**SLING MODELS**

A Sling Model is implemented as an OSGi bundle. A Java class located in the OSGi bundle is annotated with **@Model** and the adaptable class (for example, **@Model(adaptables = Resource.class**). The data members (Fields) use **@Inject** annotations.

Many Sling projects want to be able to create model objects - POJOs which are automatically mapped from Sling objects, typically resources, but also request objects. Sometimes these POJOs need OSGi services as well.

**[Design Goals](https://sling.apache.org/documentation/bundles/models.html" \l "design-goals)**

* Entirely annotation driven. "Pure" POJOs.
* Use standard annotations where possible.
* Pluggable
* OOTB, support resource properties (via ValueMap), SlingBindings, OSGi services, request attributes
* Adapt multiple objects - minimal required Resource and SlingHttpServletRequest
* Client doesn't know/care that these objects are different than any other adapter factory
* Support both classes and interfaces.
* Work with existing Sling infrastructure (i.e. not require changes to other bundles).

**[Basic Usage](https://sling.apache.org/documentation/bundles/models.html" \l "basic-usage)**

In the simplest case, the class is annotated with @Model and the adaptable class. Fields which need to be injected are annotated with @Inject:

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject**

private String propertyName;

}

In this case, a property named "propertyName" will be looked up from the Resource (after first adapting it to a ValueMap) and it is injected.

For an interface, it is similar:

@**Model**(**adaptables**=**Resource**.**class**)

public interface MyModel {

@**Inject**

String getPropertyName();

}

Constructor injection is also supported (as of Sling Models 1.1.0):

@Model(adaptables=Resource.class)

**public** **class** **MyModel** {

@Inject

**public** **MyModel**(@Named("propertyName") String propertyName) {

// constructor code

}

}

Because the name of a constructor argument parameter cannot be detected via the Java Reflection API a @Named annotation is mandatory for injectors that require a name for resolving the injection.

In order for these classes to be picked up, there is a header which must be added to the bundle's manifest:

**<Sling-Model-Packages>**

**org**.apache.sling.models.it.models

**</Sling-Model-Packages>**

This header must contain all packages which contain model classes or interfaces. However, subpackages need not be listed individually, e.g. the header above will also pick up model classes in org.apache.sling.models.it.models.sub. Multiple packages can be listed in a comma-separated list (any whitespace will be removed):

**<Sling-Model-Packages>**

**org**.apache.sling.models.it.models,

**org**.apache.sling.other.models

**</Sling-Model-Packages>**

Alternatively it is possible to list all classes individually that are Sling Models classes via the Sling-Model-Classes header.

If you use the Sling Models bnd plugin all required bundle headers are generated automatically at build time (see chapter 'Registration of Sling Models classes via bnd plugin' below).

**[Client Code](https://sling.apache.org/documentation/bundles/models.html" \l "client-code)**

[adaptTo()](https://sling.apache.org/documentation/bundles/models.html" \l "adaptto-)

Client code doesn't need to be aware that Sling Models is being used. It just uses the Sling Adapter framework:

**MyModel** model = resource.adaptTo(MyModel.class)

Or

<sling:adaptTo adaptable="${resource}" adaptTo="org.apache.sling.models.it.models.MyModel" var="model"/>

Or

${sling:adaptTo(resource, 'org.apache.sling.models.it.models.MyModel')}

As with other AdapterFactories, if the adaptation can't be made for any reason, adaptTo() returns null.

[ModelFactory (since 1.2.0)](https://sling.apache.org/documentation/bundles/models.html" \l "modelfactory-since-1-2-0-)

*See also*[*SLING-3709*](https://issues.apache.org/jira/browse/SLING-3709)

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.

**try** {

MyModel model = modelFactory.createModel(object, MyModel.class);

} **catch** (**Exception** e) {

// give out error message that the model could not be instantiated.

// The exception contains further information.

// See the javadoc of the ModelFactory for which Exception can be expected here

}

In addition ModelFactory provides methods for checking whether a given class is a model at all (having the model annotation) or whether a class can be adapted from a given adaptable.

**[Other Options](https://sling.apache.org/documentation/bundles/models.html" \l "other-options)**

[Names](https://sling.apache.org/documentation/bundles/models.html" \l "names)

If the field or method name doesn't exactly match the property name, @Named can be used:

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject** @Named("secondPropertyName")

private String otherName;

}

[Optional and Required](https://sling.apache.org/documentation/bundles/models.html" \l "optional-and-required)

@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;

}

If a majority of @Injected fields/methods are optional, it is possible (since Sling Models API 1.0.2/Impl 1.0.6) to change the default injection strategy by using adding defaultInjectionStrategy = DefaultInjectionStrategy.OPTIONAL to the @Model annotation:

@**Model**(**adaptables**=**Resource**.**class**, defaultInjectionStrategy=DefaultInjectionStrategy.OPTIONAL)

public class MyModel {

@**Inject**

private String otherName;

}

To still mark some fields/methods as being mandatory while relying on defaultInjectionStrategy = DefaultInjectionStrategy.OPTIONAL for all other fields, the annotation @Required can be used.

@Optional annotations are only evaluated when using the defaultInjectionStrategy = DefaultInjectionStrategy.REQUIRED (which is the default), @Required annotations only if using defaultInjectionStrategy = DefaultInjectionStrategy.OPTIONAL.

[Defaults](https://sling.apache.org/documentation/bundles/models.html" \l "defaults)

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

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject** @Default(values="defaultValue")

private String name;

}

Defaults can also be arrays:

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject** @Default(intValues={1,2,3,4})

**private** **int**[] **integers**;

}

OSGi services can be injected:

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject**

private ResourceResolverFactory resourceResolverFactory;

}

In this case, the name is not used -- only the class name.

[Collections](https://sling.apache.org/documentation/bundles/models.html" \l "collections)

Lists and arrays are supported by some injectors. For the details look at the table given in [Available Injectors](https://sling.apache.org/documentation/bundles/models.html#available-injectors):

@**Model**(**adaptables**=**Resource**.**class**)

public class MyModel {

@**Inject**

private List<Servlet> servlets;

}

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;

}

Is suitable for a resource structure such as:

+- resource (being adapted)

|

+- addresses

|

+- address1

|

+- address2

In this case, the addresses List will contain address1 and address2.

[OSGi Service Filters](https://sling.apache.org/documentation/bundles/models.html" \l "osgi-service-filters)

OSGi injection can be filtered:

@Model(adaptables=SlingHttpServletRequest.class)

**public** **class** **MyModel** {

@Inject

**private** PrintWriter out;

@Inject

@Named("log")

**private** Logger logger;

@Inject

@Filter("(paths=/bin/something)")

**private** List<Servlet> servlets;

}

[PostConstruct Methods](https://sling.apache.org/documentation/bundles/models.html" \l "postconstruct-methods)

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");

}

}

@PostConstruct methods in a super class will be invoked first. If a @PostConstruct method exists in a subclass with the same name as in the parent class, only the subclass method will be invoked. This is the case regardless of the scope of either method.

Since Sling Models Implementation 1.4.6, @PostConstruct methods may return a false boolean value in which case the model creation will fail without logging any exception (a message will be logged at the DEBUG level).

[Via](https://sling.apache.org/documentation/bundles/models.html" \l "via)

In some cases, a different object should be used as the adaptable instead of the original adaptable. This can be done using the @Via annotation. By default, this can be done using a JavaBean property of the adaptable:

@Model(adaptables=SlingHttpServletRequest.class)

**public** **interface** **MyModel** {

// will return request.getResource().getValueMap().get("propertyName", String.class)

@Inject @Via("resource")

String **getPropertyName**();

}

A different strategy can be used to define the adaptable by specifying a type attribute:

@Model(adaptables=Resource.**class**)

public interface MyModel {

// will **return** resource.getChild("jcr:content").getValueMap().get("propertyName", String.**class**)

@Inject @Via(value = "jcr:content", type = ChildResource.**class**)

String getPropertyName();

}

See the [Via Types](https://sling.apache.org/documentation/bundles/models.html#via-types-since-api-134implementation-140) section below for details on the included types for @Via.

