Deep Dive into Sightly, an open source templating language

@GabrielWalt, Product Manager

Specification and TCK open sourced to GitHub.
Reference implementation donated to Apache Sling.

Follow @sightlyio on Twitter.
http://docs.adobe.com/docs/en/aem/6-2/develop/sightly.html

Adobe

§ 1 Expression contexts

§ 2 Passing data to client libs

§ 3 Use statement

§ 4 Template & Call statements

§ 5 Parameters for sub-resources

Display Context Option

The context option offers control over escaping and XSS protection.

Allowing some HTML markup (filtering out scripts)

```
<div>${properties.jcr:description @ context='html'}</div>
```

Adding URI validation protection to other attributes than `src` or `href`

```
<p data-link="${link @ context='uri'}">text</p>
```

Display Context Option

```
<a href="${myLink}" title="${myTitle}">${myContent}</a>
<script> var foo = "${myVar @ context='scriptString'}"; </string>
<style> a { font-family: "${myFont @ context='styleString'}"; } </style>
```

Most useful contexts and what they do:

safer ↑

`number`	`XSSAPI.getValidNumber`
`uri`	`XSSAPI.getValidHref` (default for `src` and `href` attributes)
`attribute`	`XSSAPI.encodeForHTMLAttribute` (default for other attributes)
`text`	`XSSAPI.encodeForHTML` (default for element content)
`scriptString`	`XSSAPI.encodeForJSString`
`styleString`	`XSSAPI.encodeForCSSString`
`html`	`XSSAPI.filterHTML`
`unsafe`	disables all protection, use at your own risk.

Display Context Option

```
<a href="${myLink}" title="${myTitle}">${myContent}</a>
<script> var foo = "${myVar @ context='scriptString'}"; </string>
<style> a { font-family: "${myFont @ context='styleString'}"; } </style>
```

Preferred method for each context:

- `src` and `href` attributes: `number, uri, attribute, unsafe`
- other attributes: `number, uri, attribute, unsafe`
- element content: `number, text, html, unsafe`
- JS scripts ⊛: `number, uri, scriptString, unsafe`
- CSS styles ⊛: `number, uri, styleString, unsafe`

⊛ An explicit context is required for script and style contexts.

Don't set the context manually unless you understand what you are doing.

§ 1 Expression contexts

§ 2 Passing data to client libs

§ 3 Use statement

§ 4 Template & Call statements

§ 5 Parameters for sub-resources

Passing data to client libs

```
<!-- template.html -->
<div data-sly-use.logic="logic.js"
     data-json="${logic.json}"></div>

/* logic.js */
use(function () {
    return {
        json: JSON.stringify({
            foo: "bar",
            arr: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        })
    };
});
```

Server-side JavaScript logic

```
<!-- template.html -->
<script id="my-handlebar-template"
        type="text/x-handlebars-template"
        data-sly-include="handlebar.html"></script>

<!-- handlebar.html -->
<p>${properties.jcr:title}</p>
```

Passing data to client libraries

http://bit.ly/sightly-data-json

http://bit.ly/sightly-script-angular

Adobe

§ 1 Expression contexts

§ 2 Passing data to client libs

§ 3 Use statement

§ 4 Template & Call statements

§ 5 Parameters for sub-resources

Use Statement

Initializes a helper object.

```
<div data-sly-use.logic="logic.js">${logic.hi}</div>
```

Output:

```
<div>Hello World</div>
```

Server-side JavaScript logic

```
<!-- template.html -->
<div data-sly-use.logic="logic.js">${logic.hi}</div>

/* logic.js */
use(function () {
    return {
        hi: "Hello World"
    };
});
```

Like for the Sightly template, the objects available in the logic file are the same ones as in JSP with global.jsp

Java logic

POJO extending WCMUse

```
<!-- template.html -->
<div data-sly-use.logic="Logic">${logic.hi}</div>

/* Logic.java in component */
package apps.my_site.components.my_component;
import org.sling…sightly.WCMUsePojo;

public class Logic extends WCMUsePojo {
    private String hi;

    @Override
    public void activate() throws Exception {
        hi = "Hello World";
    }

    public String getHi() {
        return hi;
    }
}
```

When the Java files are located in the content repository, next to the Sightly template, only the class name is needed.

