Baritus Reference Documentation

Version: 1.0

Table of Contents

Preface .....

1. Concepts .....
1.1. Using POJO's when buidling web applications 1
1.2. Validation 1
1.3. Interceptors 2
2. Using Baritus .....
2.1. Flow of execution 3
2.2. A worked example 3
2.3. Logging 6
2.4. ExecutionParams 7
2.4.1. Available parameters 7
2.5. FormBeanContext 8
2.5.1. References 8
2.5.2. Attributes 8
2.5.3. Utility methods 9
3. Population .....
3.1. Population process 10
3.2. Conversion 10
3.3. Formatting 11
3.4. Custom population 12
4. Validation .....
4.1. After population 13
4.2. An example 13
4.3. Mapped and indexed properties 14
4.4. ValidationActivationRules 15
4.5. Escaping validation 15
4.6. Validator implementations 15
5. Interceptors .....
5.1. Interception 17
5.2. Interceptors 17
5.3. Throwing FlowExceptions 17
5.4. And example 18
5.5. Stacking interceptors 19
6. Extending Baritus .....

Preface

Baritus is an extension of the mature web MVC framework Maverick. Maverick is a minimalist framework which focuses solely on MVC (model 2) logic, allowing you to generate presentation using a variety of templating and transformation technologies. Read more about Maverick here [http://mav.sourceforge.net].

Baritus is an easy to use yet flexible framework that focuses on fine-grained bean population and validation from HTTP request parameters. Furthermore, Baritus gives you an alternative to frameworks that focus on using XML configuration. Instead, Baritus gives ready-to-use defaults (like using introspection for conversion), and lets you use Java to configure validation, population and ways to extend or override the default behaviour.

Baritus builds on Maverick, but does not alter it. Baritus provides a base class, `nl.openedge.baritus.FormBeanCtrl` that is modeled after `org.infohazard.maverick.ctl.FormBeanUser` from the Maverick project. Like `FormBeanUser`, instances of `FormBeanCtrl` are singletons (allthough by extending from `FormBeanCtrlBase` you can use single-use controllers) and use seperate JavaBeans for population.

Chapter 1. Concepts

This chapter explains the building blocks of Baritus.

1.1. Using POJO's when buidling web applications

The ability to work with POJO's (Plain Old Java Objects) is one of the primary focusses of this framework.

The objects that you want to have populated from the HTTP request do not need to extend any base class or implement any interface. This way, you can populate classes from, for example, your domain model without the duplication of code you see often in Java web applications. And as you can nest objects as well, you could even populate several objects in one pass.

Type conversion and property navigation is done automatically, so there's is no need for configuration. Just provide a bean to populate and Baritus will do the rest. Population and is flexible and pluggable on several levels of abstraction. The default population delegate uses Ognl [http://www.ognl.org] to resolve properties and set values.

If there are any errors during population, those errors are stored in the formBeanContext for later use. The original input values are stored in the formBeanContext as well, in order to be able to override the property value with the input value if a population or validation error occured.

As the controls are responsible for creating instances of the objects that should be populated (usually called form beans), no extra configuration is needed. Furthermore, you can do very interesting things with those objects before actually having them populated by Baritus. For instance, if you are working with a ORM tool (like Hibernate [http://www.hibernate.org]), you could load your persistent object, let `Baritus` populate it, and persist the changed object in the command method (perform) of your control. Tricks like these can be serious code savers, compared to where you work with a seperate domain- and form object and where you have to sync those two yourself.

As by default the formBeanContext is reused within one request, you have the option of reusing the allready populated form bean as well. And by setting property populateAndValidate from formBeanContext to false, you could even skip population/ validation in the current and laters steps in the command chain within one request.

Besides populating objects with HTTP request parameters, you can optionally have your objects populated with Maverick configuration parameters, request attributes and session attributes. It's all configurable for each instance of a control, or even for each request if you like.

1.2. Validation

Type conversion during population is in fact your first validation step. For further validation, like checking if a required property was set, or checking that a certain date is actually after the date of today, Baritus provides a flexible and pluggable validation mechanism.