[Source](https://sling.apache.org/documentation/bundles/models.html" \l "source)

If there is ambiguity where a given injection could be handled by more than one injector, the @Source annotation can be used to define which injector is responsible:

@Model(adaptables=SlingHttpServletRequest.class)

**public** **interface** **MyModel** {

// Ensure that "resource" is retrived from the bindings, not a request attribute

@Inject @Source("script-bindings")

Resource **getResource**();

}

[Adaptations](https://sling.apache.org/documentation/bundles/models.html" \l "adaptations)

If the injected object does not match the desired type and the object implements the Adaptable interface, Sling Models will try to adapt it. This provides the ability to create rich object graphs. For example:

@**Model**(**adaptables**=**Resource**.**class**)

public interface MyModel {

@**Inject**

ImageModel getImage();

}

@**Model**(**adaptables**=**Resource**.**class**)

public interface ImageModel {

@**Inject**

String getPath();

}

When a resource is adapted to MyModel, a child resource named image is automatically adapted to an instance of ImageModel.

Constructor injection is supported for the adaptable itself. For example:

@Model(adaptables=Resource.class)

**public** **class** **MyModel** {

**public** **MyModel**(Resource resource) {

**this**.resource = resource;

}

**private** **final** Resource resource;

@Inject

**private** String propertyName;

}

[Sling Validation (since 1.2.0)](https://sling.apache.org/documentation/bundles/models.html" \l "sling-validation-since-1-2-0-)

*See also [SLING-4161](https://issues.apache.org/jira/browse/SLING-4161)*

You can use the attribute validation on the Model annotation to call a validation service on the resource being used by the Sling model. That attribute supports three different values:

| **Value** | **Description** | **Invalid validation model** | **No validation model found** | **Resource invalid according to model** |
| --- | --- | --- | --- | --- |
| DISABLED (default) | don't validate the resource bound to the Model | Model instantiated | Model instantiated | Model instantiated |
| REQUIRED | enforce validation of the resource bound to the Model | Model not instantiated | Model not instantiated | Model not instantiated |
| OPTIONAL | validate the resource bound to the Model (if a validation model is found) | Model not instantiated | Model instantiated | Model not instantiated |

In case the model is not instantiated an appropriate error message is logged (if adaptTo() is used) or an appropriate exception is thrown (if ModelFactory.createModel() is used).

The only implementation for this Sling Models validation service is leveraging [Sling Validation](https://sling.apache.org/documentation/bundles/validation.html) and is located in the bundle [org.apache.sling.models.validation-impl](https://github.com/apache/sling-org-apache-sling-models-validation-impl). Validation is only working on models which are adapted from either Resource or SlingHttpServletRequest and if the Sling Validation Bundle is deployed.

**[Custom Injectors](https://sling.apache.org/documentation/bundles/models.html" \l "custom-injectors)**

To create a custom injector, simply implement the org.apache.sling.models.spi.Injector interface and register your implementation with the OSGi service registry. Please refer to the [standard injectors in Git](https://github.com/apache/sling-org-apache-sling-models-impl/tree/master/src/main/java/org/apache/sling/models/impl/injectors) for examples.

Injectors are invoked in order of their service ranking, from lowest to highest. See the table below for the rankings of the standard injectors.

**[Annotation Reference](https://sling.apache.org/documentation/bundles/models.html" \l "annotation-reference)**

@Model : declares a model class or interface

@Inject : marks a field or method as injectable

@Named : declare a name for the injection (otherwise, defaults based on field or method name).

@Optional : marks a field or method injection as optional

@Source : explictly tie an injected field or method to a particular injector (by name). Can also be on other annotations.

@Filter : an OSGi service filter

@PostConstruct : methods to call upon model option creation (only for model classes)

@Via : change the adaptable as the source of the injection

@Default : set default values for a field or method

@Path : only used together with the resource-path injector to specify the path of a resource

@Exporters/@Exporter/@ExporterOptions/@ExporterOption : See Exporter Framework section below

In addition all [injector-specific annotations](https://sling.apache.org/documentation/bundles/models.html#injector-specific-annotations).

**[Available Injectors](https://sling.apache.org/documentation/bundles/models.html" \l "available-injectors)**

| **Title** | **Name (for**  **@Source)** | **Service Ranking** | **Available Since (Implementation Version)** | **Description** | **Applicable To (including using @Via)** | **Accepts Null Name?** | **Array Support** | **Parameterized Type Support** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Script Bindings | script-bindings | 1000 | 1.0.0 | Lookup objects in the script bindings object by name. | A ServletRequest object which has the Sling Bindings attribute defined | no | no conversion is done | If a parameterized type is passed, the bindings value must be of a compatible type of the parameterized type. |
| Value Map | valuemap | 2000 | 1.0.0 | Gets a property from a ValueMap by name. | Any object which is or can be adapted to a ValueMap | no | Primitive arrays wrapped/unwrapped as necessary. Wrapper object arrays are unwrapped/wrapped as necessary. | Parameterized List and Collectioninjection points are injected by getting an array of the component type and creating an unmodifiable List from the array. |
| Child Resources | child-resources | 3000 | 1.0.0 | Gets a child resource by name. | Resource objects | no | none | if a parameterized type List or Collection is passed, a List<Resource>is returned (the contents of which may be adapted to the target type) filled with all child resources of the resource looked up by the given name. |
| Request Attributes | request-attributes | 4000 | 1.0.0 | Get a request attribute by name. | ServletRequest objects | no | no conversion is done | If a parameterized type is passed, the request attribute must be of a compatible type of the parameterized type. |
| OSGi Services | osgi-services | 5000 | 1.0.0 | Lookup services based on class name. Since Sling Models Impl 1.2.8 ([SLING-5664](https://issues.apache.org/jira/browse/SLING-5664)) the service with the highest service ranking is returned. In case multiple services are returned, they are ordered descending by their service ranking (i.e. the one with the highest ranking first). | Any object | yes | yes | Parameterized List and Collectioninjection points are injected by getting an array of the services and creating an unmodifiable List from the array. |
| Resource Path | resource-path | 2500 | 1.1.0 | Injects one or multiple resources. The resource paths are either given by @Path annotations, the element path or paths of the annotation @ResourcePath or by paths given through a resource property being referenced by either @Named or element name of the annotation @ResourcePath. | Resource or SlingHttpServletRequestobjects | yes | yes | none |
| Self | self | Integer.MAX\_VALUE | 1.1.0 | Injects the adaptable object itself (if the class of the field matches or is a supertype). If the @Self annotation is present it is tried to adapt the adaptable to the field type. | Any object | yes | none | none |
| Sling Object | sling-object | Integer.MAX\_VALUE | 1.1.0 | Injects commonly used sling objects if the field matches with the class: request, response, resource resolver, current resource, SlingScriptHelper. This works only if the adaptable can get the according information, i.e. all objects are available via SlingHttpServletRequest while ResourceResolver can only resolve the ResourceResolver object and nothing else. A discussion around this limitation can be found at [SLING-4083](https://issues.apache.org/jira/browse/SLING-4083). Also Resources can only be injected if the according injector-specific annotation is used (@SlingObject). | Resource, ResourceResolver or SlingHttpServletRequestobjects (not all objects can be resolved by all adaptables). | yes | none | none |

**[Injector-specific Annotations](https://sling.apache.org/documentation/bundles/models.html" \l "injector-specific-annotations)**

*Introduced with*[*SLING-3499*](https://issues.apache.org/jira/browse/SLING-3499)*in Sling Models Impl 1.0.6*

Sometimes it is desirable to use customized annotations which aggregate the standard annotations described above. This will generally have the following advantages over using the standard annotations:

* Less code to write (only one annotation is necessary in most of the cases)
* More robust (in case of name collisions among the different injectors, you make sure that the right injector is used)
* Better IDE support (because the annotations provide elements for each configuration which is available for that specific injector, i.e. filter only for OSGi services)

The follow annotations are provided which are tied to specific injectors:

| **Annotation** | **Supported Optional Elements** | **Injector** | **Description** |
| --- | --- | --- | --- |
| @ScriptVariable | injectionStrategy and name | script-bindings | Injects the script variable defined via [Sling Bindings](https://cwiki.apache.org/confluence/display/SLING/Scripting+variables). If name is not set the name is derived from the method/field name. |
| @ValueMapValue | injectionStrategy, name and via | valuemap | Injects a ValueMap value. If via is not set, it will automatically take resource if the adaptable is the SlingHttpServletRequest. If name is not set the name is derived from the method/field name. |
| @ChildResource | injectionStrategy, name and via | child-resources | Injects a child resource by name. If via is not set, it will automatically take resource if the adaptable is the SlingHttpServletRequest. If name is not set the name is derived from the method/field name. |
| @RequestAttribute | injectionStrategy, name and via | request-attributes | Injects a request attribute by name. If name is not set the name is derived from the method/field name. |
| @ResourcePath | injectionStrategy, path, and name | resource-path | Injects a resource either by path or by reading a property with the given name. |
| @OSGiService | injectionStrategy, filter | osgi-services | Injects an OSGi service by type. The filter can be used give an OSGi service filter. |
| @Self | injectionStrategy | self | Injects the adaptable itself. If the field type does not match with the adaptable it is tried to adapt the adaptable to the requested type. |
| @SlingObject | injectionStrategy | sling-object | Injects commonly used sling objects if the field matches with the class: request, response, resource resolver, current resource, SlingScriptHelper |

[Hints](https://sling.apache.org/documentation/bundles/models.html" \l "hints)

Those annotations replace @Via, @Filter, @Named, @Optional, @Required, @Source and @Inject. Instead of using the deprecated annotation element optional you should rather use injectionStrategywith the values DEFAULT, OPTIONAL or REQUIRED (see also [SLING-4155](https://issues.apache.org/jira/browse/SLING-4155)). @Default may still be used in addition to the injector-specific annotation to set default values. All elements given above are optional.

