sightly

BEAUTIFUL MARKUP

AEM6+ Component Development

@GabrielWalt, Product Manager

Adobe Experience Manager

sightly

BEAUTIFUL MARKUP

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

Follow @sightlyio on Twitter.
http://docs.adobe.com/docs/en/aem/6-0/develop/sightly.html

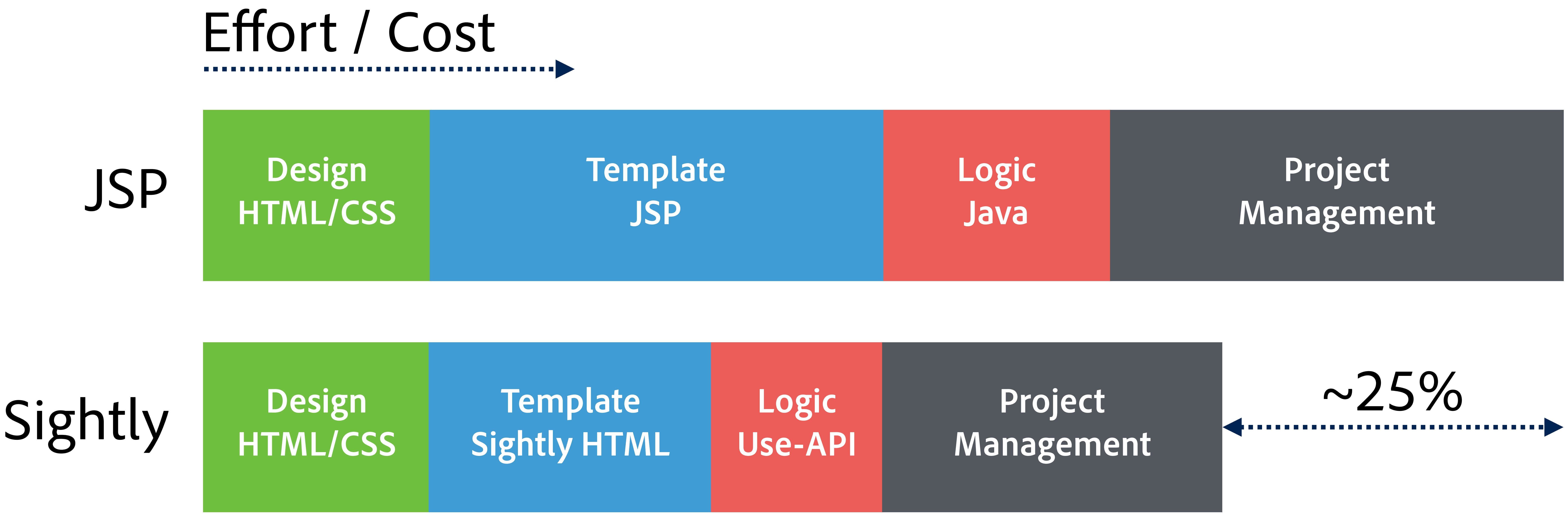
§ 1 Why Sightly

§ 2 Features

§ 3 Tooling

Project Efficiency

Adobe.com estimated that it reduced their project costs by about 25%



Effort / Cost

JSP

Design HTML/CSS

Template JSP

Logic Java

Project Management

Sightly

Design HTML/CSS

Template Sightly HTML

Logic Use-API

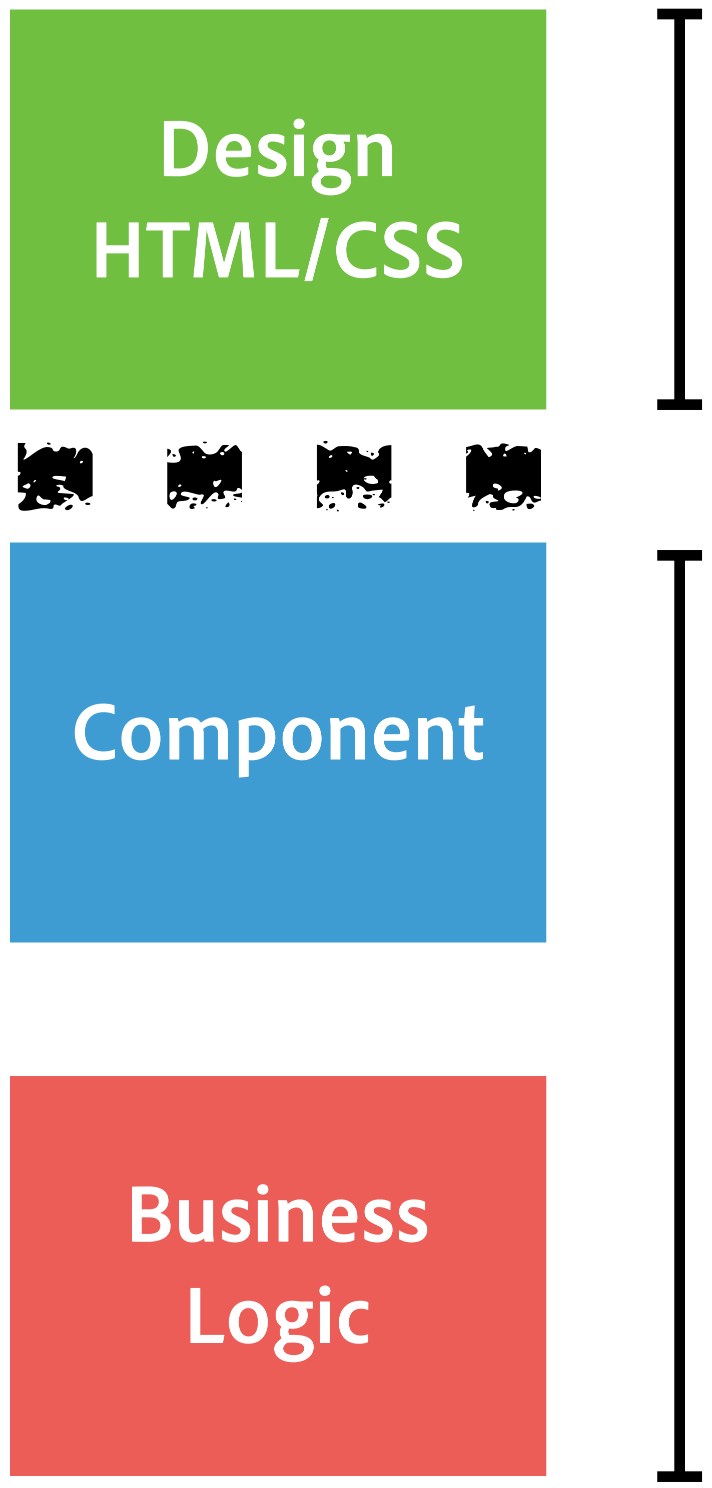
Project Management

~25%

Development Workflow

Improves project efficiency by removing the pain of JSP and Java development

Inefficient
Static HTML being
handed over…

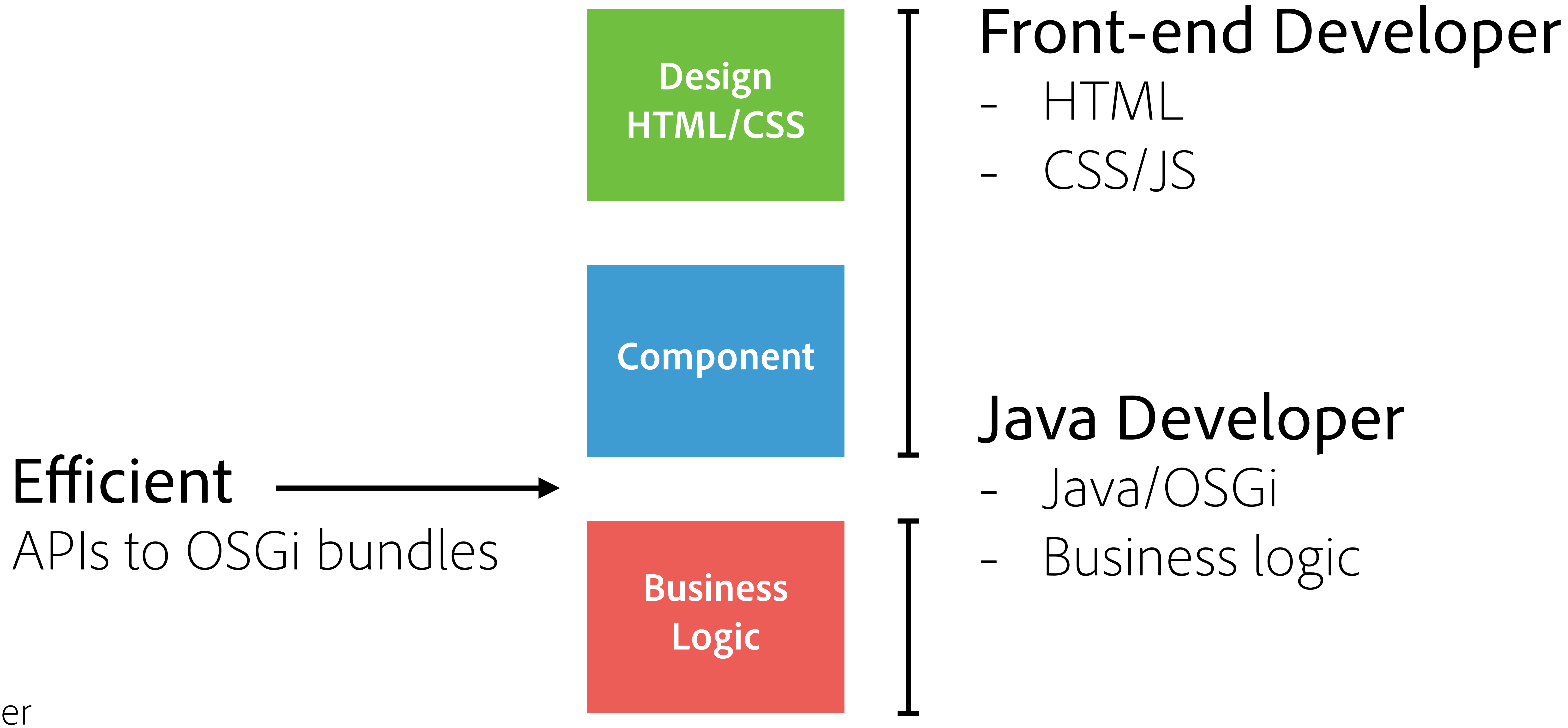


Front-end Developer
- HTML
- CSS/JS

Java Developer
- Java/OSGi
- Business logic

Development Workflow

Improves project efficiency by removing the pain of JSP and Java development



Design HTML/CSS

Component

Business Logic

