content

On submission of content to the engine, the following process kicks off:

1. **Read template definition**: The rendering engine reads the template definition assigned to the content. The template definition provides the engine with details such as a path to the templateScript and the templating language used (FreeMarker).
2. **Select renderer**: Based on the templating language, the engine selects a renderer. Magnolia ships with two renderers FreemarkerRenderer and SiteAwareFreemarkerRenderer (Git). You can also use your own custom renderer.
3. **Execute renderer**: The renderer is executed. If the template definition specifies a model class then the renderer instantiates the model object from this class. The model is a JavaBean. This is where your business logic resides. Before the renderer calls the script, the model's execute() method is called. The model can execute any Java code to retrieve data from the JCR repository or from an external source. The return value of the execute method is passed to the template script with the actionResult object.
4. **Call template script**: The renderer calls the assigned template script and passes the current content, the model and all context objects to it. FreeMarker interpolations like $(content.title) are replaced with actual values and directives like #list are processed. The output of the merge process is an HTML stream response that can be displayed in a browser.
Renderers

A renderer is a Java class library that understands a particular templating language and produces text output. For example, `FreemarkerRenderer` understands FreeMarker tags and produces output in HTML, XML, RTF, Java source code and a few other formats. A renderer is not an end-user application but rather something programmers can embed into their products. You can register a renderer in your own module configuration under a `renderers` folder. See Renderers for more.

Any renderer that knows how to call a Java method can theoretically be used to render authoring UI components. Velocity (similar to FreeMarker but lighter) and ColdFusion are examples of popular templating languages that have been implemented successfully.

Rendering order

Rendering of page content happens in a top-to-bottom fashion. Rendered HTML is sent to the output stream as soon as it gets rendered.

When a page template script calls an area, and the area in turn calls a component, the rendering engine follows referenced scripts down the rabbit hole until it finds the last script in the sequence. It then moves to the next element on the page, starting where it left off.

As a consequence, you cannot redirect in a component model as the request was already committed when the rendering of the page started. Any operations changing the HTTP response headers (redirects, setting cookies...) can only be executed in the page template’s model.

Early execution of components

Sometimes components need to be able to execute logic before page rendering starts. This is necessary for operations such as forwarding and redirecting.

Early execution is implemented using `ModelExecutionFilter`. This filter comes before page rendering in the Magnolia `filter chain`. A special request parameter `mgnlModelExecutionUUID` contains the UUID of the component that should be executed. The component model can decide if it wants to output something special or continue with rendering normally. If the model implements `EarlyExecutionAware` interface, then the `executeEarly` method is called. Otherwise the normal `execute` method is called.
Possible special instructions are:

- **skip-rendering** causes the rendering engine to abort page rendering. The template is never rendered.
- **redirect:<url>** sends a redirect HTTP header to the client and stops page rendering. The URL can be absolute or relative inside the Web application, that is, the context path is added by Magnolia.
- **permanent:<url>** sends a permanent redirect (301) to the client and stops page rendering. The URL can be absolute or relative.
- **forward:<url>** performs a forward to the returned URL. The URL can be absolute or relative.

If the component model decides to continue normally, the already-executed model and its `actionResult` are used instead of executing the model again.

**Early execution proceeds like this:**

1. ModelExecutionFilter filter is configured before page RenderingFilter in the Magnolia filter chain. This filter listens for the request parameter `mgnlModelExecutionUUID` which contains the UUID of a component that should be executed early.
2. The filter calls the `execute` method of the component's rendering model.
3. The filter checks the returned value for special instructions and performs them. If the result is not a special instruction, the filter stores the returned `actionResult` and the model as request attributes and passes them on to the rendering pipeline. The request attribute is stored with a name that includes the UUID.
4. AbstractRenderer checks for attributes set by the filter for the component it is about to render. If they are present their values are used, otherwise the model is executed normally.

**Direct component rendering**

You can request a component to be rendered directly by URL. This is useful when rendering Magnolia content in an external system, for example. The URL pattern follows the storage path in the JCR. For example, the first component in any area on your home page resides at path `/my-site/<area name>/0` in the JCR.

Request the component using the path in the URL:

| Magnolia 5 | × http://localhost:el/main/0.html × | +

| localhost:8080/magnoliaPublic/travel/main/0.html |

1.  
2.  
3.  

**Hut to Hut in the Swiss Alps**
Direct area rendering

Direct area rendering allows you to request any area that has a node in the JCR.

To request an area, insert the \texttt{mgnlArea} selector into the URL after the page name but before the .html extension. Set the value to the area name.

Example:

\texttt{~mgnlArea=<area name>~}

To request a nested area, add the path to the nested area separated by commas.

Example:

\texttt{~mgnlArea=<<main area name>,<nested area name>>~}

Direct area rendering is achieved using a rendering listener registered in \texttt{/server/rendering/engine/listeners/directAreaRendering}.

<table>
<thead>
<tr>
<th>Node name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td></td>
</tr>
<tr>
<td>rendering</td>
<td></td>
</tr>
<tr>
<td>engine</td>
<td></td>
</tr>
<tr>
<td>listeners</td>
<td></td>
</tr>
<tr>
<td>directAreaRendering</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>info.magnolia.rendering.listeners.AreaFilteringListener</td>
</tr>
</tbody>
</table>

Properties:

- **class**: AreaFilteringListener is an area rendering listener for enabling/disabling output according to rendered content.

Here's main area of the /travel/about page rendered by the URL http://localhost:8080/magnoliaAuthor/travel/about~mgnlArea=main~.html.

About Magnolia Travels

We are a full service, independent travel agency.

We offer unique tours from every continent on the planet. Get inspired and book your tour w

Learn more

Notes:

- If an area does not exist, the whole page is rendered. For example http://localhost:8080/magnoliaAuthor/your-site/about~mgnlArea=NonExistingArea~.html will return 404.
- Where an area exists but contains no content an empty page is rendered.
Populating the rendering model with request parameters

You can populate the RenderingModel with request parameters. This is useful for reacting to URL information. For example, identify a visitor who comes from an email campaign and execute custom rendering logic for them. Here is an example of parameter that tells you the visitor clicked a link in a Campaign Monitor email campaign.

http://www.your-site.com/about.html?utm_source=CampaignMonitor

Enabling the population of request parameters

You can enable and disable the population of request parameters globally for all templates or for a specific template.

If you cannot control the template, you can also do this for a specific component definition.

Globally:

1. Go to Configuration > /server/rendering/engine
2. Create a new property autoPopulateFromRequest and set the value to true or false. Default is true.

For a specific template definition, overriding the global setting:

1. Go to Configuration > /modules/<your-module>/templates/pages/<template definition>
2. Create a new property autoPopulateFromRequest and set the value to true or false. Default is null. This means the global setting applies if you don't set a value.

For a specific component definition, overriding the global setting:

1. Go to Configuration > /modules/<your-module>/templates/components/<component_definition>
2. Create a new property autoPopulateFromRequest and set the value to true or false. Default is null. This means the global setting applies if you don't set a value.

Processing the parameter in a model class

In your model class, implement a setSomeParameter(String) method. The method will be called with the attribute value as the parameter.

```java
class MyModel implements RenderingModel {
    private String someParameter;
    public void setSomeParameter(String someParameter) {
        this.someParameter = someParameter;
    }
    public String getSomeParameter() {
        return someParameter;
    }
}
```

Populating the rendering model with request attributes impacts performance. The rendering calculation takes about 8% longer when population is enabled. Enable it only for templates where it is really needed.