Getting started

Getting started with NEBA only requires two simple steps:

Step 1: Download and integrate NEBA

install the delivery package and integrate the NEBA API into your project.

Step 2: Add an OSGi blueprint context to your bundle(s)

Place one or more blueprint XML file into OSGI-INF/blueprint, like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:bp="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
  http://www.osgi.org/xmlns/blueprint/v1.0.0
  http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.3.xsd">

   <context:component-scan base-package="bundle.package.impl" />
</beans>
```

Listing 1: Sample Blueprint Context XML declaration

That's it - you may now proceed to Write resource models or Spring controllers - for instance, use the sample project below as a starting point.

NEBA directive: A sample project for NEBA

Checkout the continuously updated NEBA sample project on Github

Writing @ResourceModels

Resources, resource types and resource models

Background: How NEBA uses resource types in Sling

The Sling webframework is a REST architecture implementation. Consequently, it treats each thing it works with (every piece of content, every script, servlet) as a Res
typed using the property *sling:resourceType*. Resource types - very much like types in many other languages - also support inheritance by means of a *sling:resourceS*

Resource types are used by Sling to determine the view(s) that shall be used to render the resource. It does so by adding a prefix such as /apps or /libs to the resourc
whether it finds views for the resource in the corresponding directory.

Furthermore, any content stored in the JCR is represented by a javax.jcr.Node. Nodes are strongly typed - they always possess a *primaryType* as well as an arbitrary r
Both the primary type and the mixin types can have *super types*.

> In order to provide a JAVA model for a resource, it thus makes sense to either associate it with the resource's sling:resourceType or the resource's Node type. This as using a @ResourceModel annotation.

Let's assume we need a simple JAVA model for a sling component called "carousel". The carousel has the sling:resourceType "foundation/components/carousel". The co following properties:

```
{
 "jcr:primaryType": "nt:unstructured",
 "playSpeed": "6000",
 "jcr:lastModifiedBy": "admin",
 "pages": [
  "/content/geometrixx/en/events/techsummit",
  "/content/geometrixx/en/events/userconf"
 ],
 "jcr:lastModified": "Tue Oct 05 2010 14:14:27 GMT+0200",
 "transTime": "1000",
 "sling:resourceType": "foundation/components/carousel",
 "listFrom": "static"
}
```

Listing 2: A typical JSON view of JCR repository content

Create the following class in your bundle package (i.e. within or below the "my.bundle.namespace" package)

```
@ResourceModel(types = "foundation/components/carousel")
public class Carousel {
}
```

Listing 3: Application of the @ResourceModel annotation on a Bean

The resource type given in the @ResourceModel annotation does not have to be the exact resource type of the resource. One may also specify any resource super type of node type or mixin type. for example, the resource super type of the carousel could be "foundation/components/list". Using this resource type in a @ResourceModel woul can provide general models for super types, and more specific models for derived types, e.g. a generic model for pages and a more specific model for specific page types.

> ### Background: The @ResourceModel annotation
>
> Classes annotated with @ResourceModel are regular spring beans: The @ResourceModel annotation itself is annotated with @Component (a spring stereotype) and With these annotations, the class is detected by Spring's classpath scanning. It is entirely possible to give a model a different scope (such as singleton) if it represent content, such as configuration content stored under /etc/. Furthermore, all capabilities of the spring container are available to a model, such as IOC (dependency inje 330 @Inject annotation, bean lifecycle annotations such as @PostConstruct and @PreDestroy, @Scheduled methods and so forth.

Once a bundle with NEBA models and a blueprint context is active, all of its models must appear in the NEBA model registry (/system/console/modelregistry) in the sling blueprint context likely failed to start. In such cases analyzing the error.log is advisable.

Using models in views

Resource models can be automatically provided in either HTL (Sightly) or JSP views using the neba.js or neba:defineObjects tag library, respectively:

```
<sly data-sly-use.m="/apps/neba/neba.js"> ... </sly>
```

Listing 4: Using the neba.js to obtain the most specific NEBA model in an HTL (Sightly) view

```
<%@taglib prefix="neba" uri="http://neba.io/1.0"%>
<neba:defineObjects />
```

Listing 5: Importing the NEBA taglib namespace

Both are looking up the most specific model for the current resource. The neba:defineObjects tag always publishes the model into the scripting context under the key "m". you could access the model like so:

```
${m}
```

Listing 6: Acessing the NEBA model in HTL (Sightly) or JSP

The *most specific model* is the one who's *type* attribute points to the closest type within a resource's type hierarchy. For example, if the resource has the type "cq:Page" ar "nt:base", a model for "cq:Page" is *more specific* than one for "nt:base". If there is more than one most specific model, e.g. two models for "cq:Page", neba.js and the neba:d provide the model automatically. In such cases, you could either explicitly adapt to the desired model type or specify the desired model's bean name (you can lookup the b registry), like so:

```
<sly data-sly-use.m="${'/apps/neba/neba.js' @ beanName=carousel}"> ... </sly>
```

Listing 7: Resolving a NEBA model by name in HTL (Sightly)

```
<neba:defineObjects useModelNamed="carousel"/>
```

Listing 8: Resolving a NEBA model by name in JSP

However, one may also explicitly adapt the current resource to the model, like so:

```
<sly data-sly-use.m="my.package.MyModel"> ...</sly>
```

Listing 9: Directly adapting to a NEBA model in HTL (Sightly)