Front-end Developer

- HTML
- CSS/JS

Java Developer

- Java/OSGi
- Business logic

Efficient

APIs to OSGi bundles

Development Workflow

Can be edited by front-end developers:

√ **Client Libraries** (CSS & JS)
× **JSP** (markup & logic)

Component

Development Workflow

Can be edited by front-end developers:

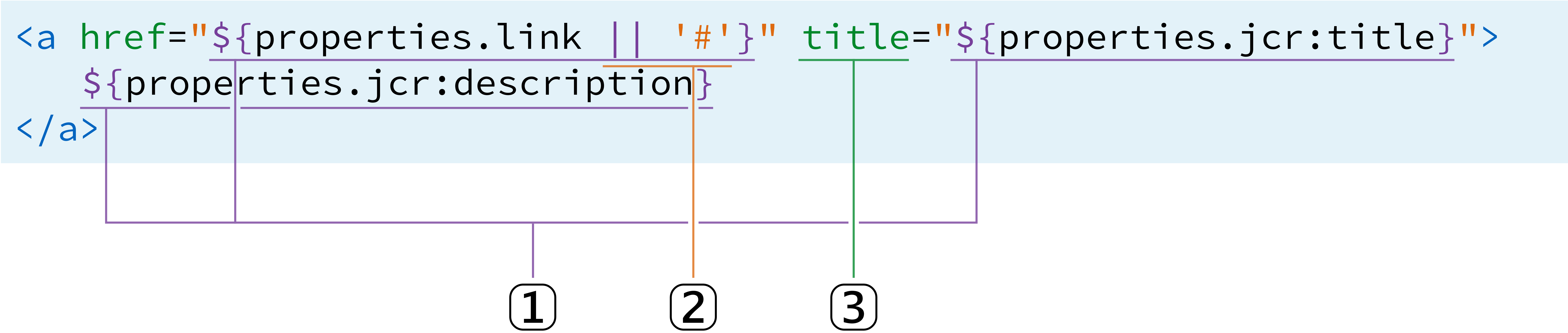
- √ **Client Libraries** (CSS & JS)
- × ~~JSP (markup & logic)~~
- √ **HTML markup** (Sightly template)
- √ **View logic** (server-side JS or Java)

Component

Sightly basic example

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

Sightly basic example



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

1 Automatic contextual HTML escaping and XSS protection of all variables

2 Fallback value if property is empty

3 Remove HTML attribute if value is empty

Sightly basic example

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

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

Sightly vs JSP

Sightly

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

JSP – Expression Language & JSTL

```
<%@include file="/libs/foundation/global.jsp"%>
<a href="${!empty properties.link ? xss:href(properties     )     }"
   <c:if test="${!empty properties['jcr:title']}">
      title="${xss:attr(properties['jcr:title'])}"
   </c:if>
>
   ${xss:text(properties['jcr:description'])}
</a>
```

No automatic security

Still complex

Adobe Experience Manager

Sightly vs JSP

Sightly

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

JSP – Custom Tag Libraries

```
<%@include file="/libs/foundation/global.jsp"%>
<a href="<out:href property='link' default='#'/>"
   <c:if test="${!empty properties['jcr:title']}">
      title="<out:attr property='jcr:title'/>"
   </c:if>
>
   <out:text property='jcr:description'/>
</a>
```

No automatic security

Many tags within tags

Sightly vs JSP

Sightly

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

JSP – TagLibs for whole HTML elements

```
<%@include file="/libs/foundation/global.jsp"%>
<my:link
   urlProperty="link"
   urlDefault="#"
   titleProperty="jcr:title">
   <my:text property="jcr:description"/>
</my:link>
```

What does it really do?