[Custom Annotations](https://sling.apache.org/documentation/bundles/models.html" \l "custom-annotations)

To create a custom annotation, implement the org.apache.sling.models.spi.injectorspecific.StaticInjectAnnotationProcessorFactory interface. This interface may be implemented by the same class as implements an injector, but this is not strictly necessary. Please refer to the [injectors in Git](https://github.com/apache/sling-org-apache-sling-models-impl/tree/master/src/main/java/org/apache/sling/models/impl/injectors) for examples.

**[Specifying an Alternate Adapter Class (since 1.1.0)](https://sling.apache.org/documentation/bundles/models.html" \l "specifying-an-alternate-adapter-class-since-1-1-0-)**

By default, each model class is registered using its own implementation class as adapter. If the class has additional interfaces this is not relevant.

The @Model annotations provides an optional adapters attribute which allows specifying under which type(s) the model implementation should be registered in the Models Adapter Factory. Prior to *Sling Models Impl 1.3.10* only the given class names are used as adapter classes, since 1.3.10 the implementation class is always being registered implicitly as adapter as well (see [SLING-6658](https://issues.apache.org/jira/browse/SLING-6658)). With this attribute it is possible to register the model to one (or multiple) interfaces, or a superclass. This allows separating the model interface from the implementation, which makes it easier to provide mock implementations for unit tests as well.

Example:

@Model(adaptables = Resource.class, adapters = MyService.class)

**public** **class** **MyModel** **implements** **MyService** {

// injects fields and implements the MyService methods

}

In this example a Resource can be adapted to a MyService interface, and the Sling Models implementation instantiates a MyModel class for this.

It is possible to have multiple models implementing the same interface. By default Sling Models will just take the first one ordered alphabetically by the class name. Applications can provide an OSGi service implementing the ImplementationPicker SPI interface which could use context to determine which implementation can be chosen, e.g. depending an a tenant or content path context. If multiple implementations of the ImplementationPicker interface are present, they are queried one after another in order of their service ranking property, the first one that picks an implementation wins.

**[Associating a Model Class with a Resource Type (since 1.3.0)](https://sling.apache.org/documentation/bundles/models.html" \l "associating-a-model-class-with-a-resource-type-since-1-3-0-)**

The @Model annotation provides an optional resourceType attribute which allows for model classes to be associated with one or more resource types. This is used in three different ways.

In the case of multiple model classes implementing the same interface, the class with the "closest" resource type will be used when adapting to the interface.

The ModelFactory service interface has methods Object getModelFromResource(Resource) and Object getModelFromRequest(SlingHttpServletRequest) which will dynamically determine the adapter class based on the Resource using its type. In the case of the SlingHttpServletRequest method, it uses the request's Resource object (i.e. by calling request.getResource())

The resource type is also used as part of the Exporter framework (see next section).

**[Exporter Framework (since 1.3.0)](https://sling.apache.org/documentation/bundles/models.html" \l "exporter-framework-since-1-3-0-)**

Sling Models objects can be exported to arbitrary Java objects through the Sling Models Exporter framework. Model objects can be programatically exported by calling the ModelFactory method exportModel(). This method takes as its arguments:

* the model object
* an exporter name
* a target class
* a map of options

The exact semantics of the exporting will be determined by an implementation of the ModelExporter service interface. Sling Models currently includes a single exporter, using the Jackson framework, which is capable of serializing models as JSON or transforming them to java.util.Map objects.

In addition, model objects can have servlets automatically registered for their resource type (if it is set) using the @Exporter annotation. For example, a model class with the annotation

@Model(adaptable = Resource.**class**, resourceType = "myco/components/foo")

@Exporter(name = "jackson", extensions = "json")

results in the registration of a servlet with the resource type and extension specified and a selector of 'model' (overridable through the @Exporter annotation's selector attribute). When this servlet is invoked, the Resource will be adapted to the model, exported as a java.lang.String (via the named Exporter) and then returned to the client.

**[Registration of Sling Models classes via bnd plugin](https://sling.apache.org/documentation/bundles/models.html" \l "registration-of-sling-models-classes-via-bnd-plugin)**

With the Sling Models bnd plugin it is possible to automatically generated the necessary bundle header to register the Sling Models classes contained in the Maven bundle project - either with maven-bundle-plugin or with bnd-maven-plugin. By default the plugin generates a Sling-Model-Classes header (only compatible with Sling Models Impl since version 1.3.4, see [SLING-6308](https://issues.apache.org/jira/browse/SLING-6308)).

Example configuration:

<**plugin**>

<**groupId**>org.apache.felix</**groupId**>

<**artifactId**>maven-bundle-plugin</**artifactId**>

<**extensions**>true</**extensions**>

<**configuration**>

<**instructions**>

<**\_plugin**>org.apache.sling.bnd.models.ModelsScannerPlugin</**\_plugin**>

</**instructions**>

</**configuration**>

<**dependencies**>

<**dependency**>

<**groupId**>org.apache.sling</**groupId**>

<**artifactId**>org.apache.sling.bnd.models</**artifactId**>

<**version**>1.0.0</**version**>

</**dependency**>

</**dependencies**>

</**plugin**>

If a Sling-Model-Packages or Sling-Model-Classes was already manually defined for the bundle the bnd plugin does nothing. So if you want to migrate an existing project to use this plugin remove the existing header definitions.

If you want to generate a bundle header compliant with Sling Models < 1.3.4 (i.e. Sling-Model-Packages) you need to specify the attribute generatePackagesHeader=true. An example configuration looks like this

<**configuration**>

<**instructions**>

<**\_plugin**>org.apache.sling.bnd.models.ModelsScannerPlugin;generatePackagesHeader=true</**\_plugin**>

</**instructions**>

</**configuration**>

**[Caching](https://sling.apache.org/documentation/bundles/models.html" \l "caching)**

By default, Sling Models do not do any caching of the adaptation result and every request for a model class will result in a new instance of the model class. However, there are two notable cases when the adaptation result can be cached. The first case is when the adaptable extends the SlingAdaptable base class. Most significantly, this is the case for many Resource adaptables as AbstractResource extends SlingAdaptable. SlingAdaptable implements a caching mechanism such that multiple invocations of adaptTo() will return the same object. For example:

// assume that resource is an instance of some subclass of AbstractResource

ModelClass object1 = resource.adaptTo(ModelClass.class); // creates new instance of ModelClass

ModelClass object2 = resource.adaptTo(ModelClass.class); // SlingAdaptable returns the cached instance

**assert** object1 == object2;

While this is true for AbstractResource subclasses, it is notably **not** the case for SlingHttpServletRequest as this class does not extend SlingAdaptable. So:

// assume that request **is** some SlingHttpServletRequest object

ModelClass object1 = request.adaptTo(ModelClass.class); // creates **new** instance **of** ModelClass

ModelClass object2 = request.adaptTo(ModelClass.class); // creates another **new** instance **of** ModelClass

assert object1 != object2;

Since API version 1.3.4, Sling Models *can* cache an adaptation result, regardless of the adaptable by specifying cache = true on the @Model annotation.

@Model(adaptable = SlingHttpServletRequest.class, cache = **true**)

**public** **class** **ModelClass** {}

...

// assume that request is some SlingHttpServletRequest object

ModelClass object1 = request.adaptTo(ModelClass.class); // creates new instance of ModelClass

ModelClass object2 = request.adaptTo(ModelClass.class); // Sling Models returns the cached instance

**assert** object1 == object2;

When cache = true is specified, the adaptation result is cached regardless of how the adaptation is done:

@Model(adaptable = SlingHttpServletRequest.class, cache = **true**)

**public** **class** **ModelClass** {}

...

