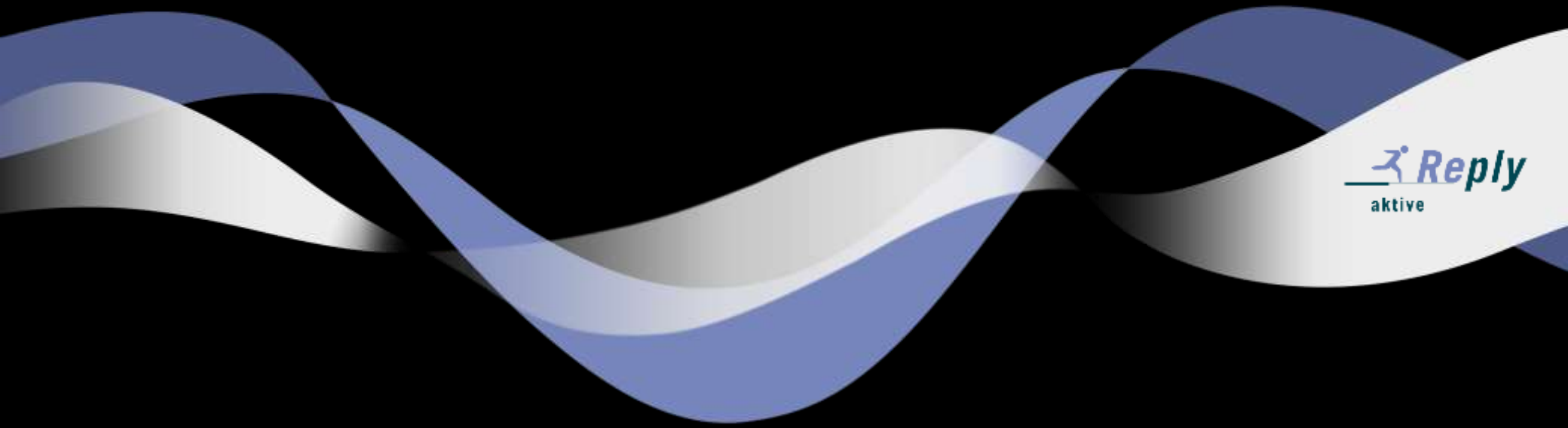
Introduction to Sightly and Sling Models in AEM6

Stefano Celentano



Summary

- What is Sightly?
- Why Sightly?
  - Separation of concerns
  - Comparing JSP and Sightly development
  - Sightly features
- Sightly: the basics of the language
- Not a silver bullet
- Sightly and Sling Models
  - An amazing combination
  - Basic usage
  - Learning from experience
  - wcm-io and ACS Commons libraries
- Useful links
- A few hands-on examples

What is Sightly

- **New secure server-side HTML Templating Language introduced with AEM6 for efficient development**
- “Sightly” (meaning “pleasing to the eye”)
- “Alternative” to JSP
- Specification and TCK open sourced to GitHub
- Reference Adobe implementation donated to Apache Sling by Adobe (September 2014)
- Sightly reference implementation has been folded into Apache Sling project
  - org.apache.sling.scripting.sightly
  - org.apache.sling.xss