Sightly FTW!

Sightly

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

Secure

Readable

Explicit

§ 1 Why Sightly

§ 2 Features

§ 3 Tooling

Building Blocks

Comments

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

Expression Language

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

Block Statements

```
<div data-sly-include="other-template.html"></div>
```

Expressions

- Literals
- Variables
- Bindings
- Operators
- Options
- Contexts

Expressions

To avoid malformed HTML, expressions can only be used in attribute values, in element content, or in comments.

```
<!-- ${component.path} -->
<a href="${properties.link}">
   ${properties.jcr:description}
</a>
```

For setting element or attribute names, see the element and attribute block statements.

Standard bindings are available (same variables as in JSP); see the list of available variables.

Expression Fundamentals

Literals (positive integers, booleans, strings, arrays)

```
${42}
${true}              ${false}
${'Hello World'}   ${"Hello World"}
${[1, 2, 3]}         ${[42, true, 'Hello World']}
```

Variables (and accessing object properties)

```
${myVar}
${properties.propName}
${properties.jcr:title}
${properties['my property']}
${properties[myVar]}
```

Expression Bindings

Most useful available variables

${properties}
${pageProperties}
${inheritedPageProperties}

Properties of the resource, page or inherited from the page structure. Access properties with the dot notation: `${properties.foo}`

${request}	The Sling Request API
${resource}	The Sling Resource API
${currentPage}	The WCM Page API
${currentDesign}	The WCM Design API
${component}	The WCM Component API
${wcmmode}	The WCM Mode – use it like that: `${wcmmode.edit}`

To avoid complex expressions in templates, Sightly doesn't support passing arguments. So only zero argument methods can be called from the template.

Expression Operators

Logical operations (not, and, or)

```
${!myVar}
${conditionOne || conditionTwo}
${conditionOne && conditionTwo}
```

Equality / Inequality (only for same types)

```
${varOne == varTwo}     ${varOne != varTwo}
```

Comparison (only for integers)

```
${varOne < varTwo}      ${varOne > varTwo}
${varOne <= varTwo}     ${varOne >= varTwo}
```

Expression Operators

Conditional

```
${myChoice ? varOne : varTwo}
```

Grouping

```
${varOne && (varTwo || varThree)}
```

Expression Options

Options allow to manipulate the result of an expression, or to pass parameters to a block statement.

Everything after the @ are comma separated options

```
${myVar @ optOne, optTwo}
```

Options can have a value (literals or variables)

```
${myVar @ optOne='value', optTwo=[1, 2, 3]}
```

Parametric expression (containing only options)

```
${@ optOne='value', optTwo=[1, 2, 3]}
```

Expression Options

String formatting

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

Internationalization

```
${'Page' @ i18n}
${'Page' @ i18n, hint='Translation Hint'}
${'Page' @ i18n, locale='en-US'}
```

Array Join

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

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.

Block Statements

- **Markup Inclusion:** Include, Resource
- **Control Flow:** Test, List, Template, Call
- **Markup Modification:** Unwrap, Element, Attribute, Text
- **Object Initialization:** Use

Block Statements

To keep the markup valid, block statements are defined by `data-sly-*` attributes that can be added to any HTML element of the markup.

```
<input data-sly-STATEMENT="foo"/>

<div data-sly-STATEMENT.identifier="${bar}">
   <p>Example</p>
</div>
```

Block statements can have no value, a static value, or an expression.

What follows the dot declares an identifier to control how the result of the statement gets exposed.

Despite using data attributes, this is all executed on the server, and no `data-sly-*` attribute is sent to the client.

Include Statement

Includes the rendering of the indicated template (Sightly, JSP, ESP, etc.)

```
<section data-sly-include="other-template.html"></section>
```

Output:

```
<section><!-- Result of the rendered script --></section>
```

Resource Statement

Includes the result of the indicated resource.

```
<article data-sly-resource="path/to/resource"></article>
```

Output:

```
<article><!-- Result of the rendered resource --></article>
```

Resource Statement Options

Manipulating selectors (selectors, addSelectors, removeSelectors)

```
<article data-sly-resource="${'path/to/resource' @
selectors='mobile'}"></article>
```

Overriding the resource type

```
<article data-sly-resource="${'path/to/resource' @
resourceType='my/resource/type'}"></article>
```

Changing WCM mode

```
<article data-sly-resource="${'path/to/resource' @
wcmmode='disabled'}"></article>
```

Test Statement

Conditionally displays or removes the element and it's content.

```
<p data-sly-test="${properties.showText}">text</p>
```

Output:

```
<p>text</p>
```

... or nothing

Test Statement

The result of a test statement can be assigned to an identifier and reused; e.g. for something similar to an else statement.

```
<p data-sly-test.show="${properties.showText}">text</p>
<p data-sly-test="${!show}">No text</p>
```

The identifier declares a variable that holds the result of the test statement.