// assume that request is some SlingHttpServletRequest object

ModelClass object1 = request.adaptTo(ModelClass.class); // creates new instance of ModelClass

ModelClass object2 = modelFactory.createModel(request, ModelClass.class); // Sling Models returns the cached instance

**assert** object1 == object2;

**[Via Types (Since API 1.3.4/Implementation 1.4.0)](https://sling.apache.org/documentation/bundles/models.html" \l "via-types-since-api-1-3-4-implementation-1-4-0-)**

As discussed in the [Via](https://sling.apache.org/documentation/bundles/models.html#via) section above, it is possible to select a different adaptable than the original value using the @Via annotation. The following standard types are provided (all types are in the package org.apache.sling.models.annotations.via)

| **@Via type value** | **Description** |
| --- | --- |
| BeanProperty(default) | Uses a JavaBean property from the adaptable. |
| ChildResource | Uses a child resource from the adaptable, assuming the adaptable is a Resource. |
| ForcedResourceType | Creates a wrapped resource with the provided resource type. If the adaptable is a SlingHttpServletRequest, a wrapped request is created as well to contain the wrapped resource. |
| ResourceSuperType | Creates a wrapped resource with the resource type set to the adaptable's resource super type. If the adaptable is a SlingHttpServletRequest, a wrapped request is created as well to contain the wrapped resource. |

[Custom Via Type](https://sling.apache.org/documentation/bundles/models.html" \l "custom-via-type)

Defining your own type for the @Via annotation is a two step process. The first step is to create a marker class implementing the @ViaProviderType annotation. This class can be entirely empty, e.g.

**public** **class** **MyCustomProviderType** **implements** **ViaProviderType** {}

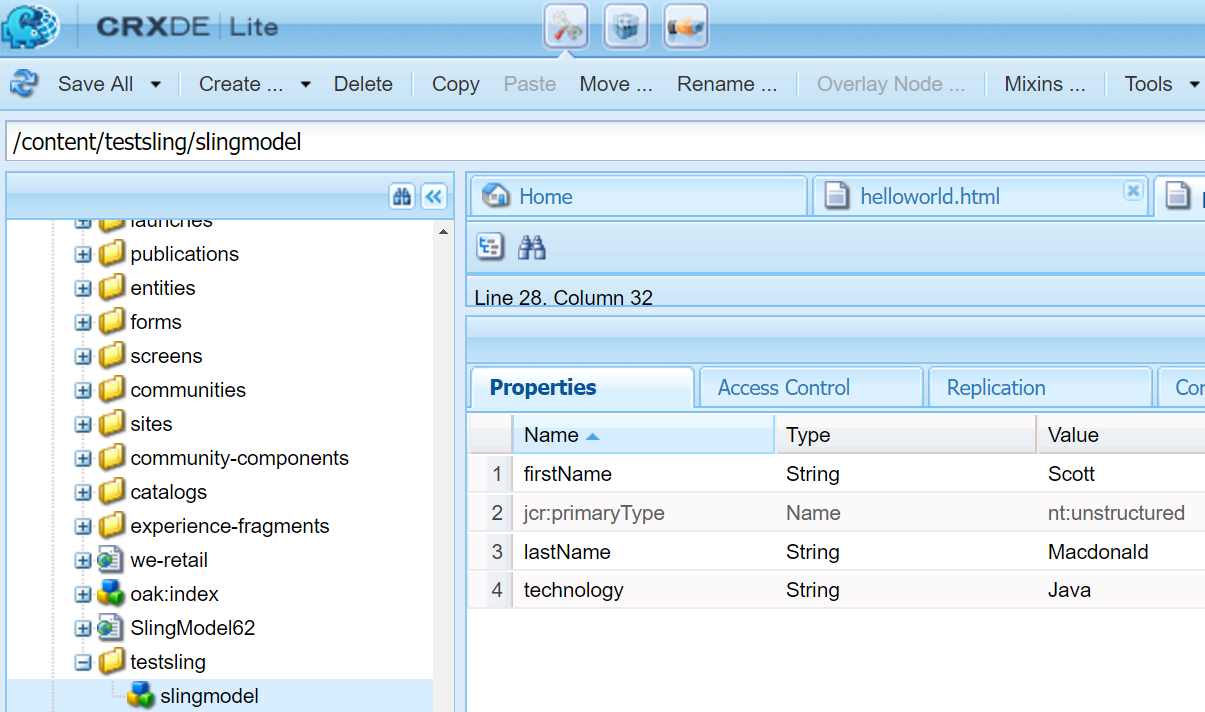
The second step is to create an OSGi service implementing the ViaProvider interface. This interface defines two methods:

* getType() should return the marker class.
* getAdaptable() should return the new adaptable or ViaProvider.ORIGINAL to indicate that the original adaptable should be used.

Consider the following Java class named **UserInfo**.

|  |  |
| --- | --- |
|  | **package SlingModel63.core;**    **import javax.inject.Inject;**    **import org.apache.sling.api.resource.Resource;**  **import org.apache.sling.models.annotations.Model;**    **@Model(adaptables = Resource.class)**  **public class UserInfo {**  **@Inject**  **private String firstName;**  **@Inject**  **private String lastName;**  **@Inject**  **private String technology;**    **public String getFirstName() {**  **return firstName;**  **}**  **public String getLastName() {**  **return lastName;**  **}**  **public String getTechnology() {**  **return technology;**  **}**    **}** |

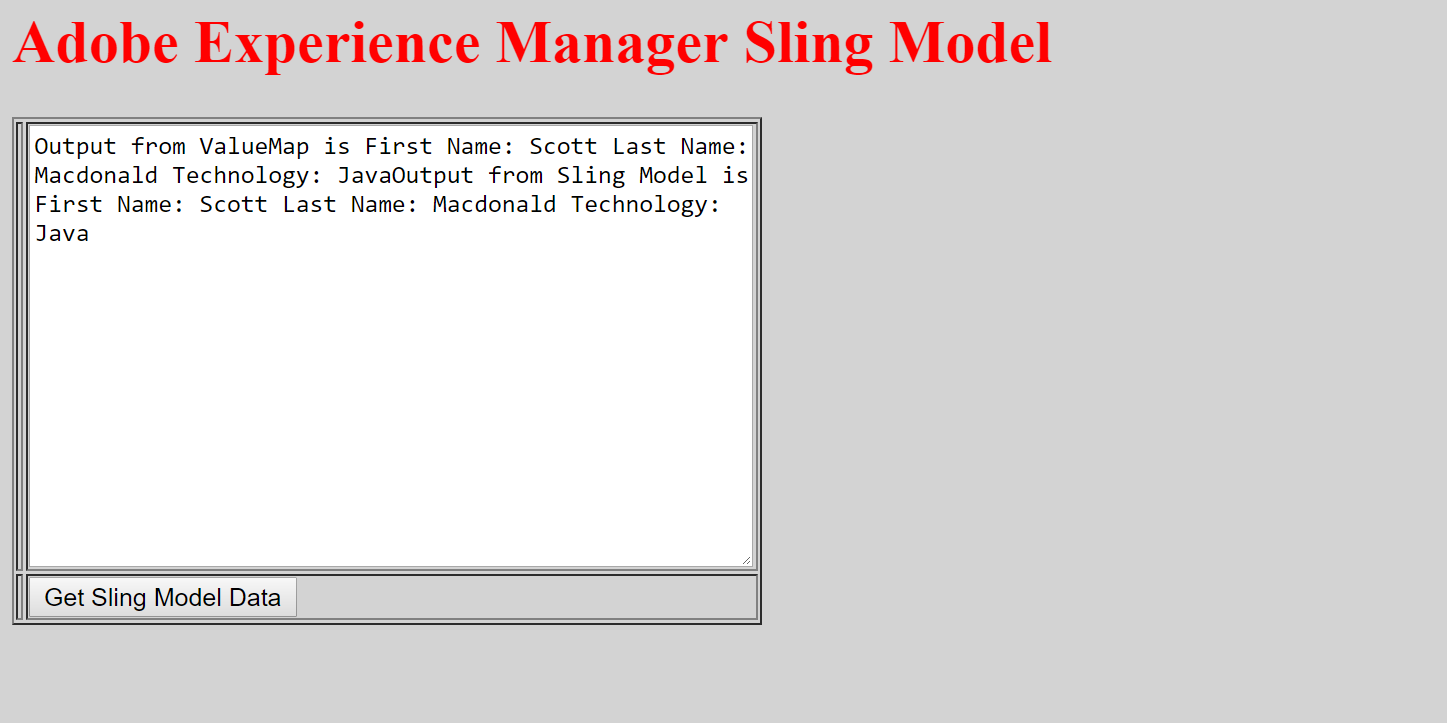
As you can see in the code example, the **@Model** annotation is used. Likewise, each data member in the **UserInfo**class is annotated using the **@Inject** annotation. This Java class is mapped to a Sling resource, like the one shown in the following illustatration.