Java logic

Adaptable with SlingModels

```
<!-- template.html -->
<div data-sly-use.logic="com.foo.Logic">${logic.hi}</div>

/* Logic.java in OSGi bundle */
package com.foo;
import javax.annotation.PostConstruct;
import org.apache.sling.api.resource.Resource;
import org.apache.sling.models.annotations.Model;

@Model(adaptables = Resource.class)
public class Logic {
    private String hi;

    @PostConstruct
    protected void init() {
        hi = "Hello World";
    }

    public String getHi() {
        return hi;
    }
}
```

When embedded in an OSGi bundle, the fully qualified Java class name is needed.

The Use-API accepts classes that are adaptable from Resource or Request.

What kind of Use-API?

Model logic

This logic is not tied to a template and is potentially reusable among components.
It should aim to form a stable API that changes little, even in case of a full redesign.

➔ **Java located in OSGi bundle**

View logic

This logic is specific to the templates and is likely to change if the design changes.
It is thus a good practice to locate it in the content repository, next to the template.

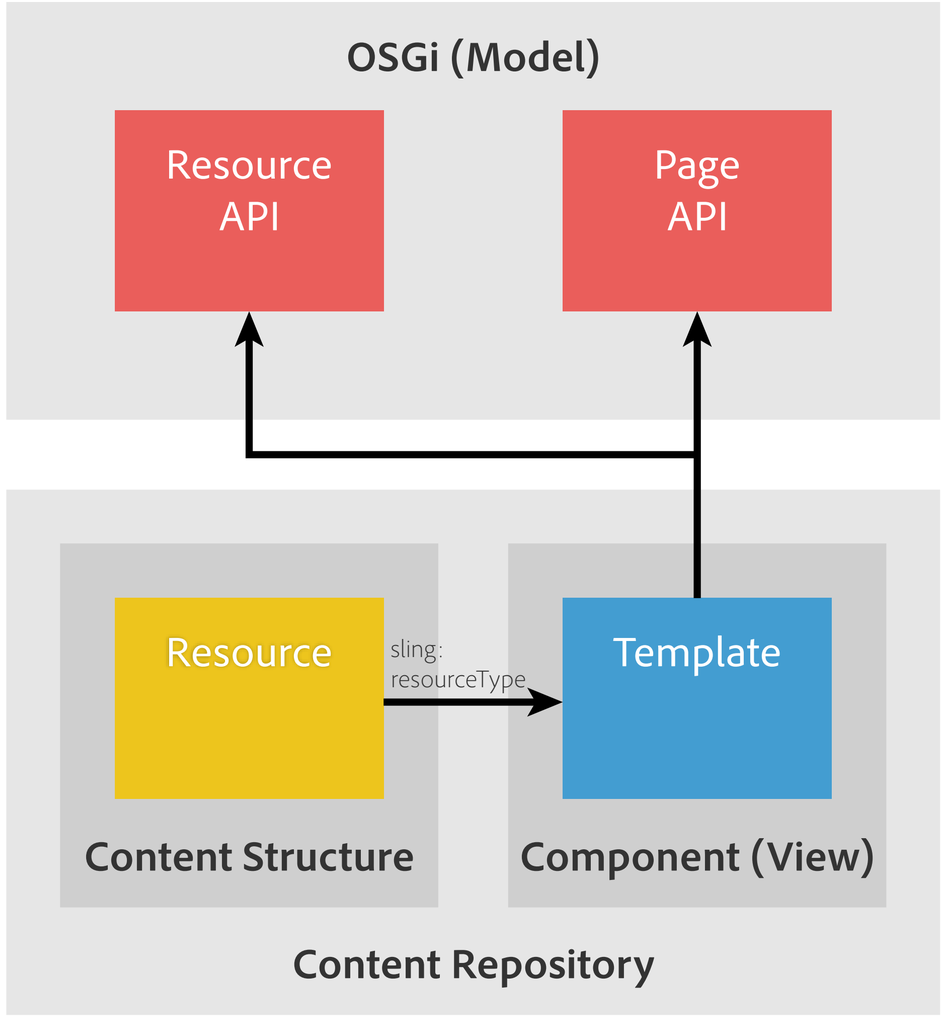
➔ **JavaScript located in component**

If components are to be maintained by front-end devs (typically with Brackets).