Why Sightly: before…

JSP

```
<%@ page session="false"
    import="java.util.Date,
            java.text.SimpleDateFormat,
            org.apache.commons.lang3.StringUtils" %><%
%><%@include file="/libs/foundation/global.jsp" %><%

    // Set last-modified date
    String lastModified = null;
    if(!properties.get("cq:lastModified", "").equals("")) {
        SimpleDateFormat sdf = new SimpleDateFormat("d MMM yyyy HH:mm:ss z");
        lastModified = sdf.format(properties.get("cq:lastModified", Date.class) );
    }

    // Set favIcon
    String favIcon = currentDesign.getPath() + "/favicon.ico";
    if(resourceResolver.getResource(favIcon) == null) {
        favIcon = null;
    }

%>
<head>
    <meta charset="utf-8">
    <title><%= currentPage.getTitle() == null ? xssAPI.encodeForHTML(currentPage.getName()) : xssAPI.encodeForHTML(currentPage.getTitle()) %></title>
    <cq:include script="/libs/wcm/core/components/init/init.jsp" />
    <cq:includeClientLib categories="apps.geometrixx-main" />

    <meta name="description" content="<%= xssAPI.encodeForHTMLAttr(properties.get("jcr:description", "")) %>" />
    <meta name="keywords" content="<%= xssAPI.encodeForHTMLAttr(WCMUtils.getKeywords(currentPage, false)) %>" />

<% if(properties.get("cq:tags", new String[0]).length > 0) { %>
    <meta name="tags" content="<%= xssAPI.encodeForHTMLAttr( StringUtils.join(properties.get("cq:tags", new String[0]), ",") ) %>" />
<% } %>
<% if(!properties.get("subtitle", "").equals("")) { %>
    <meta name="subtitle" content="<%= xssAPI.encodeForHTMLAttr(properties.get("subtitle", "")) %>" />
<% } %>
<% if(!properties.get("cq:lastModifiedBy", "").equals("")) { %>
    <meta name="author" content="<%= xssAPI.encodeForHTMLAttr(properties.get("cq:lastModifiedBy", "")) %>" />
<% } %>

<% if(lastModified != null) { %>
    <meta http-equiv="last-modified" content="<%= xssAPI.encodeForHTMLAttr(lastModified) %>" />
<% } %>

<% if(favIcon != null) { %>
    <link rel="icon" type="image/vnd.microsoft.icon" href="<%= xssAPI.getValidHref(favIcon) %>" />
    <link rel="shortcut icon" type="image/vnd.microsoft.icon" href="<%= xssAPI.getValidHref(favIcon) %>" />
<% } %>
```

Why Sightly: …after!

HTML

```
<head data-block-use="Component">
    <meta charset="utf-8">
    <title>${currentPage.title OR currentPage.name}</title>

    <script data-block-include="/libs/wcm/core/components/init/init.jsp"></script>
    <script data-block-clientlib="apps.geometrixx-main"></script>

    <meta name="description" content="${properties.jcr:description}" />
    <meta name="keywords" content="${component.keywords}" />
    <meta name="tags" content="${component.tags}" data-block-test="${component.tags}" />
    <meta name="subtitle" content="${properties.subtitle}" data-block-test="${properties.subtitle}" />
    <meta name="author" content="${properties.cq:lastModifiedBy}" data-block-test="${properties.cq:lastModifiedBy}" />
    <meta http-equiv="last-modified" content="${component.lastModified}" data-block-test="${component.lastModified}" />

    <link rel="icon" type="image/vnd.microsoft.icon" href="${component.favIcon}" data-block-test="${component.favIcon}" />
    <link rel="shortcut icon" type="image/vnd.microsoft.icon" href="${component.favIcon}" data-block-test="${component.favIcon}" />
</head>
```