Output:

```
<p>text</p>
```

… or nothing

List Statement

Repeats the content for each enumerable item.

```
<ol data-sly-list="${currentPage.listChildren}">
    <li>${itemList.count}: ${item.title}</li>
</ol>
```

In that example, the `item` object is a Page object.

`itemList` has following members:

`index:`	zero-based counter.
`count:`	one-based counter.
`first:`	**`true`** for the first item.
`middle:`	**`true`** when not first or last.
`last:`	**`true`** for the last item.
`odd:`	**`true`** if index is odd.
`even:`	**`true`** if index is even.

Output:

```
<ol>
    <li>1: Triangle Page</li>
    <li>2: Square Page</li>
</ol>
```

List Statement

With the block statement dot notation, the `item*` variables can also be named explicitly, which can be useful for nested lists.

```
<ol data-sly-list.child="${currentPage.listChildren}">
    <li>${childList.count}: ${child.title}</li>
</ol>
```

Output:

```
<ol>
    <li>1: Triangle Page</li>
    <li>2: Square Page</li>
</ol>
```

Template & Call Statements

Declare and call a markup snippet with named parameters.

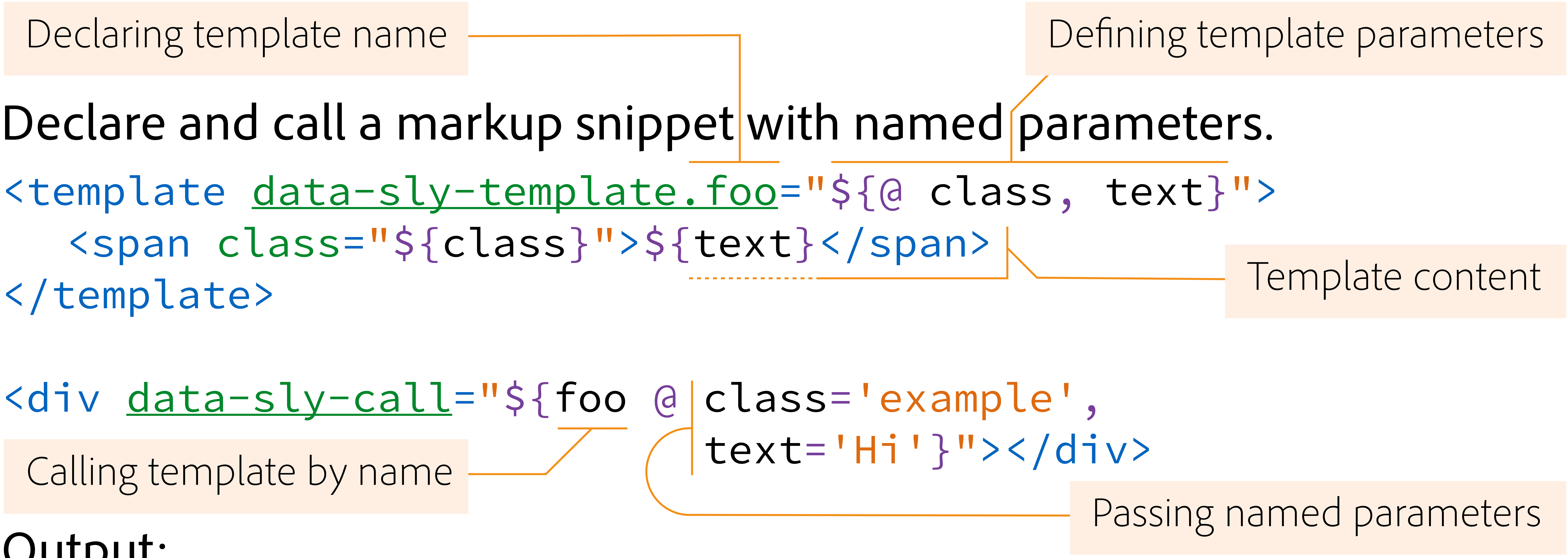
```
<template data-sly-template.foo="${@ class, text}">
   <span class="${class}">${text}</span>
</template>

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

Output:

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

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>
```

Template content

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

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>
```

Unwrap Statement

Removes the host element while retaining its content.

```
<div data-sly-test="${properties.showText}"
     data-sly-unwrap>text</div>
```

Output:

`text` … or nothing

Use unwrap only when there's no other way to write your template: prefer as much as possible to add statements on existing elements than to add elements for the sightly statements and removing them again with unwrap. This will help making the template look as close as possible to the generated output.

Unwrap Statement

Unwrap can also conditionally remove the outer element.