➔ **Java located in component**

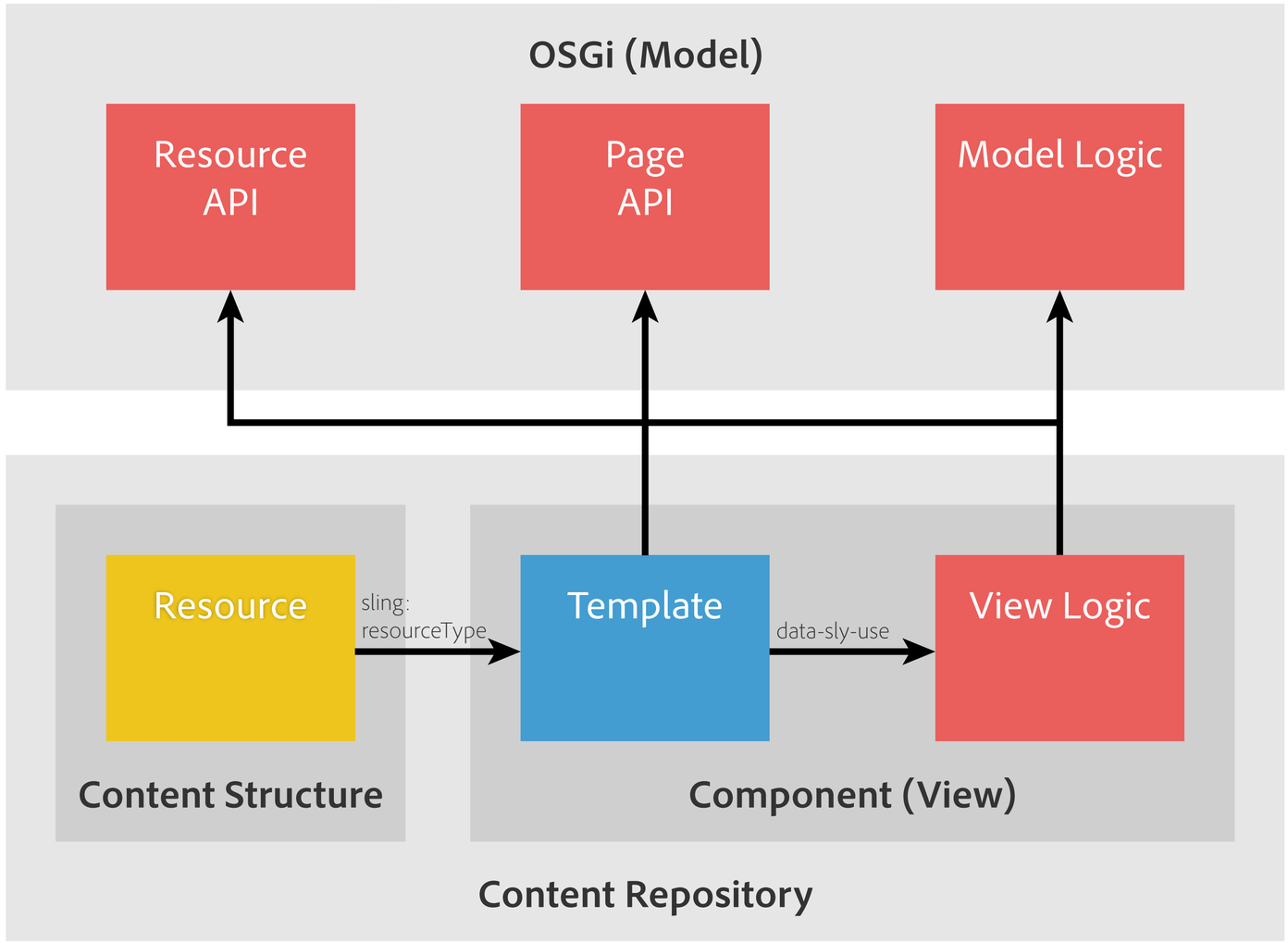
If performance is critical (e.g. when many requests are not cached by the dispatcher).