Why Sightly: separation of concerns

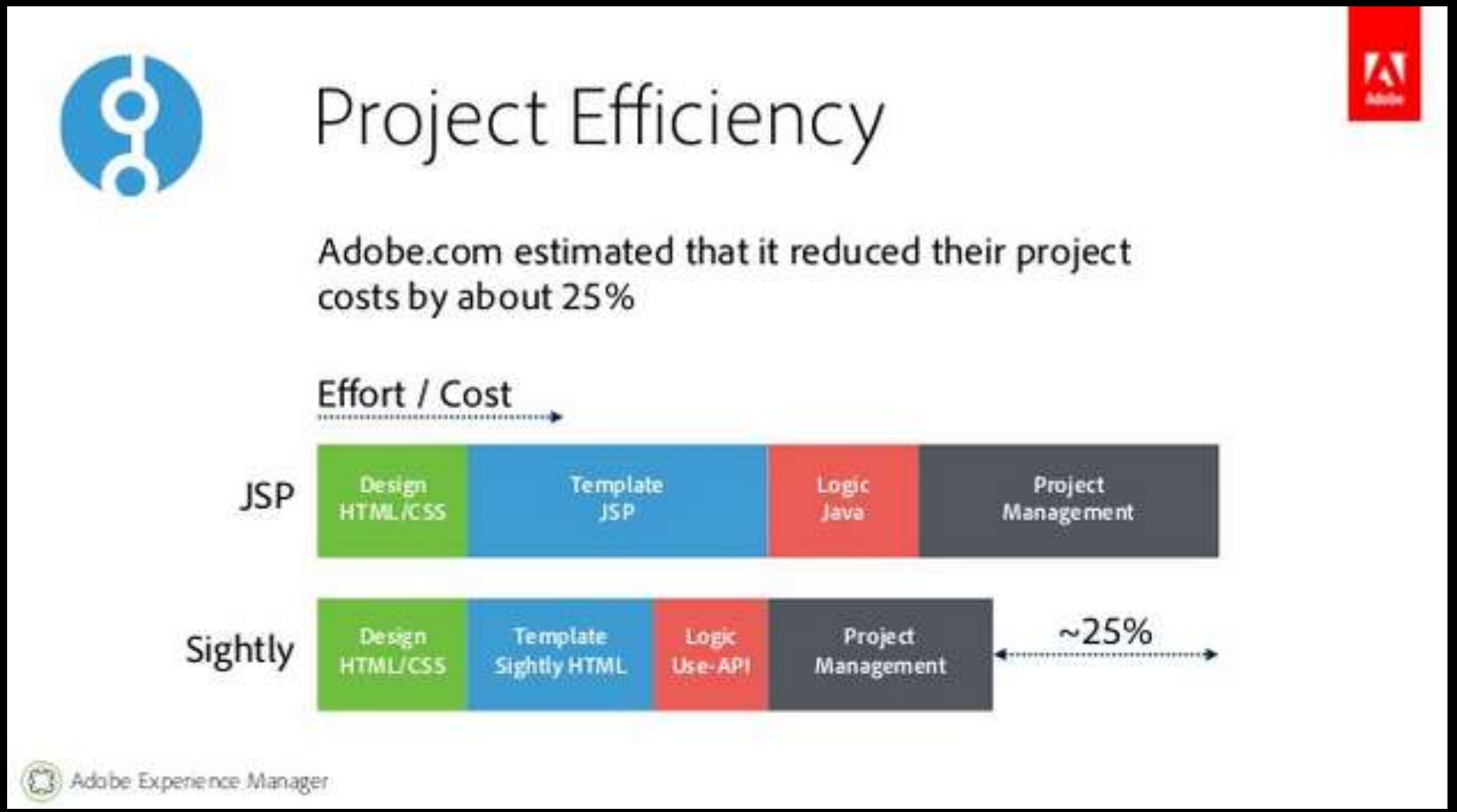
The standard workflow:

- Front end developers create HTML mark up, designs with all necessary client side libraries
- (AEM) back end developers take this well formed, high-fidelity ***static*** HTML "prototype"
  - Systematically split the whole template in pieces
  - Put it together again as JSP templates and components
  - Add custom business logic inside JSP templates and components