```
<div class="editable" data-sly-unwrap="${!wcmmode.edit}">
    text
</div>
```

Output in edit mode:

```
<div class="editable">
    text
</div>
```

Output on publish:

```
text
```

Element Statement

Changes the name of the current element.

```
<h1 data-sly-element="${titleElement}">Title</h1>
```

Output:

```
<h3>Title</h3>
```

There's a whitelisted number of allowed elements that can be used. This security can be disabled with the `@ context='unsafe'` option.

Use element with care as it allows to define parts of the markup from the logic, which can lessen the separation of concerns.

Attribute Statement

Sets multiple attributes to the current element.

```
<input data-sly-attribute="${keyValueMapOfAttributes}"/>
```

Output:

```
<input type="text" name="firstName" value="Alison"/>
```

Use attribute with care as it allows to define parts of the markup from the logic, which can lessen the separation of concerns.

Attribute Statement

The attribute statement can be used with an identifier to indicate one single attribute to set/overwrite.

```
<a href="#" data-sly-attribute.href="${link}">link</a>
```

Would have been equivalent to:

```
<a href="${link}">link</a>
```

Output:

```
<a href="link.html">link</a>
```

But the first example is valid HTML5, while second one fails the W3C HTML5 validation, because the expression is an invalid URL path.

Text Statement

Replaces the content of the element with the provided text.

```
<div data-sly-text="${content}">Lorem ipsum</div>
```

Would have been equivalent to:

```
<div>${content}</div>
```

Output:

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

The text statement can be interesting to “annotate” with behavior a static HTML that has been handed over by a designer, while keeping it visually unchanged when the file is opened in a browser.

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 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;
    }
}
```

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.