De default provided validation mechanism is based on validation with Java objects that do the validation, and that are registered in the controls that want to use them. The default mechanism is probably powerfull enough for everything you want to do. It is possible however, to plug in one or more additional mechanisms. This way, you could use Commons Validator, FormProc, or possibly another validation framework with Baritus (though at this time no implementations are available).

The objects that do the validation come in two main forms:

- `nl.openedge.baritus.validation.FieldValidator`. A field validator is coupled to one field, like a request parameter or a session attribute. Instances of FieldValidator are registered by name. The name that a field validator was registered with is matched against the population parameters. Field validators are useful for most common cases, like checking the length of an input etc. Field validators that are registered with the same name, are executed in the order that they were registered, UNTIL one of the validators fails or all validators passed.
- `nl.openedge.baritus.validation.FormValidator`. A form validator is not coupled to one specific field, but is coupled to the control. This means that, independent of what fields are actually provided, the instances of FormValidator are executed. Form validators are useful for checking whether a property is not null (note that this differs from checking whether a field is not null) or checking more properties against each other.

Exactely when validators are used can be tuned with ValidationActivationRules. Validation activation rules can be registered on control (/form bean) level. These rules are checked before any validation is done. If one of the rules fails, no validation is done at all. If the control level rules allow validation, the FieldValidators are executed. Instances of FieldValidator can hold a reference to one rule. When a FieldValidator has a rule, this rule is used to check whether the field validator should be executed. If you want to stack rules for a field, you can put them in an instance of `nl.openedge.baritus.validation.impl.NestedValidationActivationRule`, and register this instance with the field validator. If all field validators succeeded, the form validators are executed, possibly dependent on a registered validation rule. If property doFormValidationIfFieldValidationFailed in ExecutionParams is true, form validation will allways be executed, whether the field validation succeeded or not.

If all population actions and validation actions have succeeded succesfully, the command method (perform) will be executed. If one of the actions failed, the command method will not be executed and the view that results from getErrorView (error by default) will be used. If property doPerformIfPopulationFailed from ExectuionParams is set to true (default is false), the command method is execution even if the population/ validation failed.

1.3. Interceptors

Interceptors provide a means to encapsulate cross-cutting code that is executed on pre-defined points in the line of execution.