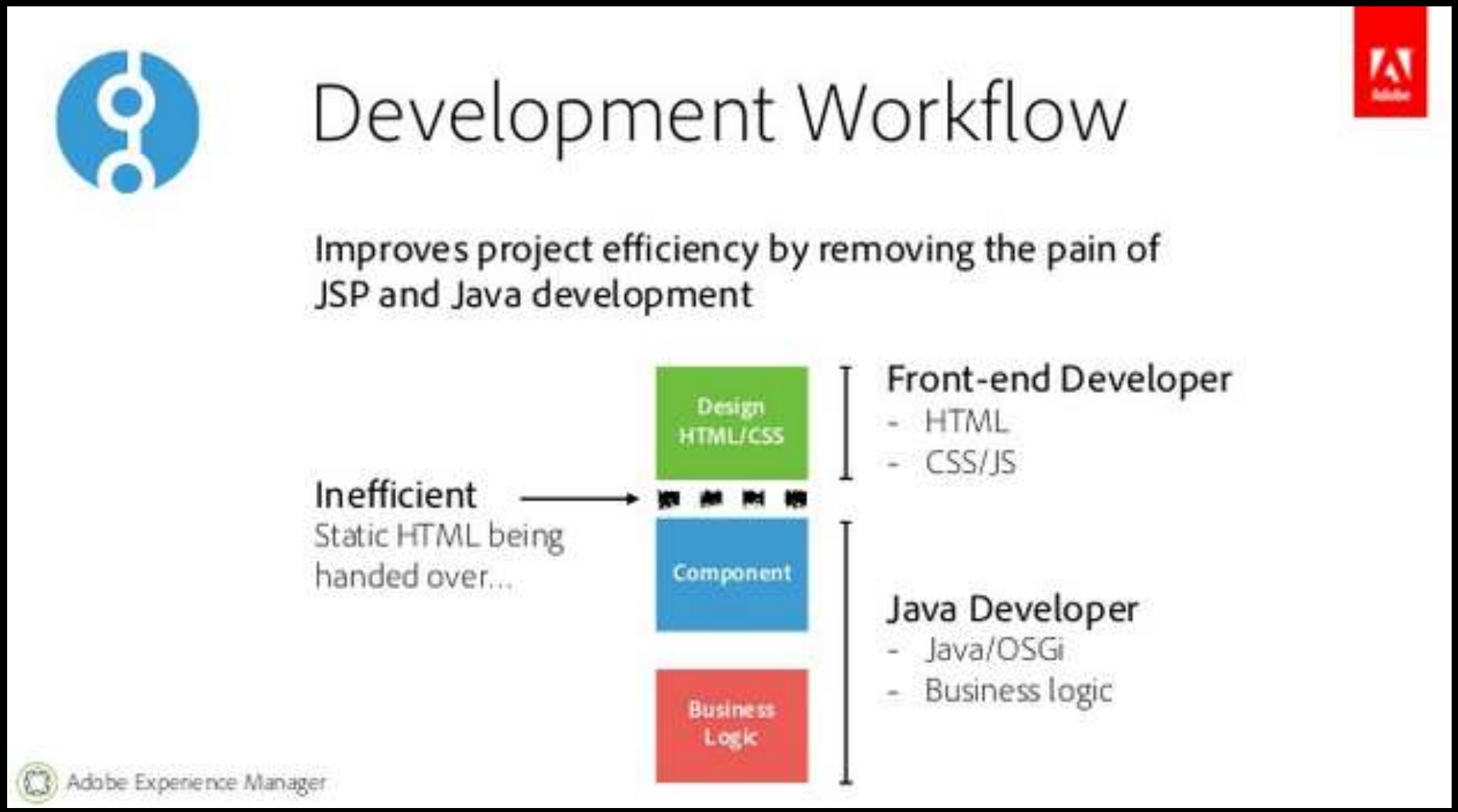
This leads to well known issues:

- The process of splitting and putting together is error prone
- The process itself is time consuming
- This way of development does not provide good (and simple) separation of concerns between UI and business logic
- As front end developers can't easily maintain JSPs, they don't develop new components "inside" AEM
  - They don't directly cope with components issues in Author mode (both visualization and UI issues)

Why Sightly: separation of concerns



Why Sightly: separation of concerns



Why Sightly: comparing JSP and Sightly

Sightly vs JSP

Sightly

```
<a href="${properties.link || '#'}" title="${properties.jcr:title}">
   ${properties.jcr:description}
</a>
```

JSP – Scriptlets

```
<%@include file="/libs/foundation/global.jsp"%>
<a href="<%= xssAPI.getValidHref(properties.get("link", "#")) %>"
   String title = properties.get("jcr:title", "");
   if (title.length() > 0) {
      %>title="<%= xssAPI.encodeForHTMLAttr(title) %>"<
   } %>>
   <%= xssAPI.encodeForHTML(properties.get("jcr:des ion", "")) %>
</a>
```