```
<% MyModel model = resource.adaptTo(my.package.MyModel.class); %>
```

Listing 10: Directly adapting to a NEBA model in JSP

Mapping properties and resources to models

Following, we will map the properties of the carousel sample content, as defined in the previous chapter:

```
{
  "jcr:primaryType": "nt:unstructured",
  "playSpeed": "6000",
  "jcr:lastModifiedBy": "admin",
  "pages": [
    "/content/geometrixx/en/events/techsummit",
    "/content/geometrixx/en/events/userconf"
  ],
  "jcr:lastModified": "Tue Oct 05 2010 14:14:27 GMT+0200",
  "transTime": "1000",
  "sling:resourceType": "foundation/components/carousel",
  "listFrom": "static"
}
```

Listing 11: A typical JSON view of JCR repository content

Let's take "pages", "playSpeed", "transTime" and add them to the model:

```
@ResourceModel(types = "foundation/components/carousel")
public class Carousel {
    private String playSpeed;
    private String transTime;
    private List<String> pages;

    public String getPlaySpeed() {
        return playSpeed;
    }
    public String getTransTime() {
        return transTime;
    }
    public List<Resource> getPages() {
        return pages;
    }
}
```

Listing 12: A simple @ResourceModel with mapped fields

Now build and deploy your bundle again and output the properties in the view like so:

```
Play speed: ${m.playSpeed}<br />
Trans time: ${m.transTime}<br />
Pages: ${m.pages}
```

You will see the following output:

```
Play speed: 6000
Trans time: 1000
Pages: ["/content/geometrixx/en/events/techsummit", ...]
```

Annotations for resource to model mapping

All NEBA annotations can also supported as meta-annotations - thus, custom annotations annotated with any of the annotations provided by NEBA are treated as if the cc annotation was directly applied.

Resolving references with the @Reference annotation

NEBA automatically maps the properties of a Resource to fields named like the properties, unless the field is annotated with @Unmapped. Notice that play speed and tran above example, whereas one would expect them to be integers. This is simply because these values are defined to be Strings in the component's CQ dialog. Had they beer could also retrieve them as integers. You will also notice that "pages" is null. And no wonder - the "pages" property is actually of type "String[]", so the field should also be of List<Resource>. However, NEBA offers a simple way to declare that a field contains one ore more *references* to other Resources. Simply add the @Reference annotation t

```
@Reference
private List<Resource> pages;
```

Listing 13: Using the @Reference annotation

Deploy, and load the page again. Now all referenced pages get listed:

```
Play speed: 6000
Trans time: 1000
Pages: [JcrNodeResource, type=cq:Page, superType=null, path=/content/geometrixx/en/events/techsummit, JcrNodeResource, type=cq:Page, superType=null,
path=/content/geometrixx/en/events/userconf]
```

When NEBA detects a @Reference annotation, it assumes that the *value* of the corresponding property (here: "pages") is one or more resource paths. It obtains the corres provides them as a Collection (or a single resource, if the annotated field is not a collection type). However, you are not limited to using "Resource" for your reference. The provided there is a "Page" model the referenced resources can be adapted to:

```
@Reference
private List<Page> pages;
```

Listing 14: Using the @Reference annotation with resource adaptation

Here, NEBA loads the resource referenced in the "pages" property, adapts each resource to "Page" and returns a collection containing the resulting Page instances.

One can also alter the path of the reference prior to resolution by appending a relative path segment to the reference(s), like so:

```
@Reference(append = "/jcr:content")
@Path("pages")
private List<PageContent> pageContents;
```

Listing 15: Appending paths to the resources resolved via the @Reference annotation

Here, instead of resolving and adapting the paths in the property "pages" directly, "/jcr:content" is appended to all of the paths prior to resolution.

Resolving children with the @Children annotation

While the resource hierarchy is conveniently navigable using the generic Resource model (i.e. using getChildren()), one often has to subsequently adapt the children. Cons adapted in a loop, including a null check for each adaptation result. Here, NEBA offers another powerful annotation: @Children. This annotation can be used on a Collectio (similar to the @Reference-annotation). Then, NEBA injects the children of either the current resource (if no other annotation is present), or of the resource defined by the the field:

```
@Children
private List<Resource> children;
```

Listing 16: Using the @Children annotation

Of course, NEBA will also automatically adapt the children to the generic type of the list - for instance, you could write:

```
@Children
private List<Page> childPages;
```

Listing 17: Using the @Children annotation with adaptation

@Children can be combined with both @Reference and @Path to even fetch the children of a reference resource or a resource designated by a specific path, i.e. the follow

```
@Path("/content/site")
@Children
```

```
private List<Page> countryPages;

@Reference
@Path("link")
@Children
private List<Page> linkChildren;

@Reference
@Children
private List<Page> link;
```

Listing 18: Combining @Children, @Reference and @Path

Furthermore, you may specify a relative path to be resolved below every child using the property "resolveBelowEveryChild" instead of returning the direct children of the de example, you could obtain all nodes called "jcr:content" underneath all children of the current resource, like so:

```
@Children(resolveBelowEveryChild = "jcr:content")
private List<PageContent> childPageContents;
```

Listing 19: Altering the paths of the resources resolved by @Children

Using the @Path annotation to specify property names or resource paths

Now, lets add another property to the model, jcr:lastModified.