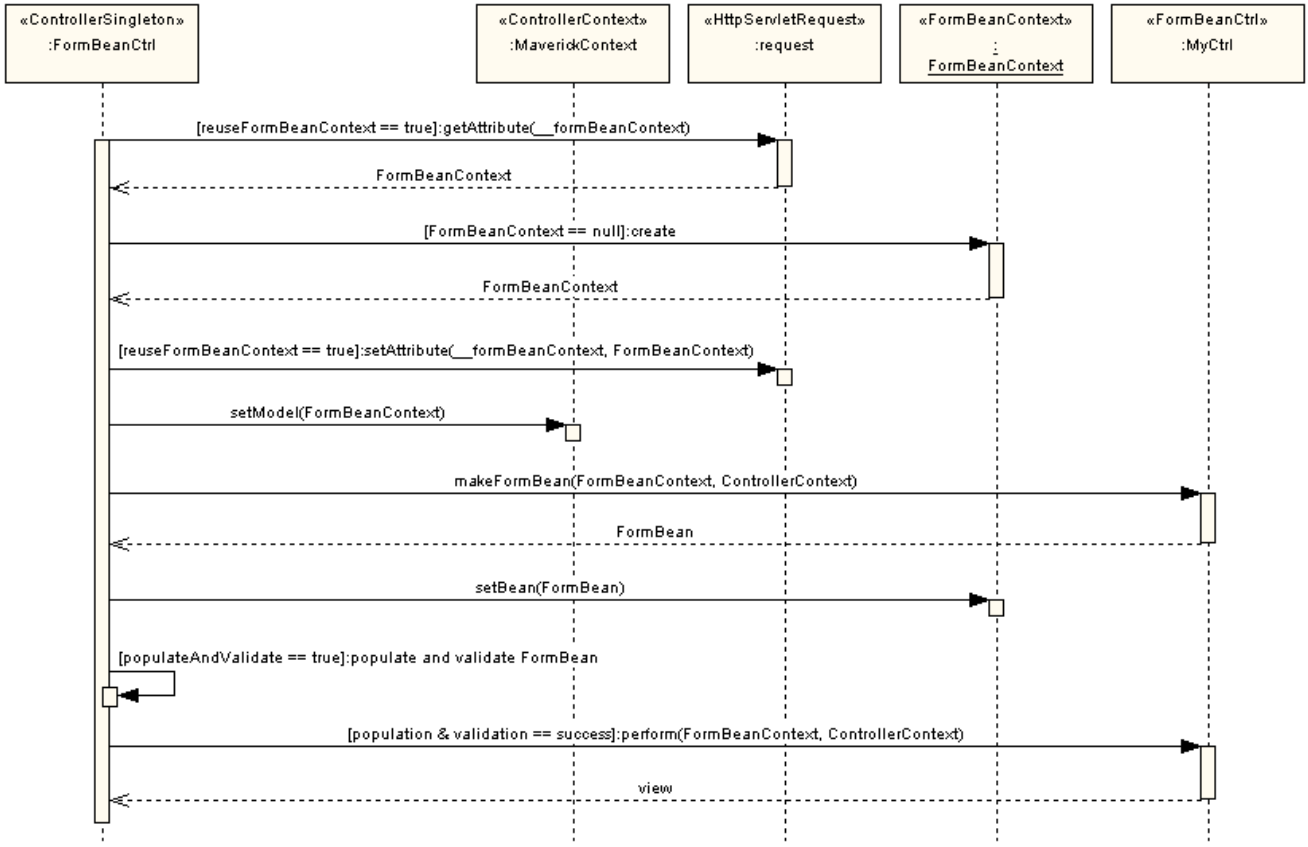
Interceptors can be used to 'enrich' de model (formBeanContext), eg. load data, save session attributes, etc. Furthermore, as interceptors can throw 'FlowExceptions', interceptors can have an effect on the flow. They can overrule the command (perform method) and redirect to another view or do an arbitrairy redirect right away. This behaviour can be used to work with multi page forms coupled to one command or for instance implementing a generic 'system error' type of view.

Chapter 2. Using Baritus

This chapter will show you the basics you need to know to get started with Baritus.

2.1. Flow of execution

This sequence diagram displays the commons flow of execution with Baritus.



The minimal thing you have to do to use Baritus - besides using Maverick -, is to extend from `nl.openedge.baritus.FormBeanCtrl` and implement the two abstract methods `makeFormBean` and `perform`.

2.2. A worked example

Let's start from the begining and work our way up to a working example.

You should know how to work with Maverick. If you are not aquainted with Maverick yet, please read the Maverick tutorial [http://mav.sourceforge.net/maverick-manual.html] first. After you know how to work with Maverick, you should create a web application project in your favorite IDE, or you can download a stub project using Maverick, Baritus and Velocity here [http://sourceforge.net/project/showfiles.php?group_id=102886]. The stub project is has a Maven project descriptor in order to assist you with getting the dependencies you need fast. Read more about Apache Maven here [http://maven.apache.org].

We begin by creating a command in the Maverick configuration. A command in Maverick is the combination of a url-mapping (name), a controller class and one or more views.

```
<command name="example">
        <controller class="nl.openedge.examples.baritus.ExampleCtrl" />
```

```
        <view name="success" path="/example1.vm"/>
        <view name="error" path="/example1.vm"/>
</command>
```

Next, we create a controller class that extends from `nl.openedge.baritus.FormBeanCtrl` and a bean that we will use for population.

```

package nl.openedge.examples.baritus;

import nl.openedge.baritus.FormBeanContext;
import nl.openedge.baritus.FormBeanCtrl;
import org.infohazard.maverick.flow.ControllerContext;

public class ExampleCtrl extends FormBeanCtrl
{

        protected String perform(
                FormBeanContext formBeanContext,
                ControllerContext cctx)
                throws Exception
        {

                ExampleBean bean = (ExampleBean)formBeanContext.getBean();

                //... logic comes here...

                return "success";
        }

        protected Object makeFormBean(
                FormBeanContext formBeanContext,
                ControllerContext cctx)
        {
                // create a new instance of the bean we want to populate
                return new ExampleBean();
        }
}
```

```

package nl.openedge.examples.baritus;

public class ExampleBean
{
        private Integer myInteger;

        public Integer getMyInteger()
        {
                return myInteger;
        }

        public void setMyInteger(Integer integer)
        {
                myInteger = integer;
        }
}
```