*A JCR node that is used in the Sling Model*

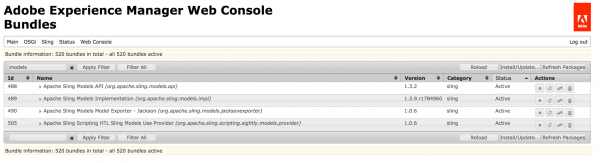
Notice that the class members in the **UserInfo**class map to the String properties that belong to the **/content/testsling/slingmodel** node. This article walks you through creating a Sling Servlet that uses a Sling Model to map to this resource.

The following illustration shows the output of the Sling Servlet that uses Sling Models. Notice that the values in the node properties are displayed.



*An AEM web page displaying values returned from a Sling Servlet that uses Sling Models*

Because AEM 6.3 is built on top of Sling Models API and Implementation version 1.3, and the latest version for those are also 1.3, you don’t need to manually import the updated bundles to AEM in order to use the 1.3 features (for example, Exporter Framework and Associating a Model Class with a Resource Type).

[](https://blogs.perficient.com/adobe/files/2017/05/Screen-Shot-2017-05-25-at-1.27.33-PM.png)

You will just need to check #3 and 4 from above to make sure your project is set up properly for Sling Models.

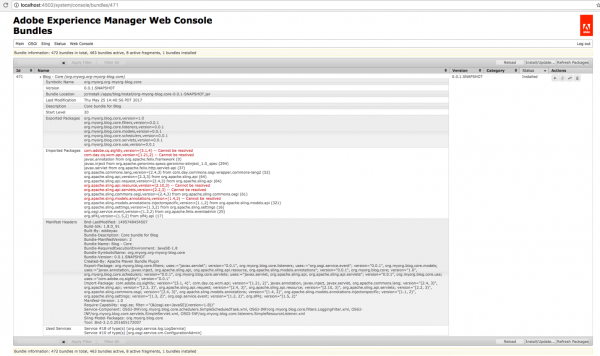
If Sling has a new major release that you want to use, you can still manually import them into your 6.3 server and check in Adobe’s documentation for additional package to be installed in order to support the code at that time.

Here are some sample of dependencies you may need for your project to use Sling Models.

1. **<dependency>**
2. **<groupId>**org.apache.sling**</groupId>**
3. **<artifactId>**org.apache.sling.models.api**</artifactId>**
4. **<version>**1.3.0**</version>**
5. **<scope>**provided**</scope>**
6. **</dependency>**
7. **<dependency>**
8. **<groupId>**org.apache.sling**</groupId>**
9. **<artifactId>**org.apache.sling.models.jacksonexporter**</artifactId>**
10. **<version>**1.0.4**</version>**
11. **<scope>**provided**</scope>**
12. **</dependency>**

For a complete reference, I have created a blog project that’s available on [Github](https://github.com/guangweiyao/blog" \t "_blank). I will start using it to put source codes for the demonstration of all my blogs.

Notice that this project was created from Adobe Archetype 10 and set up for AEM 6.3. If you deploy the code to AEM 6.2 or lower, you may find some imported packaged cannot be resolved. To fix that, you can either lower the version number for related packages in the POM files, or manually upload the latest version of related bundles in AEM bundles (http://localhost:4502/system/console/bundles).

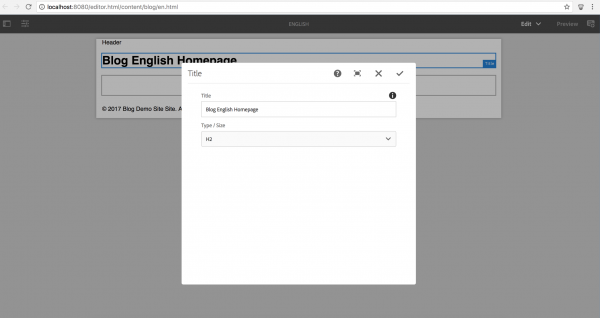
[](https://blogs.perficient.com/adobe/files/2017/05/Screen-Shot-2017-05-25-at-2.41.47-PM.png)

Since Sling Models are annotation-driven Plain Old Java Objects (POJOs), annotations are used a lot. They allow you to map resource properties, assign default values, inject OSGI services and much more.

For example, in my blog project, I have a title component at /apps/blog/components/content/title. And I created two classes related to it, one with WCMUsePojo API, org.myorg.blog.core.use.TitleUse, the other with Sling Models, org.myorg.blog.core.models.TitleModel.

In the TitleModel class, the @Model is required to register the Java class as a Sling Model. You can specify adaptables, resourceType, injection strategy and validation in this annotation. In that class, I was adapting a SlingHttpServletRequest object, and associating the class with the title resource type, so it can be used in the Sling Model exporter later on. Usage of Model annotation can be found in [Sling API documentation](https://sling.apache.org/apidocs/sling8/org/apache/sling/models/annotations/Model.html).

The @Exporter is for Jackson exporter, which basically scan through all the getters that follow the naming convention in the class and serialized them into JSON format. You will need to add a resourceType element in the @Model, and point it to your component’s resourceType. You can request the Sling Models JSON for title component with a “model” selector and “json” extension.

[](https://blogs.perficient.com/adobe/files/2017/05/Screen-Shot-2017-05-25-at-2.00.18-PM.png)

[https://blogs.perficient.com/adobe/files/2017/05/Screen-Shot-2017-05-25-at-1.59.48-PM-600x62.png](https://blogs.perficient.com/adobe/files/2017/05/Screen-Shot-2017-05-25-at-1.59.48-PM.png)

I then used couple injector-specific annotations to get the sling binding object, and map component properties.

The @PostConstruct is usually for initModel()or other methods to call after model option is created. It’s similar to the activate() method in WCMUsePojo that it holds the main logic for processing the data.

You can find reference of all available annotations [here](https://sling.apache.org/documentation/bundles/models.html).

Lastly in TitleModel class, you will find the getters for the class to return the value for HTL to consume.

After completing my experiment of creating two Java classes for a title component, I’ve found the major differences between implementing with WCMUsePojo and Sling Models are:

1. WCMUsePojo will need to be extend from that class, whereas Sling Models can be standalone class with @Model annotation and no keyword
2. With Sling Models, it’s simpler and cleaner to retrieve common objects or property values, instead of writing more line of code to use API
3. You may use Felix annotation @Reference to reference to an available OSGI service, whereas in Sling Models, you will use @Inject or @OSGiService
4. With Sling Models API 1.3, you can serialize the model and export it as a JSON file with Jackson exporter, so your front-end application can leverage the same model. It’s not available for WCMUsePojo.
5. For WCMUsePojo, you will need to overwrite the activate() method, whereas in Sling Models, your init method will be called in the @PostConstruct annotation

It’s also possible to create your own custom injectors and annotations. For custom injectors, you will create a OSGI service that implements the org.apache.sling.models.spi.Injector interface. And for custom annotations, you will create a OSGI service that implements the org.apache.sling.models.spi.injectorspecific.StaticInjectAnnotationProcessorFactor interface. Also, you can use servicing ranking to change priority of the injectors. They are invoked from lowest number to highest. Available injectors ranking and information can be found [here](https://sling.apache.org/documentation/bundles/models.html#available-injectors).

And if you are planning to develop custom injector and annotation, you can reference the [source code of OOTB injectors](http://svn.apache.org/repos/asf/sling/trunk/bundles/extensions/models/impl/src/main/java/org/apache/sling/models/impl/injectors/) and [ACS AEM Commons project](https://github.com/Adobe-Consulting-Services/acs-aem-commons).

In terms of presentation layer, which is HTML Template Language (HTL) in AEM, I find both WCMUsePojo and Sling Models are being used the same way, with data-sly-use block statement and calling the getter from Java class.

**SLING MODEL EXPORTER**

Apache Sling Models 1.3.0 introduces Sling Model Exporter, an elegant way to export or serialize Sling Model objects into custom abstractions. This article juxtaposes the traditional use-case of using Sling Models to populate HTL scripts, with leveraging the Sling Model Exporter framework to serialize a Sling Model into JSON.

Sling Model Exporter was introduced in Sling Models v1.3.0. This new feature allows new annotations to be added to Sling Models that define how the Model an can be exported as a different Java object, or more commonly, serialized into a different format such as JSON.