```
private Date lastModified;
```

This will not yet work - NEBA mapps the properties by *name*, in this case the name contains characters ("jcr:") unsuitable for a field name. In this case, use the @Path annc property from which the field's value shall be obtained:

```
@Path("jcr:lastModified")
private Date lastModified;
```

Listing 20: Using the @Path annotation

Now the property is mapped! @Path has even more interesting features, for instance allowing absolute and relative paths. Try this:

```
@Path("/content/geometrixx/en")
private Resource en;
```

Listing 21: Using the @Path annotation with absolute paths

Furthermore, one can use placeholders in the @Path to dynamically provide path elements. A better version of above example is:

```
@Path("/content/geometrixx/${language}")
private Resource homepage;
```

Listing 22: Using placeholders in the @Path annotation

To resolve the ${language} placeholder, one simply provides a bean implementing the PlaceholderVariableResolver interface. Such a bean instance will be asked to resolv placeholder key, i.e. "language" in the above example. A completely hardcoded example would be:

```
@Service
public class MyVariableResolver implements PlaceholderVariableResolver {
   public String resolve(String variableName) {
      if ("language".equals(variableName)) {
         return "en";
      }
      return null;
   }
}
```

Listing 23: A dummy PlaceholderVariableResolver implementation

The @This annotation

Fields annotated with @This are injected with the current resource, or anything the resource is adaptable to. This annotation is thus not just useful for access to the plain l significant architectural value. Using @This, one may split up a model into multiple aspects and compose them dynamically, like so:

```
@ResourceModel(types = ...)
public class MyModel {
   @This
   // OtherModel models a different aspect of the same resource.
```

```
    private OtherModel resource;
}
```

Listing 24: Using the @This annotation

You can also use the @This annotation to obtain the Resource that is mapped onto the model:

```
@This
private Resource resource;
```

Listing 25: Using the @This annotation to retrieve the resource that is mapped to the model

Models for crosscutting concerns

A @ResourceModel is not restricted to sling:resourceType's - you can also map it to the JCR primary node type or any of the mixin types of a Node. For example, let's assu requirement to implement a custom access protection of *arbitrary* content. In addition, a content manager may *configure* access restriction to *any* content (e.g. pages or a case, we know neither the resource types nor the content paths to which the protection applies in advance. Here, using a mixin node type could help. A mixin can be assig time anywhere in the content hierarchy. It can define additional properties that can be set on the node, such as a set of properties defining how an access restriction is cor protected content is assigned the mixin "mix:AccessRestricted". You may then create a model for any access restricted content like so:

```
@ResourceModel(types = "mix:AccessRestricted")
public class AccessRestricted {
 ....
}
```

Listing 26: Defining a @ResourceModel for a mixin type

Finally, you could adapt to this model (e.g., in a view or filter) to determine whether the corresponding resource has restricted access, and what the restrictions are.

Further reading: JCR repository spec, administering node types.

Lazy loading

Reading data into an object graph - such as NEBA models - bears the risk of loading more data than required for rendering. Especially when there is a significant amount c being able to load models *on demand* rather then up front is *crucial* for building high-performance implementations. To achieve this, NEBA enables lazy-loading model rela

Lazy loading collections of references

All collection-typed references are automatically provided as lazy-loading proxies, for example in case of @Children and @Reference collections:

```
@ResourceModel(types = "...")
public class MyModel {
  @Children
  private List<Page> children; // Provided as a lazy-loading proxy
  @Reference
  private Collection<Page> pages; // Provided as a lazy-loading proxy as well
}
```

Listing 27: Automatic lazy-loading of collection-type instance in NEBA

The contents of these collection are loaded as soon as a collection method - such as get, size, isEmpty, iterator - is called. The lazy-loading behavior for collections is thus

Declaring lazy-loading relationships using the Lazy<T> interface

1:1 relationships are not automatically lazy. In order to make them lazy-loading, NEBA provides the Lazy interface:

```
@ResourceModel(types = "...")
public class MyModel {
  @Reference
  private Lazy<Page> page;
  @Path("/content/path")
  private Lazy<Resource> otherResource;
}
```

Listing 28: Explicit lazy-loading using the Lazy interface

With Lazy, the relationship is loaded when one of Lazy's methods for model retrieval is invoked, such as "orElse".

Background: Why 1:1 relationships cannot be lazy by default

Let us assume there is a resource model that references a resource "r":

```
@ResourceModel(types = "...")
public class MyModel {
 @Reference
 private Resource r;

 public void doSomething() {
   if (r != null) {
     // work with r
   }
 }
}
```

Of course, any client working with "r" must know whether it exists. However, there is no natural "empty" representation for Resource - it is either null or not. Now, if r wa loading proxy that proxy instance would never be null (since determining whether the resource represented by "r" exists would mean having to load it, which defeats th loading).

A user of "r" would only find out that "r" does not exist when accessing a method of the "r" lazy-loading proxy - and receiving an unchecked exception in return, since th the method call as the loaded "r" is null.

Consequently, lazy 1:1 relationships *must* be explicit to allow clients to determine whether the relationship exists. Collections, however, do have a natural representati (isEmpty) and are thus automatically provided as lazy-loading proxies by NEBA.

The Lazy<T> interface provides all features of the JAVA 8 "Optional" type. NEBA automatically provides an implementation that will load the reference upon request. Of co the interface for collection-typed references:

```
@ResourceModel(types = "...")
public class MyModel {
  @Children
  private Lazy<Collection<Page>> children;
  @Reference
  private Lazy<Collection<Page>> pages;
}
```

Listing 29: Explicit lazy loading of collection instances

In this case, NEBA will not provide the collection as a lazy-loading proxy, but simply load it when requested via the Lazy interface implementation.

Performing additional initializations

It is often required to perform some additional initializations after all properties of a @ResourceModel are mapped. NEBA supports this use case with the @PostMapping annotations. Similar to @PostConstruct and @PreDestroy from standard Java, these annotations mark methods to be invoked after all properties of a resource model are resource:

```
@ResourceModel(types = "my/model/type")
public class MyModel {
  @This
  private Resource resource;
  @PostMapping
  public void initializeSomething() {
    // resource is initialized at this point.
    this.resource.adaptTo...
  }
}
```

Listing 30: Using the @PostMapping annotation

Likewise, @PreMapping allows for method execution before any properties of the resource are mapped.

Using Spring MVC in Sling

Background: How Spring MVC is integrated into Sling

By default, Sling supports three kinds of scripts: JSPs, HTL (Sightly) templates and plain Servlets. While the former two are decent to render dynamic resources, i.e. cc components that have no fixed URL, servlets are often used to provide RESTful services with a fixed URL. However, Servlets are quite primitive. They do not provide s and controller or common yet complex features such as form data binding. With NEBA, one can alternatively use all of Spring's MVC features to enable clean, simple fixed URLs.

To avoid conflict with resource resolution and servlet mappings, Spring MVC is integrated into Sling using a Servlet with the fixed URL "/bin/mvc" (you may use servle change this path). Since each distinct path in sling maps to a different resource (servlets are resources, too), the path /bin/mvc/url would not point to the /bin/mvc/ s request handled by the MVC servlet, the pseudo-extension".do" is used. A spring controller with the URL /my/controller/path is thus always addressed using /bin/mvc One may also use an extension and selectors (e.g. /bin/mvc.do/my/controller/path.selector1.selector2.xml).

While NEBA provides a default MVC infrastructure similar to the defaults provided by Spring's DispatcherServlet, it is recommended to explicitly configure MVC support in using <mvc:annotation-driven />. This also enables advanced features, such as automated conversion of controller responses to JSON

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    ...
    xsi:schemaLocation="
     http://www.springframework.org/schema/mvc
     http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
     ...">

  <mvc:annotation-driven />
</beans>
```

Listing 31: MVC schema snippet for blueprint XML declaration

Let's create a simple Controller called "DemoController". We will use Spring's @Controller annotation; other styles (such as mapping by bean names etc.) are also supporte controller is to echo a parameter that we provide in a RESTful way, i.e. as a path element (this allows caching!).

```
@Controller
public class DemoController {
}
```

Listing 32: @Controller annotation usage

A @Controller is automatically detected and registered (as a singleton) by Spring's classpath scanning. Let's use Spring's REST support to both define the URL the control parameter we expect:

```
@Controller
public class DemoController {
  @RequestMapping("/echo/{param}")
  @ResponseBody
  public String echo(@PathVariable("param") String paramToEcho) {
    return paramToEcho;
  }
}
```

Listing 33: An echo @Controller example

That's it. You may now call your controller e.g. with /bin/mvc.do/echo/HelloWorld.txt.

Furthermore, NEBA automatically provides handler method argument resolvers for Sling-specific request elements, SlingHttpServletRequest and -Response, RequestPath ResourceResolver, like so:

```
@Controller
public class DemoController {
  @RequestMapping("/echoTitle")
  @ResponseBody
  public String echo(ResourceResolver resolver, RequestPathInfo info) {
    ...
  }

  @RequestMapping("/echoTitle")
  @ResponseBody
  public String echo(SlingHttpServletRequest request) {
```

```
    ...
  }
}
```

Listing 34: NEBA argument resolvers for controller methods

It is thus not necessary to obtain these elements from the request.

Note that the annotation-driven controllers have a lot more very powerful options. See the Spring MVC documentation for more examples. NEBA also supports Spring's sp redirecting and forwarding views: returning "redirect:/some/path" from a controller method will cause a redirect to /some/path, returning forward:/some/path will forward to /some/path, see also MVC redirecting.

You may also directly work with the response. For this, provide your own org.springframework.web.servlet.View in the org.springframework.web.servlet.ModelAndView pr this way, your View's render method is used to render the response. Alternatively, you may also simply write to the response and either let the controller method return "nu however bad practice and thus discouraged.

Using Sling Scripts as controller views

Since Version 4, NEBA supports using sling scripts to render controller views. NEBA provides a view resolver that resolves resource type names to the corresponding sling to render views. Consider the following controller method:

```
@Controller
public class DemoController {
  @RequestMapping("/myMethod")
  public String echo(@ResourceParam Page page) {
    // do something
    return "app/controllerViews/myView"
  }
}
```

Listing 35: Calling Sling Views from Spring @Controllers

Here, NEBA's view resolver will resolve "app/controllerViews/myView" to the corresponding resource, e.g. "/apps/app/controllerViews/myView". It will then look for a suital type, e.g. "myView.html" for a HTL (Sightly) template or "myView.jsp" for a JSP view. If no such script is found, the view resolver will re-attempt to resolve the default view f view resource. This enables inheritance and overriding - just like for regular Sling resource views.

Note that, contrary to the standard view resolution for Sling resources, controller view resolution does not take into account request methods, selectors or extensions. The address the Spring Controller and cannot be leveraged for view resolution without conflicting the Spring controller mapping.