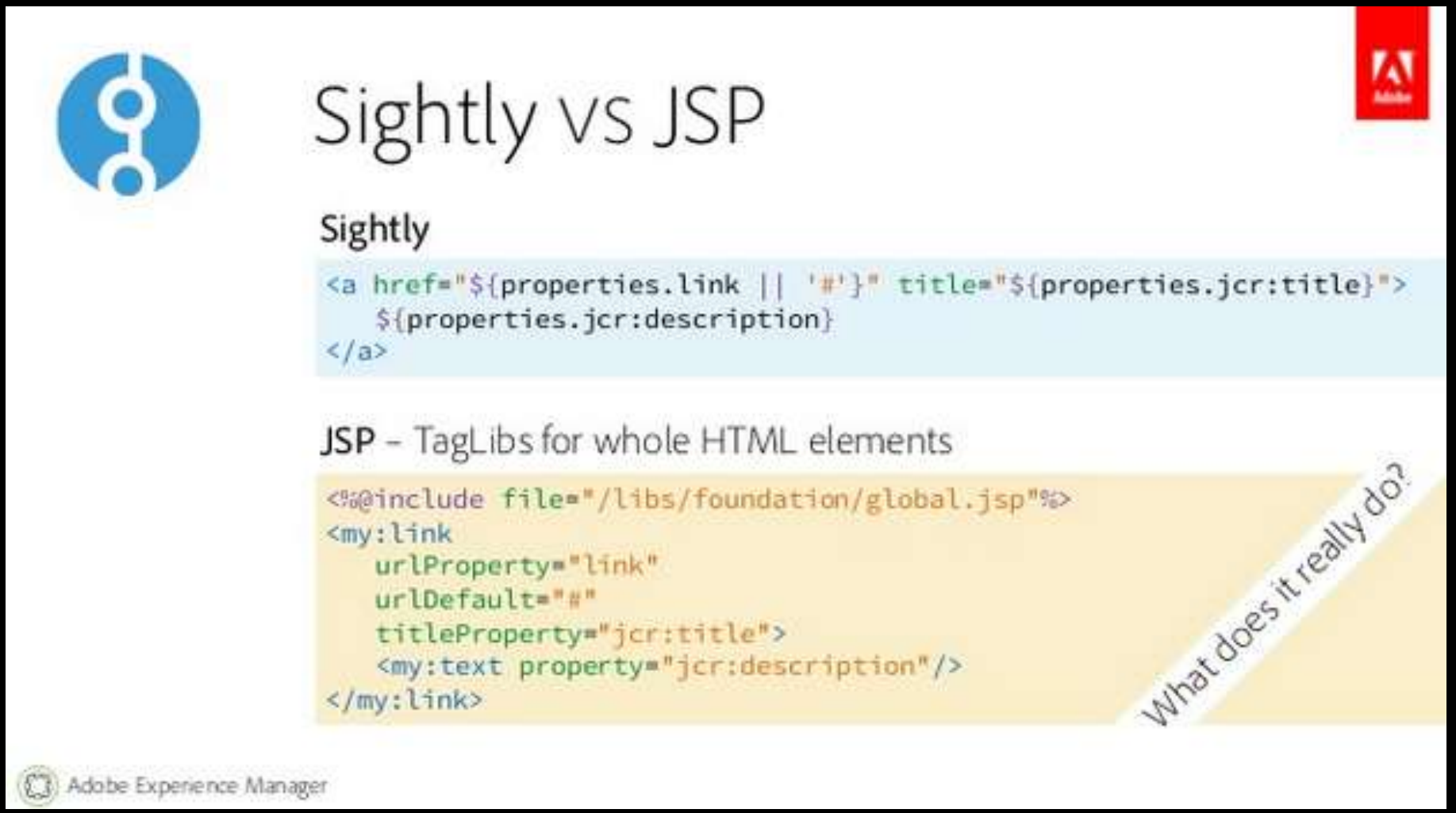
Please try again...

Adobe Experience Manager

Why Sightly: comparing JSP and Sightly

Why Sightly: comparing JSP and Sightly

Why Sightly: comparing JSP and Sightly



Why Sightly: comparing JSP and Sightly

- Writing custom tag lib is not easy
  - Need to mantain both Java class and TLD
- Custom tag lib lifecycle is hard to understand and difficult to integrate inside the templates
- Have you ever effectively used it for your projects?

	JSP	Sightly
Based on Published Standards / Open Source? Y (*)	N	Y
IDE Support?	Y	Y/N
Officially Documented / Supported?	Y	Y
Documented Extension Model?	Y	N
Includes XSS escaping?	Y (**)	Y
Allows Basic Logic?	Y	Y
Enables Bad Coding Practices?	Y	N

* Some proprietary TagLibs used for interacting with CQ

** Provided by additional tag libraries

Why Sightly: features

- Code-less language, forcing strict separation of concerns
- Powerful extension points with the Use-API
- Automatic contextual HTML escaping and XSS protection
- Automatically removes HTML attributes if value is empty
- Reuses HTML blocks for statements

Sightly: the basics of the language

Expression language example:

```
<a href=''{properties.link || '#'}'' title=''{properties.jcr:title}''>
        ${properties.jcr:description}
</a>
```

Features:

- Automatic contextual HTML escaping and XSS escaping (warning!)
- Fallback value if property is empty
- Remove HTML attribute if value is empty

Sightly: the basics of the language

- Sightly comments

```
<!-- /* This will disappear from the output *-->
```

- Expression language

```
${properties.myProperty}
```

  - They can only be used in attribute values, comments or in element content
  - Standard bindings are available as in JSPs

- Block statements

```
<div data-sly-include=''another-template.html''></div>
```

Sightly: the basics of the language

- Expression bindings:
- ${properties}
- ${pageProperties}
- ${inheritedPageProperties}

Access properties with dot notation ${properties.foo}

• ${request}	Sling Request
• ${resource}	Sling Resource
• ${currentPage}	WCM Page
• ${currentDesign}	WCM Design
• ${component}	WCM Component
• ${wcmmode}	WCM Mode

To avoid complex expressions inside templates, Sightly does not allow passing arguments to function call. Only zero argument calls are allowed from templates.