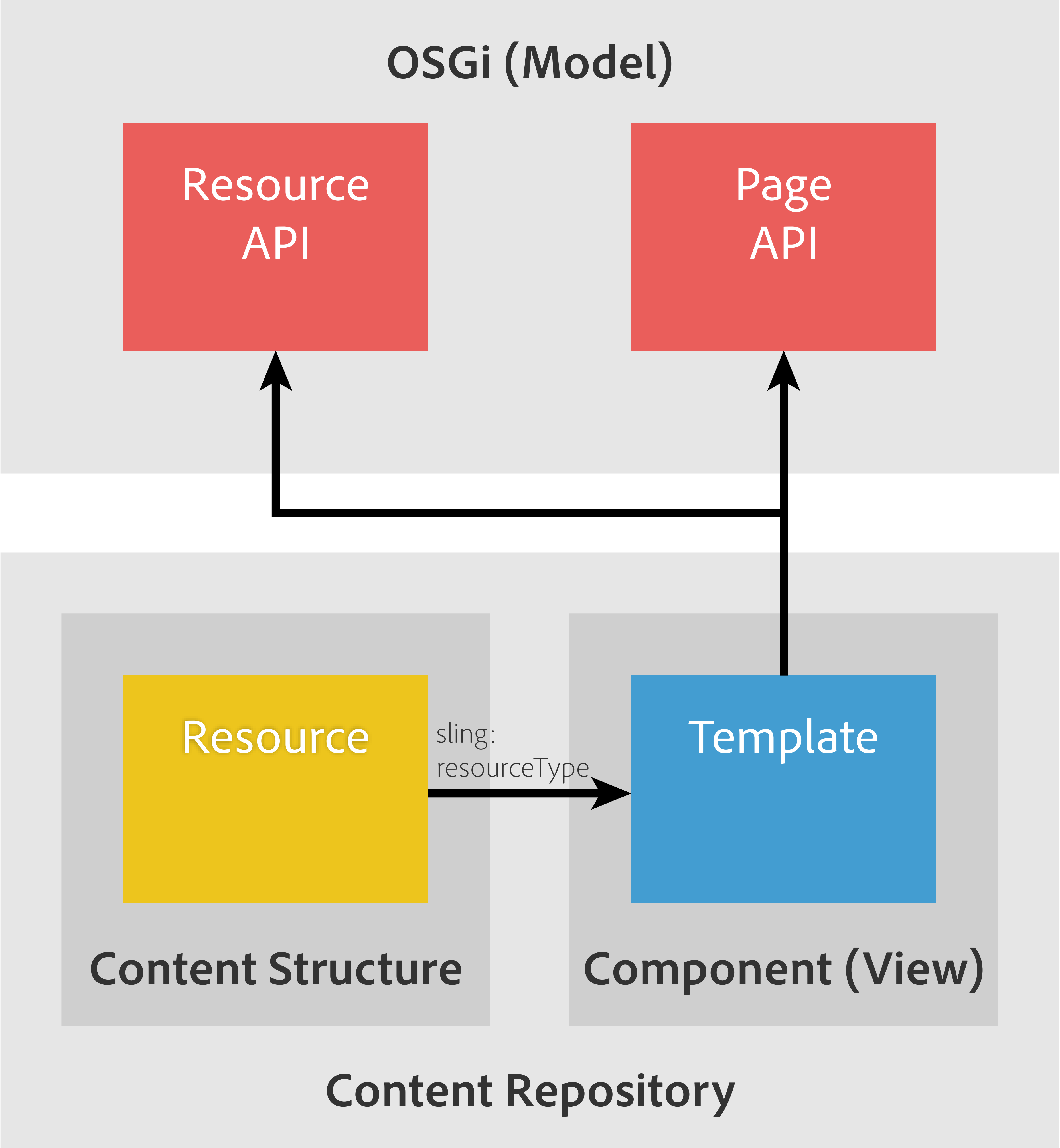
→ **JavaScript located in component**

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

→ **Java located in component**

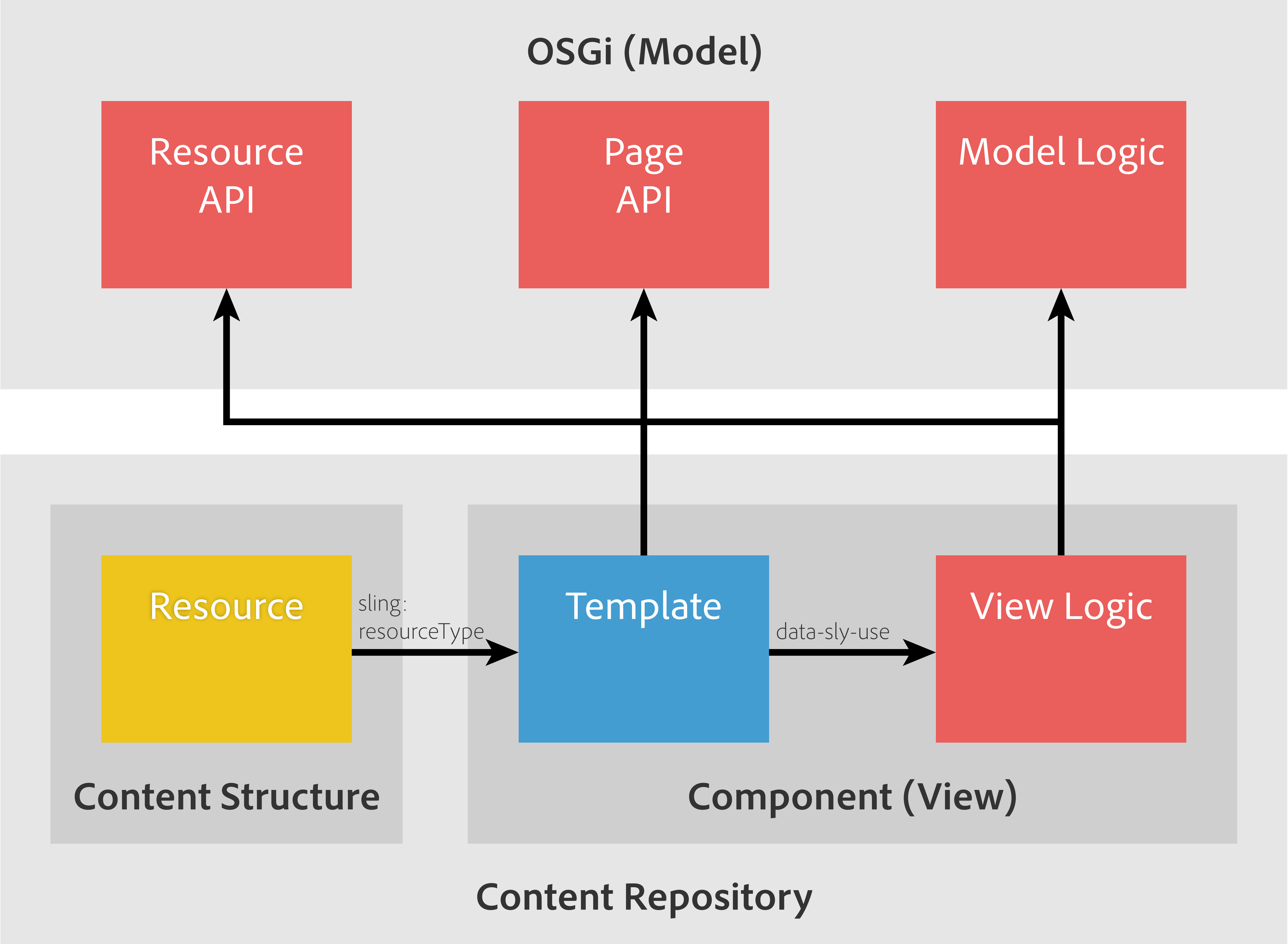
If performance is critical (e.g. when many requests are not cached by the dispatcher).