Each request to a Spring @Controller is associated with a generic Model. The attributes of this model are available as request attributes to Sling Scripts used as controller instance, a model attribute "page" can be accessed via:

```
${page}
```

Listing 36: Accessing Spring model attributes in JSP

In HTL (Sightly) however, request attributes could only be accessed indirectly up to HTL Engine 1.0.20, e.g. via the HTL use API (see for instance SLING-5812). Consequen had to write additional code to obtain the Spring Controller invocation model data. Since NEBA 4.2.0, the Spring Controller Model is provided by the ValueMap representati Resource provided to the view script. Thus, a model attribute "page" could be accessed in HTL (Sightly) - and thus also in JSP - like so:

```
${properties.page}
```

Listing 37: Accessing Spring model attributes in HTL (Sightly)

Resolving resource path parameters with the @ResourceParam annotation

NEBA supports the common use case of handling resource paths in controllers with a convenience annotation:

```
@Controller
public class DemoController {
  @RequestMapping("/echoTitle")
  @ResponseBody
  public String echo(@ResourceParam Page page) {
    return page.getTitle();
  }
}
```

Listing 38: Using the resource param annotations in Spring @Controllers

Here, a request parameter "page" is expected to contain a path to a resource. This resource is resolved and adapted to the parameter type in case the parameter type is nc
@ResourceParameters can be optional or required. In addition, they can have a default value (a default resource path) that will be used in case the parameter is not prese

```
@Controller
public class DemoController {
  @RequestMapping("/echoTitle")
  @ResponseBody
  public String echo(@ResourceParam(defaultValue = "/default/resource/path") Page page) {
    return page.getTitle();
  }
}
```

Listing 39: Using a default value for the @ResourceParam annotation

Here, "page" is implicitly considered optional (since there is a default value)

Furthermore, the content paths resolved by the @ResourceParam annotations can be altered by appending an arbitrary sub-path:

```
@Controller
public class DemoController {
  @RequestMapping("/echoTitle")
  @ResponseBody
  public String echo(@ResourceParam(append = "/jcr:content") PageContent pageContent) {
    return pageContent.getTitle();
  }
}
```

Listing 40: Altering the path resolved by the @ResourceParam annotation

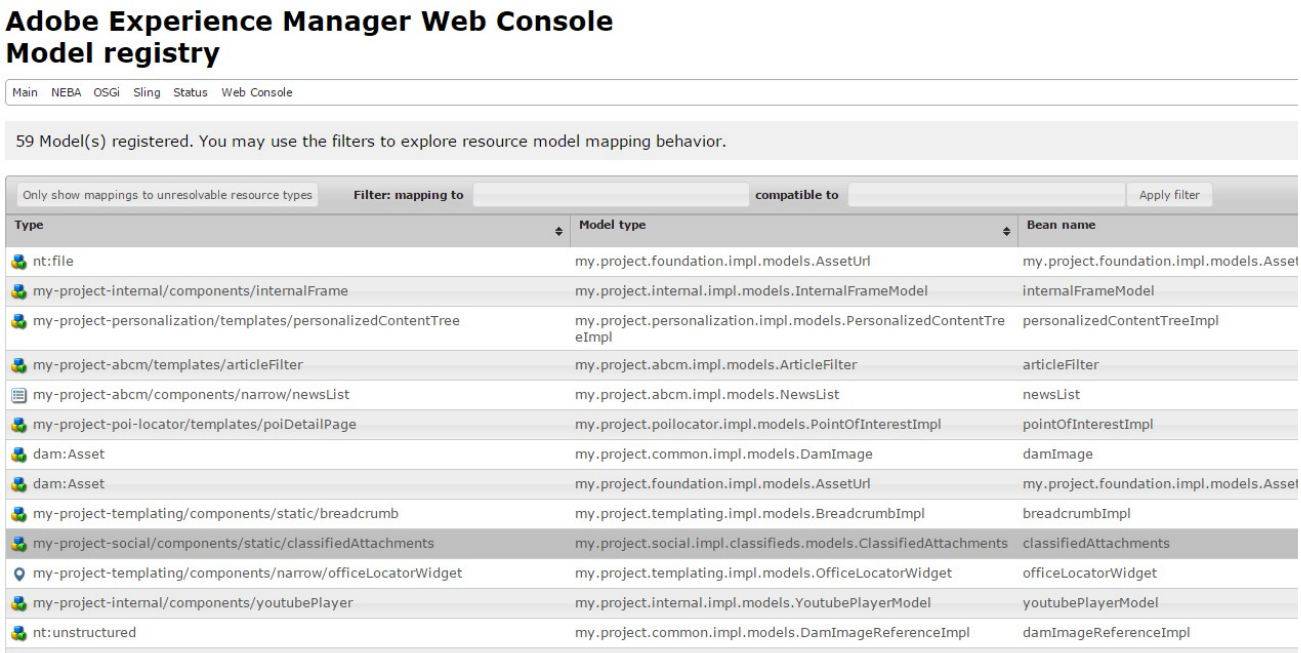
The path specified in "append" is appended to the provided path or the specified default path prior to resolution.