Sightly: the basics of the language

- Options allow to manipulate the result of an expression, or to pass arguments to block statements
  - Everything after the @ are comma separated options:

```
${myVar @ optionOne, optionTwo}
```

- Examples:
  - String formatting:

```
${'Page {0} of {1}' @ format=[current, total]}
```

  - Internationalization:

```
${'Page' @ i18n}
```

  - Array join:

```
${['one', 'two'] @ join='; '}
```

Sightly: the basics of the language

- Display context
  - Every expression in Sightly has a display context
  - Display context depends on the location within the HTML structure
  - Example: text node, attribute, …
  - Sightly automatically detect the context of expressions and escape them appropriately (to prevent XSS)
    This is not true for script (JS) and style (CSS) contexts
    In this case we should explicitly set the context
- Example

```
<a href="${properties.link}" title="${properties.title}">${properties.text}</a>
```

Three variables, three different contexts.
No explicit context setting is required in the above cases

Sightly: the basics of the language

- Display context option
  - The context option offers control over escaping and XSS protection
  - Explicit context must be set for style contexts:

```
<span style="color: ${properties.color @
context='styleToken'};">...</span>
```

  - To safely output markup (filtering out scripts)

```
<div>${properties.richText @ context='html'}</div>
```

    uses AntiSamy policy rules
    The default antisamy configuration is present at /libs/cq/xssprotection/config.xml, which can be overlaid with your custom config within /apps.
  - Adding URI validation protection to other attributes than src or href

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

Sightly: the basics of the language

- Most useful contexts and what they do:
  - uri — To display links and paths (validates URI)
  - attribute — Encodes HTML special characters
  - text — Encodes HTML special characters
  - scriptString — Encodes characters that would break out the string
  - styleString — Validates CSS tokens. Outputs nothing if it fails
  - html — Filters HTML to meet AntiSamy policy rules, removing what doesn’t match
  - unsafe — Disable all escaping and XSS protections

Sightly: the basics of the language

- Block statements
- To keep markup valid, block statements are defined by data-sly-* attributes that can be added to any element on the markup
- `<input data-sly-STATEMENT=''foo'' />`
  - Block statements can have no value, a static value, or an expression

```
<input data-sly-STATEMENT=''${bar}'' />
```

- Despite using data-attributes, block statements are all executed on the server and no data-sly-* attribute is sent to the client!
- Sightly block statements:
  - Markup inclusion: Include, Resource
  - Control flow: test, list, template, call
  - Markup modification: unwrap, element, attribute, text
  - Object initialization: use

Sightly: the basics of the language

- Template and call statements
  - Similar to data-sly-include
  - Main difference: you can pass parameters to the included template
  - Templates must be declared and called from another template
  - <data-sly-template> declares a template

```
<template data-sly-template.header> <div> my template </div> </template>
```

Defines a template called *header*

Notice: the host element are not output by Sightly. If you call this template the only printed mark up will be

```
<div> my template </div>
```

  - `<div data-sly-call=''header''></div>` calls the template *header* defined above
  - Templates can be located in a different file
  - Templates accept parameters

```
<template data-sly-template.two=''${ @ title}''> <h1>${title}</h1> </template>
<div data-sly-call=''${two @ title=properties.jcr:title}''></div>
```

Sightly: the basics of the language

- Unwrap statement
  - Removes the host element while retaining its content

```
<div data-sly-unwrap> <h1> Title </h1> </div>
```

Output: `<h1> Title </h1>`

- Warning!
  - Use unwrap only when there's no other way to write your template
  - Prefer adding statements to existing elements
  - Templates can easily become difficult to read
  - Unwrap can also be subject to condition

```
<div class=''edit-md'' data-sly-unwrap=''${wcmmode.edit}''>
        Text
</div>
```