Apache Sling provides a Jackson JSON exporter to cover the most common case of exporting Sling Models as JSON objects for consumption by programmatic web consumers such as other web services and JavaScript applications.

Sling Model Exporter is a feature of the Apache Sling Project and not directly bound to the AEM product release cycle. Sling Model Exporter is compatible with AEM versions:

**AEM 6.3**

* No extra dependencies required

**AEM 6.2**

* Requires installation of
  + [Models API 1.3.0+](https://sling.apache.org/downloads.cgi#components)
  + [Models Implementation 1.3.0+](https://sling.apache.org/downloads.cgi#components)
  + [Models Jackson Exporter 1.0.0](https://sling.apache.org/downloads.cgi#components)
  + [AEM 6.2 Communities/Livefyre - FP2](https://www.adobeaemcloud.com/content/marketplace/marketplaceProxy.html?packagePath=/content/companies/public/adobe/packages/cq620/social/fp2/AEM-6.2-Communities-Livefyre-Feature-Pack-2)

**AEM 6.1**

* Requires installation of
  + [Models API 1.3.0+](https://sling.apache.org/downloads.cgi#components)
  + [Models Implementation 1.3.0+](https://sling.apache.org/downloads.cgi#components)
  + [Models Jackson Exporter 1.0.0](https://sling.apache.org/downloads.cgi#components)
  + [AEM 6.1 Communities/Livefyre - FP6](https://www.adobeaemcloud.com/content/marketplace/marketplaceProxy.html?packagePath=/content/companies/public/adobe/packages/cq610/social/featurepack/fp6/AEM-6.1-Communities-Livefyre-Feature-Pack-6)

Sling Model Exporter is perfect for leveraging Sling Models that already contain business logic that support HTML renditions via HTL (or formerly JSP), and expose the same business representation as JSON for consumption by programmatic Web services or JavaScript applications.

Enabling Exporter support on a Sling Model is as easy as adding the **@Exporter** annotation to the Java class.

For AEM 6.1 and AEM 6.2 projects, ensure the Sling Models 1.3.x API dependency is included ABOVE the AEM uber-jar dependency.

|  |  |
| --- | --- |
|  | **<!-- Put this above AEM 6.2 and lower Uber-Jar -->**  **<dependency>**  **<groupId>org.apache.sling</groupId>**  **<artifactId>org.apache.sling.models.api</artifactId>**  **<version>1.3.0</version>**  **<scope>provided</scope>**  **</dependency>** |

Ensure the Maven SCR Plug-in as 1.23.0+.

|  |  |
| --- | --- |
|  | **<plugin>**  **<groupId>org.apache.felix</groupId>**  **<artifactId>maven-scr-plugin</artifactId>**  **<version>1.23.0</version>**  **</plugin>** |

Sling Model Exporter supports passing per-model Exporter options to the Exporter implementation to drive how the Sling Model is finally exported. These options generally apply "globally" to how the Sling Model is exported, versus per data point which can be done via inline annotations described below.

Jackson Exporter options include:

* [Mapper Feature options](http://static.javadoc.io/com.fasterxml.jackson.core/jackson-databind/2.8.5/com/fasterxml/jackson/databind/MapperFeature.html)
* [Serialization Feature options](http://static.javadoc.io/com.fasterxml.jackson.core/jackson-databind/2.8.5/com/fasterxml/jackson/databind/SerializationFeature.html)

Exporters implementations may also support annotations that can be applied inline on the Sling Model class, that can provide a finer level of control how the data is exported.

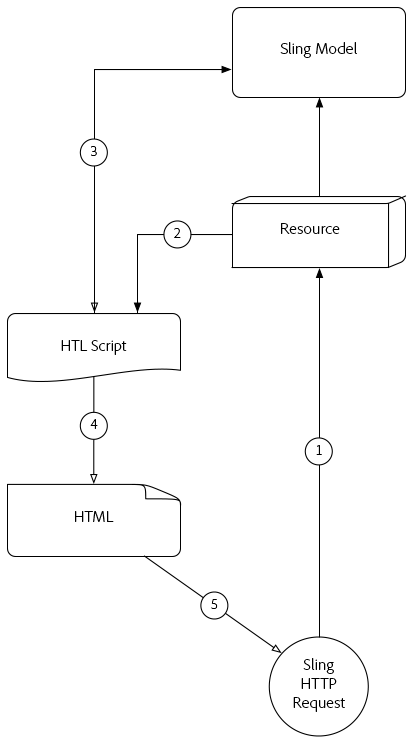
* [Jackson Exporter annotations](https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations)

## Traditional Sling Model request flow

The traditional use-case for Sling Models is to provide a business abstration for a resource or request, which provides HTL scripts (or, previously JSPs) an interface for accessing business functions.

Common patterns are developing Sling Models that represent AEM Components or Pages, and using the Sling Model objects to feed the HTL scripts with data, with an end result of HTML that's displayed in the browser.

### Sling Model Request Flow



1. HTTP GET Request is made for a resource in AEM.

Example: **HTTP GET /content/my-resource.html**

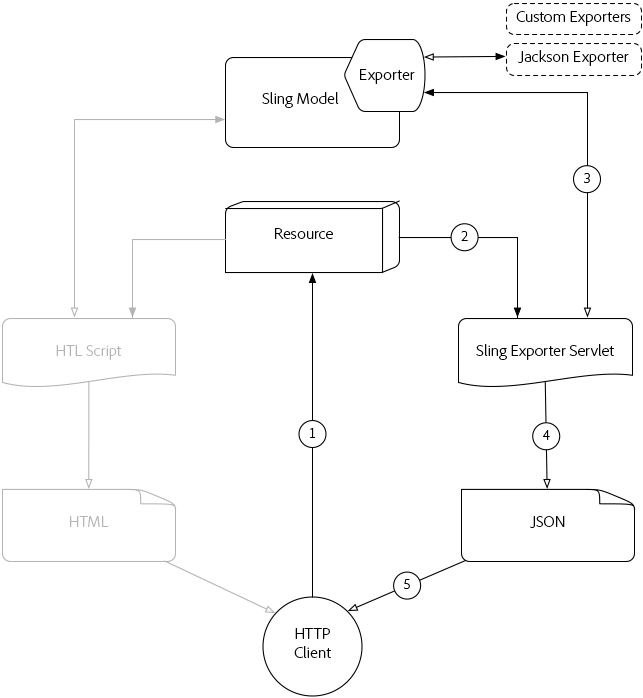
1. Based on the request resource’s **sling:resourceType**, the appropriate Script is resolved.
2. The Script adapts the Request or Resource to the desired Sling Model.
3. The Script uses the Sling Model object to generate the HTML rendition.
4. The HTML generated by the Script is returned in the HTTP Response.

This traditional pattern works well in the context of generating HTML as the Sling Model can be easily leveraged via HTL. Creating more structured data such a JSON or XML is a far more tedious endeavour as HTL doesnt naturally lend itself to the definition of these formats.

## Sling Model Exporter request flow

Apache Sling Model Exporter comes with a Sling provided Jackson Exporter that automatically serializes an "ordinary" Sling Model object into JSON. The Jackson Exporter, while quite configurable, at its core inspecs the Sling Model object, and generates JSON using any "getter" methods as JSON keys, and the getter return values as the JSON values.

The direct serialization of Sling Models allows them to service both normal Web requests with their HTML responses created using the traditional Sling Model request flow (see above), but also expose JSON renditions that can be consumed by web services or JavaScript applications.



This flow describes the flow using the provided Jackson Exporter to produce JSON output. Use of custom exporters follow the same flow but with their output format.

1. HTTP GET Request is made for a resource in AEM with the selector and extension registered with the Sling Model’s Exporter.

Example: **HTTP GET /content/my-resource.model.json**

1. Sling resolves the the requested resource’s **sling:resourceType**, selector and extension to a dynamically generated Sling Exporter Servlet, which is mapped to the Sling Model with Exporter.
2. The resolved Sling Exporter Servlet invokes the Sling Model Exporter against the Sling Model object adapted from the request or resource (as determined by the Sling Models adaptables)
3. The exporter serializes the Sling Model based on the Exporter Options and Exporter-specific Sling Model annotations and returns the result to the Sling Exporter Servlet.
4. The Sling Exporter Servlet returns the JSON rendition of the Sling Model in the HTTP Response.

**Note:**

While the Apache Sling project provides the Jackson Exporter that serializes Sling Models to JSON, the Exporter framework also supports custom Exporters. For example, a project could implement a custom Exporter that serializes a Sling Model into XML.