NEBA tooling for developers and administrators

To support developers beyond writing resource models, NEBA ships with a set of useful development and administration tools for exploring resource model characteristic
resolving the resource model / content relationships. Finally, a log viewer integrated into the Felix console greatly improves issue analysis in cases where serial access to

NEBA model registry

The *model registry* allows viewing all currently registered resource models, their source bundle and the resource types they are mapping to. In addition, the models can be
mapping to specific content or models compatible to specific classes.



nt:unstructured	my.project.common.impl.models.DefaultSlingUrl	defaultSlingUrl
nt:unstructured	my.project.foundation.impl.resource.InheritingPropertiesImpl	my.project.foundation.impl.resource.InheiesImpl#0
my-project-templating/templates/toolsOverviewPage	my.project.templating.impl.models.ToolsOverviewPage	toolsOverviewPage
my-project-templating/components/static/leftNavigation	my.project.templating.impl.models.LeftNavigation	leftNavigation
my-project-templating/components/analytics/googleAnalytics	my.project.templating.impl.models.GoogleAnalytics	googleAnalytics
my-project-poi-locator/components/static/actions/poiDataReportAction	my.project.poilocator.impl.models.PoiDataReportAction	poiDataReportAction
my-project-internal/components/photoGalleryFs	my.project.internal.impl.models.PhotoGalleryFullScreenModel	photoGalleryFullScreenModel
my-project-social/components/static/classifiedAttachments/classifiedAttachment	my.project.social.impl.classifieds.models.ClassifiedAttachment	classifiedAttachment

NEBA model statistics

For every detected resource model, NEBA extracts metadata that is reused to efficiently map resources to these models. Whenever resources are mapped to models, NEE the mapping duration. This combined resource model data is available via a JSON REST API in the model statistics console (/system/console/modelstatistics/api/statisti

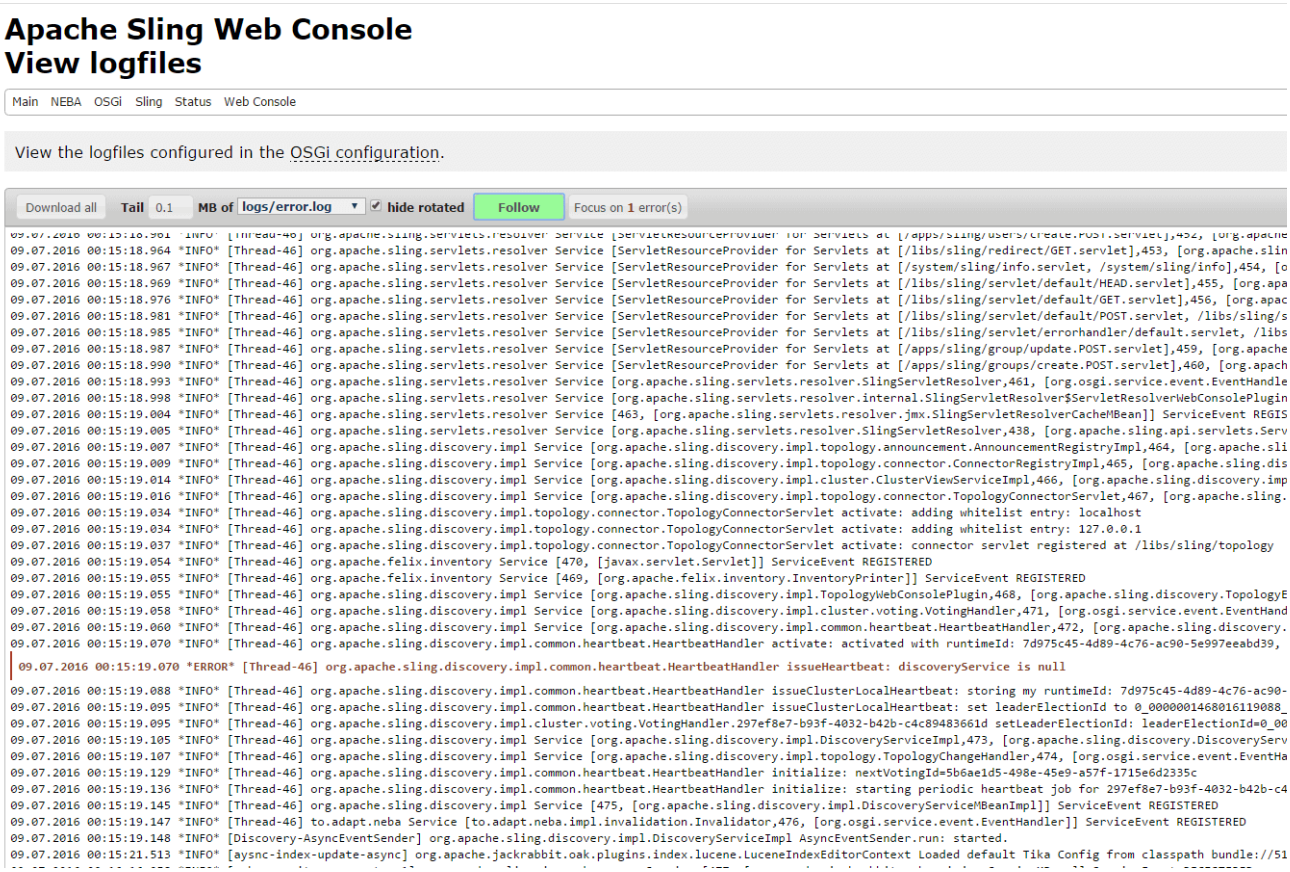
The NEBA model statistics console (/system/console/modelstatistics) uses this data to visualize the corresponding model characteristics. In addition, the console allows display models with certain features, for instance to discover models performing below average and/or excessively loading data from the repository.

This unique feature makes visible what could usually only be discovered through time-consuming analysis using a profiling tool. Furthermore, the model statistics allow re monitoring and analysis on *any system*. This is especially valuable for productive systems - there, content changes may trigger performance degradation and attaching a option. NEBA's model statistics console makes finding performance bottleneck's in resource models fast and easy - further adding to better performing implementations.