Output: Text in EDIT mode

Output in PREVIEW/PUBLISH mode:

```
<div class=''edit-md''>
        Text
</div>
```

  - Use data-sly-test to remove the element content as well

Sightly: the basics of the language

- Use statement: Sightly Javascript Use API. Enables a Sightly file to access helper code written in Javascript.
  - Initialize a helper objects

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

Inside logic.js file

```
use(function()) {
        return {
                value: ''Hello World''
        };
});
```

Output:

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

- Use Javascript Use API only for very simple tasks (date formatting, text formatting, simple conditional logic, …):
  - Javascript Use API is server-side Javascript (some JS native features are not fully supported)
  - Cannot be debugged
  - Very hard to find errors
  - It's slow
  - Very hard to write Javascript Use API code for more complex task (e.g., cycling on repository nodes, calling external services, etc.)

Sightly: the basics of the language

- Use statement: Sightly Java Use API enables a Sightly file to access helper methods in a custom Java class.
- POJO extending WCMUse class
  - WCMUse has been deprecated from AEM 6.1 and replaced with WCMUsePojo which uses the new Sightly API from Apache Sling

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

/* Logic.java in component */
package apps.my_site.components.my_component;
import com.adobe.cq.sightly.WCMUse;

public class Logic extends WCMUse {
    private String hi;

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

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

- Local Java class: when the Java files are located in the content repository, next to the Sightly template, only the class name is required to call the logic
- Bundle Java class: the Java class must be compiled and deployed within an OSGi bundle (recommended when Java code implements logic common to many components)

Sightly: the basics of the language

- Many ways to do the same thing. But, what kind of Use-API is better?
- **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
Example: Java class retrieving information from a web service; Java class implementing logic for computing the menu structure

- **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).

Example: code implementing logic to output certain css classes inside the template mark up (e.g., list menu, with selected/non selected items)

Not a silver bullet

- Sightly is a good option to improve the maintanability of your AEM components but…
- You should follow best practices and guidelines otherwise your template code can explode easily
  - data-sly-unwrap can be evil
  - wrap author/preview version of HTML inside smaller templates to be included; this makes your components easier to read
  - avoid use context=‘‘unsafe’’
- It is not extensible with new block statements or options
- Can be hard to debug
  - <img data-sly-use.logic=‘‘foo.test.SomeLogic’’ data-sly-test=‘‘${logic.value}’’ >
  - if the test is false, everything below img tag won’t be output; this can be hard to debug
  - always use self closed elements
- Can lead to many Use API files containing the same logic
  - Before implementing a new one, think about a common class extending WCMUse or Sling Model
- Can be frustrating, sometimes
  - Write Use API external code even for the easy tasks
- Follow a style guide: https://github.com/Netcentric/aem-sightly-style-guide

Sightly and Sling Models: an amazing combination

- Sling Models are POJOs implementing the adapter pattern. They can be automatically mapped from Sling objects
  - Resource
  - SlingHttpRequest
- Entirely annotation driven
- OOTB, support resource properties (implemented with ValueMap), SlingBindings, OSGi services, request attributes
- Current latest version: 1.2.0
- AEM6+SP2 comes with 1.0.0
- Versions > 1.0.0 contain ***very*** useful features:
  - 1.0.6: injector-specific annotations
  - 1.1.0: @Self, @SlingObject, @ResourcePath annotations
  - 1.2.0: Sling validation

Sightly and Sling Models: basic usage

```
package com.foo.core;

@Model(adaptables=Resource.class)
public class MyModel {
        @Inject private String propertyName;
        public String getResult() {
                return ‘‘Hello World ’’ + propertyName;
        }
}
```

- Class is annotated with **@Model**
- **adaptbles** option defines which types of objects are adaptable to this Sling Model
- Fields that need to be injected are annotated

  In this case, propertyName is a property coming from the adapted resource

- Constructor injection (since 1.1.0)

```
@Model(adaptables=Resource.class)
public class MyModel {
    @Inject
    public MyModel(@Named("propertyName") String propertyName) {
      // constructor code
    }
}
```

Sightly and Sling Models: basic usage

@Injected fields/methods are assumed to be required. To mark them as optional, use @Optional:

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject @Optional
    private String otherName;
}
```

A default value can be provided (for Strings & primitives):

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject @Default(values="defaultValue")
    private String name;
}
```

Sightly and Sling Models: basic usage

OSGi services can be injected:

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject
    private ResourceResolverFactory resourceResolverFactory;
}
```

List injection for child resources works by injecting grand child resources (since Sling Models Impl 1.0.6). For example, the class

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject
    private List<Resource> addresses;
}
```

```
+- resource (being adapted)
 |
 +- addresses
    |
    +- address1
    |
    +- address2
```

addresses will contain address1 and address2

Sightly and Sling Models: basic usage

The @PostConstruct annotation can be used to add methods which are invoked upon completion of all injections:

```
@Model(adaptables=SlingHttpServletRequest.class)
public class MyModel {

    @Inject
    private PrintWriter out;

    @Inject
    @Named("log")
    private Logger logger;

    @PostConstruct
    protected void sayHello() {
        logger.info("hello");
    }
}
```

Sightly and Sling Models: basic usage

Available injectors:

https://sling.apache.org/documentation/bundles/models.html#available-injectors

Sightly and Sling Models: basic usage

Injector-specific annotation vs normal annotations

Those annotations replace @Via, @Filter, @Named, @Optional, @Required, @Source and @Inject. @Default may still be used in addition to the injector-specific annotation to set default values. All elements given above are optional.

Annotation	Supported Optional Elements	Injector
@ScriptVariable	optional and name	script-bindings
@ValueMapValue	optional, name and via	valuemap
@ChildResource	optional, name and via	child-resources
@RequestAttribute	optional, name and via	request-attributes
@ResourcePath	optional, path, and name	resource-path
@OSGiService	optional, filter	osgi-services
@Self	optional	self
@SlingObject	optional	sling-object

Sightly and Sling Models: client code

From a Sightly template:
<div data-sly-use.model=‘‘com.foo.core.MyModel’’>${model.result}

From Java code:
MyModel model = resource.adaptTo(MyModel.class)

Sightly and Sling Models: learning from experience

- Sling Models are instantiated everytime they are used with data-sly-use or adaptTo.

  Issues can happen when the Sling Model instantiation includes connection to DB or web service calls. Best practice: put Sling Model instance inside request object (in case the same model is used within the same request).
- They can be hard to debug sometimes…

  Since Sling Models 1.2.0 there is another way of instantiating models. The OSGi service ModelFactory provides a method for instantiating a model that throws exceptions. This is not allowed by the Javadoc contract of the adaptTo method. That way null checks are not necessary and it is easier to see why instantiation of the model failed.
- Why Sling Models and NOT WCMUse

  Sling Models are based on the adapter pattern

  Dependency injection

  Less simpler code
  It's a POJPO → unit testable

Sightly and Sling Models: ACS AEM commons and wcm.io

https://adobe-consulting-services.github.io/acs-aem-commons/features/aem-sling-models-injectors.html

Allows for Sling Models classes and interfaces to be injected with common AEM-related objects, namely those made available using <cq:defineObjects/>: resource, resourceResolver, componentContext, pageManager, etc.

```
@Inject
private Page resourcePage;
```

http://wcm.io/sling/models/

Support injection request-derived context objects on all models, not only when the adaptable is a request

Useful links

- https://github.com/Adobe-Marketing-Cloud/aem-sightly-sample-todomvc/

  Sightly TodoMVC Example

- https://github.com/Adobe-Marketing-Cloud/aem-sightly-repl

  AEM Sightly Read–Eval–Print Loop

A few hands-on examples

- AEM 6 Project Archetype version 10

https://github.com/Adobe-Marketing-Cloud/aem-project-archetype

- AEM 6 Project bootstrap inside Eclipse + AEM Developer Tools for Eclipse

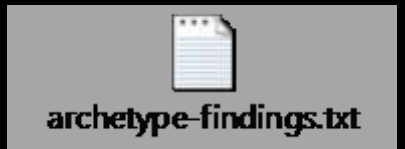
http://docs.adobe.com/content/docs/en/dev-tools/aem-eclipse.html

1. Project overview
2. Automatic synch: FS → repo (both content package and bundle)
3. On demand synch: repo → FS
4. Change properties from Eclipse and automatic synch FS → repo

- Brackets

1. Syntax highlighting, auto completion (data-sly-* and variables used inside expression within data attributes)
2. Automatic synch: FS → repo
3. On demand synch

- Archetype findings…



Thanks

Stefano Celentano
s.celentano@reply.it