**Note:**

Not only does Sling Model Exporter serialize Sling Models, it can also export them as Java objects. The exporting to other Java objects does not play a role in the HTTP Request flow, and thus does not appear in the above diagram.

**DOCUMENTATION PAGE WITH VIDEO TUTORIALS**

<https://helpx.adobe.com/experience-manager/kt/platform-repository/using/sling-model-exporter-tutorial-develop.html>

**HELPFUL BLOGS & LINKS**

<http://sgaem.blogspot.in/2017/08/deep-dive-on-sling-model-in-aem-63-part.html>

**UNIT TESTING FOR SLING MODELS**

As you may know, unit testing and test-driven development (TDD) are important for making sure your code complies with the design, is scalable among your team, and provides automated regression. Often times, the JUnit test and component back-end Java code come hand in hand. An AEM developer who writes the component logic is also responsible to write the JUnit test code for the class.

[In my previous blog post](https://blogs.perficient.com/adobe/2017/05/30/how-to-switch-from-wcmusepojo-to-sling-models-in-aem/), I talked about how you can switch from WCMUsePojo API to Sling Models for Adobe Experience Manager (AEM) component. Here in part two, I am going to discuss how you can make the switch in terms of the JUnit test.

I have seen two approaches to writing the JUnit test class for component class that extends the WCMUsePojo class. One is using Mockito and mocking each AEM/Sling object (i.e. bindings, resource, page, properties, SlingHttpServletRequest, SlingHttpServletResponse…) you will need for your test class, wiring those mocks with each other using Mockito’s when().thenReturn(), or PowerMockito.doReturn().when(), and activating the ComponentUse class with the properties and bindings passed in from each test case. The second approach is using [AemContext class](http://wcm.io/testing/aem-mock/index.html) from [wcm.io](http://wcm.io/) and setting it as your JUnit test rule, and then using a test content json file in your Test Resources Root folder to provide test page/resource content for your test cases.

If you haven’t heard of wcm.io, it’s an open-source project that is hosted on [GitHub](https://github.com/wcm-io) and provides handy libraries and extensions for AEM developers. We will be focusing on the AEM Mocks feature in wcm.io specifically, as it can be used and helpful for both WCMUsePojo and Sling Models test classes.

One of the reasons I like using AEM Mocks here is that it’s very robust and provides access to all mocked environments in Sling project ([Sling Mocks](https://sling.apache.org/documentation/development/sling-mock.html), [OSGI Mocks](https://sling.apache.org/documentation/development/osgi-mock.html) and [JCR Mocks](https://sling.apache.org/documentation/development/jcr-mock.html)) and also all the context objects (i.e. SlingBindings, resource, page, properties, SlingHttpServletRequest…), so you don’t need to create and wire mocked objects individually and you can then write cleaner test codes. Secondly, it fully supports Sling Models.

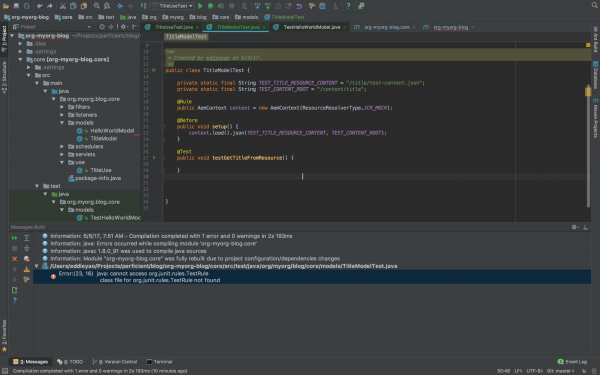
In here, I am taking the title component I developed from [my previous blog](https://blogs.perficient.com/adobe/2017/05/30/how-to-switch-from-wcmusepojo-to-sling-models-in-aem/) as an example. I have written two sample JUnit test classes, one is for TitleUse.java, which extends WCMUsePojo, the other is for TitleModel.java, which is a Sling Models class. You can find all source code in [my GitHub project](https://github.com/guangweiyao/blog).

*Note: this was tested on AEM 6.2, 6.3*

When you use the AemContext object in your test class, and your project skeleton was generated by Adobe Maven Archetype 10, like mine, you may find several issues when you run the test code. Those can be fixed by modifying the Maven dependencies in your POM file. Issues:

1. java.lang.NoClassDefFoundError: org/junit/rules/TestRule

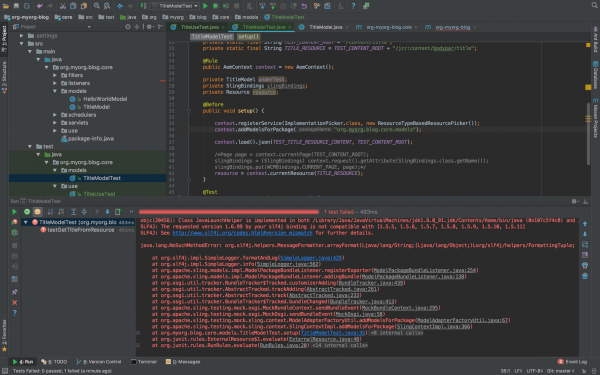
java.lang.ClassNotFoundException: org.junit.rules.TestRule

[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-06-at-8.01.23-AM.png)

Resolved by: validating the maven dependencies of test scope, here’s a working copy in my parent POM:

1. **<dependency>**
2. **<groupId>**junit**</groupId>**
3. **<artifactId>**junit**</artifactId>**
4. **<version>**4.12**</version>**
5. **<scope>**test**</scope>**
6. **</dependency>**
7. **<dependency>**
8. **<groupId>**org.slf4j**</groupId>**
9. **<artifactId>**slf4j-simple**</artifactId>**
10. **<version>**1.7.6**</version>**
11. **<scope>**test**</scope>**
12. **</dependency>**
13. **<dependency>**
14. **<groupId>**org.mockito**</groupId>**
15. **<artifactId>**mockito-core**</artifactId>**
16. **<version>**1.10.19**</version>**
17. **<scope>**test**</scope>**
18. **</dependency>**
19. **<dependency>**
20. **<groupId>**org.powermock**</groupId>**
21. **<artifactId>**powermock-api-mockito**</artifactId>**
22. **<version>**1.6.4**</version>**
23. **<scope>**test**</scope>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.powermock**</groupId>**
27. **<artifactId>**powermock-module-junit4**</artifactId>**
28. **<version>**1.6.4**</version>**
29. **<scope>**test**</scope>**
30. **</dependency>**
31. **<dependency>**
32. **<groupId>**org.apache.commons**</groupId>**
33. **<artifactId>**commons-imaging**</artifactId>**
34. **<version>**1.0-R1534292**</version>**
35. **<scope>**test**</scope>**
36. **</dependency>**
37. **<dependency>**
38. **<groupId>**io.wcm**</groupId>**
39. **<artifactId>**io.wcm.testing.aem-mock**</artifactId>**
40. **<version>**2.1.0**</version>**
41. **<scope>**test**</scope>**
42. **<exclusions>**
43. **<exclusion>**
44. **<groupId>**org.apache.commons**</groupId>**
45. **<artifactId>**commons-imaging**</artifactId>**
46. **</exclusion>**
47. **</exclusions>**
48. **</dependency>**
49. <!-- for testing we need the new ResourceTypeBasedResourcePicker -->
50. **<dependency>**
51. **<groupId>**org.apache.sling**</groupId>**
52. **<artifactId>**org.apache.sling.models.impl**</artifactId>**
53. **<version>**1.3.0**</version>**
54. **<scope>**test**</scope>**
55. **</dependency>**

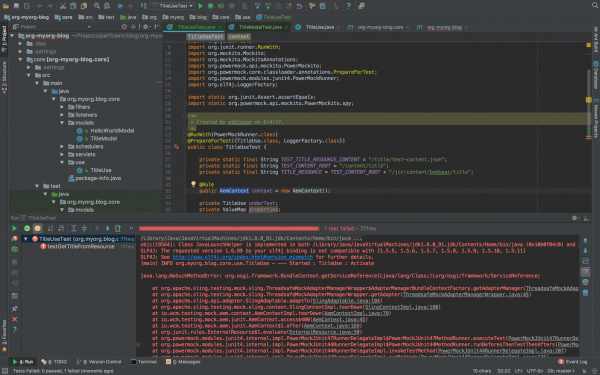
2. lang.NoSuchMethodError:  
org.osgi.framework.BundleContext.getServiceReference(Ljava/lang/Class;)Lorg/osgi/framework/ServiceReference;

[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-07-at-8.56.06-PM.png)