Start simple: first, no code!



- Sling plays the role of the **controller** and resolves the sling:resourceType, deciding which component will render the accessed resource.
- The component plays the role of the **view** and it's Sightly template builds the corresponding markup.
- The Resource and Page APIs play the role of the **model**, which are available from the template as variables.

Add logic only where needed



- **Model Logic** is needed only if the logic to access the data is different to what existing APIs provide.
- **View Logic** is needed only when the template needs additional data preparation.

Use-API Bindings

The logic can access the same variables than exist in the template.

JavaScript:

```
var title = properties.get("title");
```

Java extending WCMUse:

```
String title = getProperties().get("title", String.class);
```

Java with SlingModels:

```
@Inject @Optional
private String title;
```

Use-API Parameters

With the same notation as for template parameters, named parameters can be passed to the Use-API.

```
<a data-sly-use.ext="${'Externalize' @ path='page.html'}"
    href="${ext.absolutePath}">link</a>
```

Output:

```
<a href="/absolute/path/to/page.html">link</a>
```

Don't pass variables that are part of the global binding (like `properties` or `resource`) as they can be accessed from the logic too.

Use-API Parameters

These parameters can then be read in from the various Use-API.

JavaScript:

```
var path = this.path;
```

Java extending WCMUse:

```
String path = get("path", String.class);
```

Java with SlingModels (works only when adapting from Request):

```
@Inject @Optional
private String path;
```

Use with Template & Call

The use statement can also load `data-sly-template` markup snippets located in other files.

```
<!-- library.html -->
<template data-sly-template.foo="${@ text}">
   <span class="example">${text}</span>
</template>

<!-- template.html -->
<div data-sly-use.library="library.html"
     data-sly-call="${library.foo @ text='Hi'}"></div>
```

Output:

```
<div><span class="example">Hi</span></div>
```

Adobe

§ 1 Expression contexts

§ 2 Passing data to client libs

§ 3 Use statement

§ 4 Template & Call statements

§ 5 Parameters for sub-resources

Template & Call Statements

Declaring template name

Defining template parameters

Declare and call a markup snippet with named parameters.

```
<template data-sly-template.foo="${@ class, text}">
   <span class="${class}">${text}</span>
</template>

<div data-sly-call="${foo @ class='example',
                           text='Hi'}"></div>
```

Template content

Calling template by name

Passing named parameters

Output:

```
<div><span class="example">Hi</span></div>
```

Template & Call Statements

Advanced example of a recursive site map with template, call and list.

```
<ol data-sly-template.listChildren="${@ page}"
    data-sly-list="${page.listChildren}">
    <li>
        <div class="title">${item.title}</div>
        <ol data-sly-call="${listChildren @ page=item}"></ol>
    </li>
</ol>

<ol data-sly-call="${listChildren @ page=currentPage}"></ol>
```

Adobe

§ 1 Expression contexts

§ 2 Passing data to client libs

§ 3 Use statement

§ 4 Template & Call statements

§ 5 Parameters for sub-resources

Parameters for sub-resources

http://bit.ly/sightly-attributes

http://bit.ly/sightly-synthetic

Thank you!

@GabrielWalt, Product Manager

- **Documentation**
  https://docs.adobe.com/docs/en/aem/6-2/develop/sightly.html
- **Specification**
  https://github.com/Adobe-Marketing-Cloud/sightly-spec
- **REPL** (Read-Eval-Print Loop) live Sightly execution environment
  https://github.com/Adobe-Marketing-Cloud/aem-sightly-repl

Adobe