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>
```

Sightly Don’ts

- Don’t use the option `context="unsafe"`, unless there’s no other choice. When doing so, carefully assess the consequences.
- Don’t use `data-sly-unwrap` if there’s another HTML element on which you can put the Sightly instructions.
- Avoid to use `data-sly-element` and `data-sly-attribute` in a way that makes the logic define parts of the generated markup.

Sightly FAQ

What does Sightly enable that isn't possible with JSP?

- Systematic state-of-the-art protection against cross-site scripting injection.
- Possibility for front-end developers to easily participate on AEM projects.
- Strictly enforced separation of concern, yet with flexible binding for logic.
- Tailored to the Sling use-case.

Should JSPs be refactored to Sightly?

Refactoring can be expensive, which goes against the goal of Sightly to increase development efficiency. If the quality of existing JSPs is sufficient, it is rather advised to use Sightly for newly built/improved components.

Will JSP go away?

As of today, there are no plans for that.

§ 1 Why Sightly

§ 2 Features

§ 3 Tooling

IDE & Developer Mode

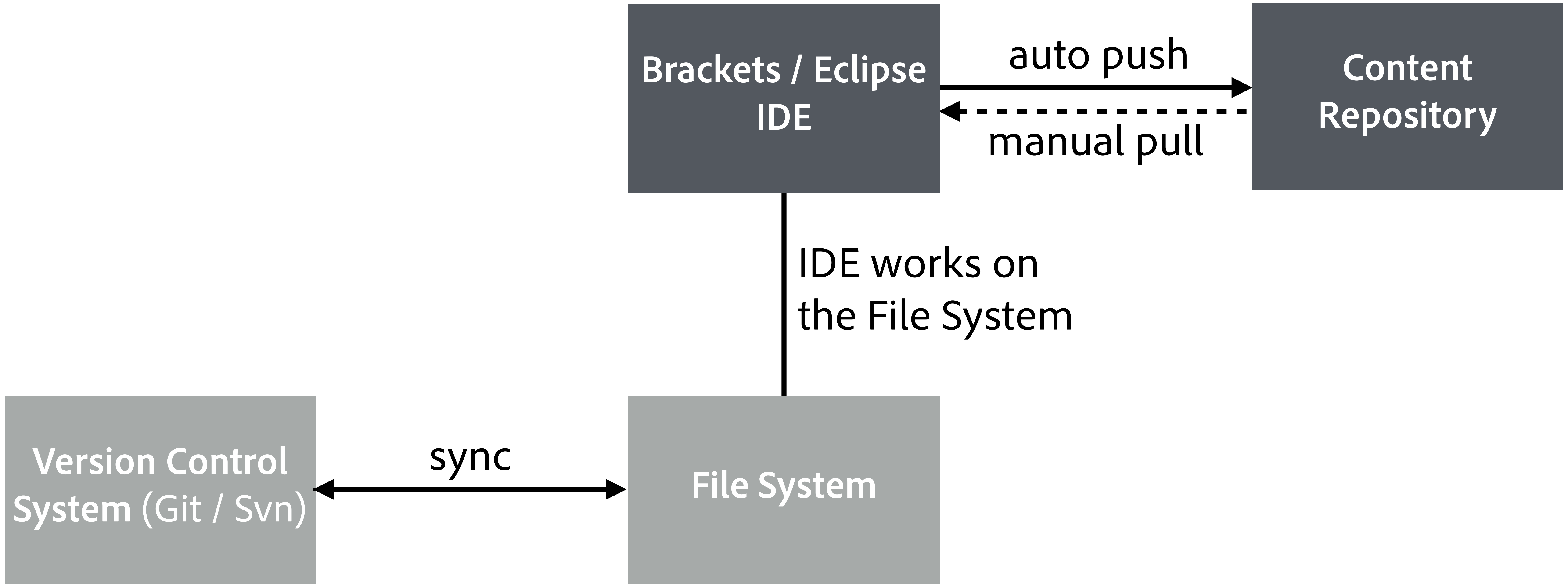
- Improve learning curve and efficiency of developers
- An IDE plugin for each developer role

Brackets plugin
for the Front-End developers
http://docs.adobe.com/docs/en/dev-tools/sightly-brackets.html

Eclipse plugin
for the Java developers
http://docs.adobe.com/docs/en/dev-tools/aem-eclipse.html

IDE Sync

Work on file system + transparent sync & content editing



Brackets / Eclipse IDE

auto push

manual pull

Content Repository

IDE works on the File System

Version Control System (Git / Svn)

sync

File System

Thank you!

Sightly

- Documentation
- Specification
- Sightly AEM Page Example *(requires instance on localhost:4502)*
- TodoMVC Example

Tools

- Live Sightly execution environment
- Sightly Brackets extension
- AEM Developer Tools for Eclipse
- AEM Developer Mode

Adobe