Resolved by: validating the version of the osgi-core library, here’s a working copy in my parent POM:

1. **<dependency>**
2. **<groupId>**org.osgi**</groupId>**
3. **<artifactId>**osgi.core**</artifactId>**
4. **<version>**6.0.0**</version>**
5. **<scope>**provided**</scope>**
6. **</dependency>**
7. **<dependency>**
8. **<groupId>**org.osgi**</groupId>**
9. **<artifactId>**osgi.cmpn**</artifactId>**
10. **<version>**6.0.0**</version>**
11. **<scope>**provided**</scope>**
12. **</dependency>**

3. lang.NoSuchMethodError: org.slf4j.helpers.MessageFormatter.arrayFormat(Ljava/lang/String;[Ljava/lang/Object;]Lorg/slf4j/helpers/FormattingTuple;

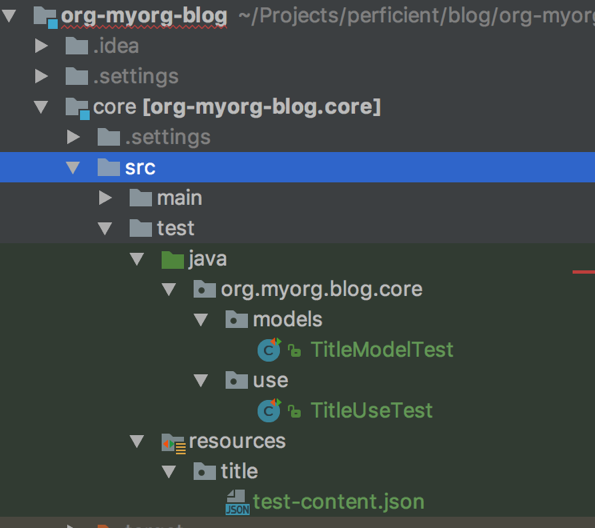
[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-07-at-8.33.26-AM.png)

Resolved by: validating the version of the slf4j library, here’s a working copy in my parent POM:

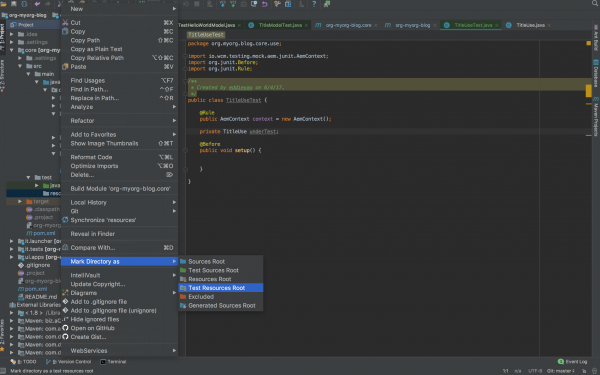
1. **<dependency>**
2. **<groupId>**org.slf4j**</groupId>**
3. **<artifactId>**slf4j-api**</artifactId>**
4. **<version>**1.7.6**</version>**
5. **<scope>**provided**</scope>**
6. **</dependency>**
7. **<dependency>**
8. **<groupId>**org.slf4j**</groupId>**
9. **<artifactId>**slf4j-simple**</artifactId>**
10. **<version>**1.7.6**</version>**
11. **<scope>**test**</scope>**
12. **</dependency>**

The package version numbers above are based on AEM 6.3, since I am using Sling Models API 1.3 features like associating a model class with a resource type and exporter framework. If you are on AEM 6.2 or lower, you may find some imported packages cannot be resolved in your bundle, you can either manually install the Sling Models 1.3 bundle, or adjust your package version number. Simply check the unresolved bundle in Package Dependencies (http://localhost:4502/system/console/depfinder) and locate the maven dependency in your POM file. Also, be aware of the version number of uber-jar or other bundles to provide AEM APIs and match those with your AEM version.

The Adobe Maven Archetype (10 or 11) didn’t generate test resources structure, so if you want to use test resources for your test classes, you will need to set up the structure in your project.

[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-07-at-8.46.31-AM.png)

Basically you will create a folder under /core/src/test/resources, and put your component test resources in there. In IntelliJ, after you create the directory, you can mark it as Test Resource Root. The resource files you put in the resources folder can be loaded from your test class.

[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-04-at-4.47.36-PM.png)

Now that you have everything set up for you to write JUnit test cases for your component class, here’re the steps:

1. Create the test class in the same package path under /core/src/test/java.

2. Know the JUnit annotations that you are going to use.

a. If you are using wcm.io’s AEM mock context object, you will need the [@Rule annotation](http://junit.org/junit4/javadoc/4.12/org/junit/Rule.html). The rule will run any Before methods, then the Test method, and finally any After methods, throwing an exception if any of these fail, so you don’t need to define the object repeatedly in those scenarios.

b. @Before annotation is used for set up methods (like assigning common mocked values, loading test content and binding it to Sling request variables) to be called before the actual test cases run.

c. @Test annotation holds statements for each test case to be run for the test class.

3. For Sling Models specifically:

a. If you are using wcm.io’s AEM mock context object, you will need to register models from package by context.addModelsForPackage("org.myorg.blog.core.models");

b. If you are using [resourceType feature](https://sling.apache.org/documentation/bundles/models.html" \l "associating-a-model-class-with-a-resource-type-since-130) in Sling Models API 1.3, you may register [ResourceTypeBasedResourcePicker service](https://fisheye.apache.org/browse/sling/trunk/bundles/extensions/models/impl/src/main/java/org/apache/sling/models/impl/ResourceTypeBasedResourcePicker.java?hb=true) in mocked OSGI environment, by context.registerService(ImplementationPicker.class, new ResourceTypeBasedResourcePicker());

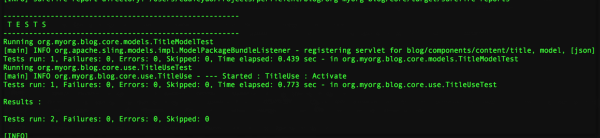
c. If you are using @ScriptVariable in your Sling Models class to provide script objects (i.e. currentPage, properties…), you may use [SlingBindings](https://sling.apache.org/apidocs/sling5/org/apache/sling/api/scripting/SlingBindings.html) class in your test class to add those objects by

slingBindings = (SlingBindings) context.request().getAttribute(SlingBindings.class.getName());  
slingBindings.put(WCMBindings.*CURRENT\_PAGE*, page);

d. Call the Sling Models class by underTest = context.request().adaptTo(TitleModel.class);

4. Write different test cases based on your code design and logic

5. Run your unit test class

[](https://blogs.perficient.com/adobe/files/2017/06/Screen-Shot-2017-06-07-at-9.56.43-PM.png)

Differences between writing test class for WCMUsePojo (ComponentUseTest.java) and for Sling Models (ComponentModelTest.java):

1. In ComponentUseTest you mock/spy an instance of your use class, whereas in ComponentModelTest you call the Sling Models class directly;

2. In ComponentUseTest you heavily rely on Mockito/PowerMockito to mock the objects returned from WCMUsePojo APIs, whereas in ComponentModelTest you can just set up the context objects and Sling Models will be able to inject the properties/script variables from those context objects;

3. In ComponentUseTest you initialize the use class by calling activate() method, whereas in ComponentModelTest you initialize the Sling Models class by calling adaptTo() method.

I hope after this article, you get more knowledge about writing JUnit test class for your component Java code and know the difference between writing test class for WCMUsePojo and for Sling Models. If you want to know more about unit testing and AEM mocks, I found these two decks online that are helpful, one is an [AEM GEMS resource](https://docs.adobe.com/content/ddc/en/gems/From-Unit-Testing-to-Integration-Test-of-an-AEM-Application/_jcr_content/par/download/file.res/AEM%20GEMS%20-%20Testing%2012-14-2016.pdf), the other is an [adaptTo() presentation](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwixzdioo7TUAhVLxmMKHRarBOIQFgguMAE&url=https%3A%2F%2Fadapt.to%2Fcontent%2Fdam%2Fadaptto%2Fproduction%2Fpresentations%2F2016%2FadaptTo2016-Unit-Testing-with-Sling-and-AEM-Mocks-Stefan-Seifert.pdf%2F_jcr_content%2Frenditions%2Foriginal.media_file.download_attachment.file%2FadaptTo2016-Unit-Testing-with-Sling-and-AEM-Mocks-Stefan-Seifert.pdf&usg=AFQjCNE8LSsJOXDWMnGg_4zSnkuQnm0f9w&sig2=6tAVmteW4r0TtfBTy-fniw).