The log viewer

Developers do not always have serial access to a system in order to analyze log files - however, being able to investigate the log files of a Sling instance is often key to finc timely. Here, NEBA provides the log viewer Felix console plugin - a safe way to view analyze and obtain the configured log files of a Sling instance for authenticated uses o

The log viewer automatically detects errors in the current log file excerpt of an error.log and allows tailing logfiles in real-time. It also allows downloading logfiles as ZIP co offline analysis.



Support for web-specific bean scopes

Beans in Spring have scopes. By default, beans are application scoped (i.e. singletons). @ResourceModels have prototypical scope, i.e. they are re-created every time they bean factory. In a web context, objects may also be scoped to the current request or the current session.
NEBA provides support for the request scope, i.e. a bean may be defined like so:

```
@Component
@Scope(WebApplicationContext.SCOPE_REQUEST)
public class MyType {
  ...
}
```

Listing 41: Using Spring's @Scope annotation on NEBA models

However, the session scope is currently not supported since it leads to a lot of issues (lack of scalability due to the need of session stickiness to an instance, high memory In a RESTful Application, the session should not be used. User data may instead be stored in a cookie; if server-side state inevitable, storing it in a shared data store (e.g. a relational DB) and retrieving it upon request is a much more scalable, cleaner solution.

Extending NEBA

Implementing custom annotations

NEBA allows registering custom mappers for annotated fields. A custom mapper is an OSGi service implementing the AnnotatedFieldMapper interface. Implementations transient annotated field, including fields directly or indirectly (meta-annotations) annotated with any of NEBA's standard annotations. Thus, AnnotatedFieldMappers are a override NEBA's standard annotation-based mapping behavior.

Customizing the Spring MVC infrastructure

You may provide implementations of Handler adapters, Handler exception resolvers and Handler mappings in your application context and / or use Spring's <mvc> XML n without limitations.

Lifecycle callbacks

Additional functionality is usually provided by simply adjusting the spring context configuration to the project needs (i.e. configuring beans and customizing the bean lifecy

However, there is an additional lifecycle for @ResourceModels beans: after creation, dependency injection and initialization, a resource model is injected with the properties is adapted to the model. For programmatic extensibility, the NEBA API contains the lifecycle callback interface *ResourceModelPostProcessor*.
Providing an OSGi service with this interface allows customization and even overriding of a ResourceModel before and / or after the resource properties are mapped on it.

Caching and cache extension

NEBA's central and most performance critical feature is the adaptation of Resources resource models. There, resource and model resolution, content-to-object mapping a occurs. This core functionality features an extensible caching interface. Any OSGi service implementing the NEBA API interface ResourceModelCache will automatically be cache the result of the resource to model adaptation.

NEBA ships with a safe and sensible default implementation of this interface contained in the core, the *request-scoped resource model cache*. This cache can be disabled (configuration tab). If you would like to provide your own caching implementation, consult the Javadoc of the ResourceModelCache for further information.

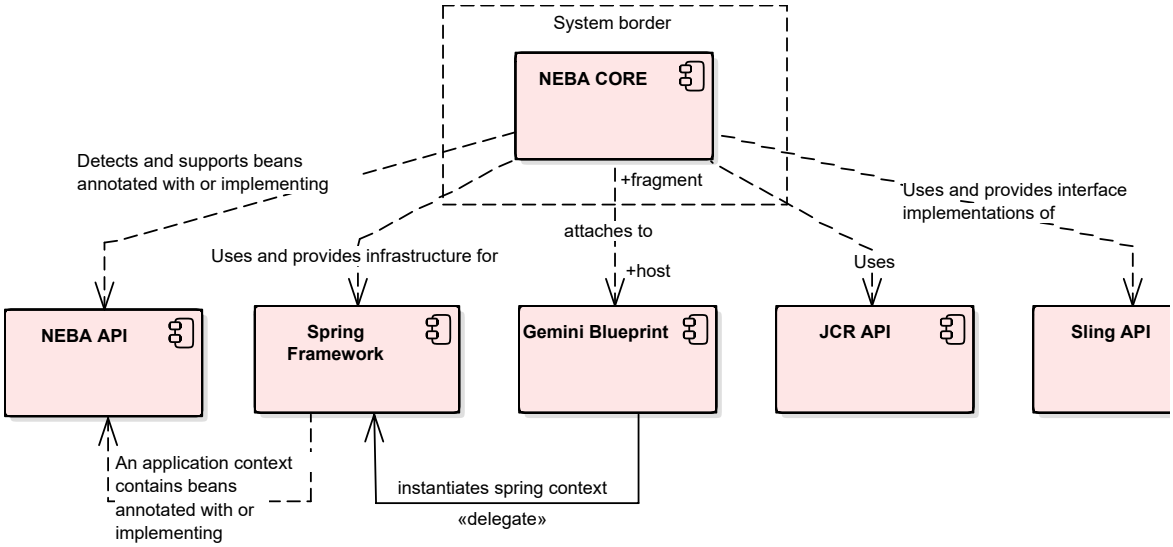
Architecture

The following documentation is not meant to be complete in a sense of covering every implementation detail. It strives to illustrate the core design ideas, as well as the m components and processes of NEBA.

The diagrams in this document are created using Sparx systems Enterprise Architect. You can Download the EA file.

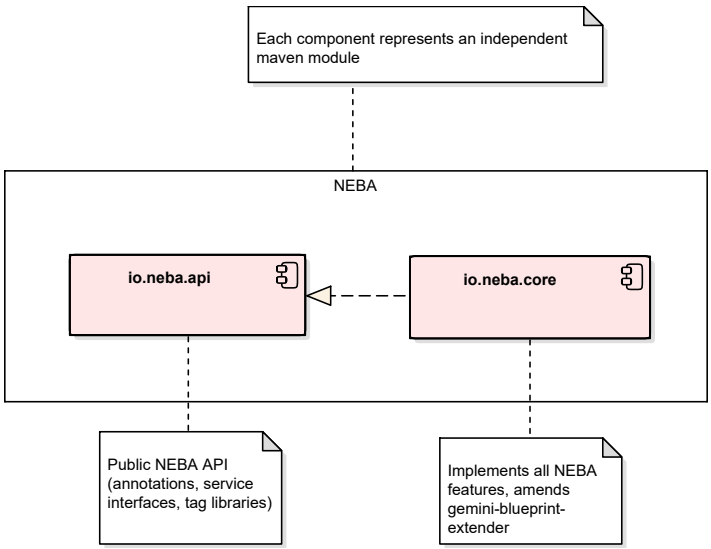
Container view

NEBA amends Sling's open core with further open source software, which in turn is internally structured in modules. We will refer to such component compositions using
container. NEBA is comprised of the following containers:



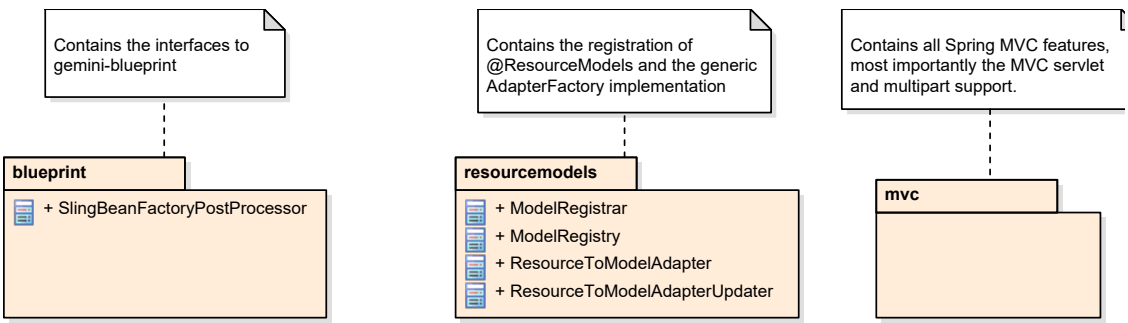
Modules view

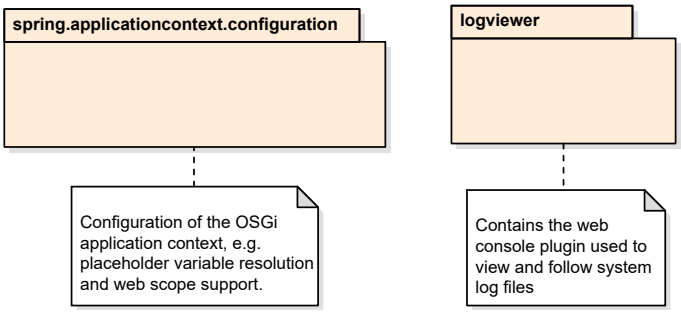
NEBA consists of the following modules. Each module is represented by a single OSGi bundle and a corresponding maven module.



Core packages view

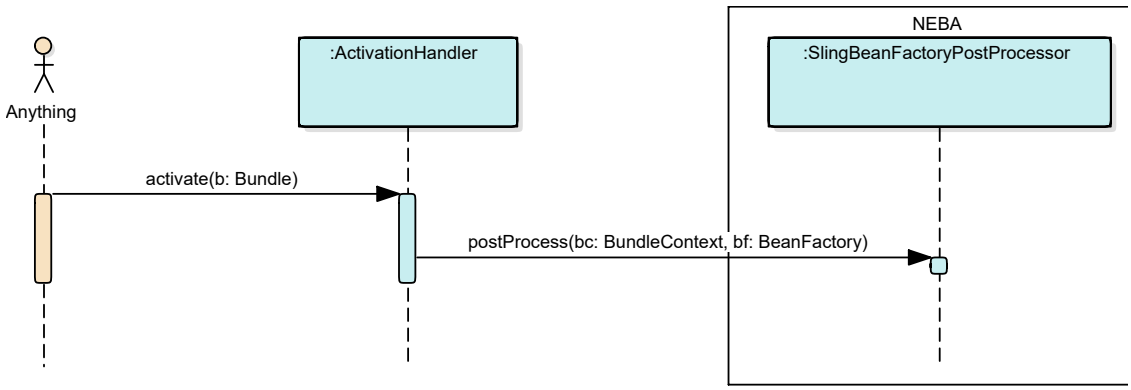
Each package of the core represents a semantically grouped amount of features. The following diagram shows the most important feature groups and illustrates the over
the core





Application context post-processing

The central feature of NEBA is the registration of Spring beans using the NEBA API, for instance beans annotated with @ResourceModel. To do so, NEBA attaches to the c
Spring application contexts. As soon as all beans are successfully created, but prior to the publication of the context, NEBA post-processes the context and looks up bean
annotations or implementing NEBA API interfaces.



Resource model registration and adapter factory provisioning

The registration of resource models and the subsequent management of the resource model metadata for mapping purposes account for the most valuable NEBA featur
a well-designed collaboration of context post-processing, metadata creation and -storage as well as updates of the generic resource model adapter factory. The following
illustrates the complete resource model registration process, from a bundle change event to the publication of the registered resource model metadata via a Sling adapter