And finally, we create a Velocity template. If you are new to Velocity, you can check out their website here [http://jakarta.apache.org/velocity]. Note that you can use whatever view type you like. We from OpenEdge mainly use Velocity, but you could use JSP's, XML/XSLT, webmacro [http://www.webmacro.org/], freemarker [http://freemarker.sourceforge.net/] etc. just the same.

Note that we use the default Maverick configuration, which saves the current instance of FormBeanContext as request attribute 'model'.

```

```

```
<HTML>
<head>
        <title>First Baritus VM</title>
</head>
<body>
        <form action="${request.contextPath}/example.m" method="POST">

                Please give an integer:
                <input type="text" name="myInteger" size="20"
                        value="$!{model.bean.myInteger}"/>

                <br><br>
                <input type="submit" value="save">

        </form>
</body>
</HTML>
```

Actually, the above code only works when population and validation succeeded. If it did not, the property myInteger was probably not set, and thus this example displays the old (null) value in case of an error. If population or validation errors occure, the user input is saved in a special map property 'overrideFields' in the FormBeanContext.

Now, instead of using override fields directely, we can use one of the utility methods that are available in FormBeanContext for displaying properties. In this case we use method 'displayProperty(String)', with the name of the form bean property to display as its parameter.

Method 'displayProperty' will not only lookup if an override value should be displayed, but also formats the output according to the current locale. For other formats, you can use 'displayProperty(propertyName, format)', but more about formatting later on. Here's our new version.

```

<HTML>
<head>
        <title>First Baritus VM</title>
</head>
<body>
        <form action="${request.contextPath}/example.m" method="POST">

                Please give an integer:
                <input type="text" name="myInteger" size="20"
                        value="$!{model.displayProperty('myInteger')}"/>

                <br><br>
                <input type="submit" value="save">

        </form>
</body>
</HTML>
```

To further fine tune our displaying we will create a Velocity macro [http://jakarta.apache.org/velocity/user-guide.html#Velocimacros].

```

#macro( forminput $fieldname $type $size )

<input type="${type}" name="${fieldname}" size="${size}"
        #if( $model.errors.get($fieldname) )
                class="fielderror"
        #else
                class="field"
        #end
        value="$!{model.displayProperty($fieldname)}">

#end
```

```
```

And a cascading stylesheet definition file (we'll call it style.css here).

```
.field {
        color: black;
        background-color: white;
}
.fielderror {
        color: white;
        background-color: red;
}
.error {
        color: red;
}
```

Now we change our Velcocity template, that now also includes the displaying of a list of error messages. Note that you probably create this only once, like in a footer (check out the Maverick transform options).

```
<HTML>
<head>
        <title>First Baritus VM</title>
        <link rel="stylesheet" type="text/css" href="${request.contextPath}/style.css">
</head>
<body>
        <form action="${request.contextPath}/example.m" method="POST">

                Please give an integer:
                #forminput( "myInteger" "text" 20 )

                <br><br>
                <input type="submit" value="save">

        </form>

        #foreach( $err in $model.errors )
                <span class="error">$!{err}</span><br>
        #end

</body>
</HTML>
```

That's all there is to it. You should experiment yourself now. Try adding properties of different types and try playing around with nested objects as well.

Be aware that nested objects that are null references will NOT get populated, so be sure to provide instances of all the objects that you want to have populated. The only execption to this rule is working with arrays.

2.3. Logging

In order to help you debug your web applications more easily, the following loggers are available. As a logging API Baritus uses Jakarta Commons Logging [http://jakarta.apache.org/commons/logging/], so you may use any of the logging mechanisms that are supported by Commons Logging.

The two loggers currently in use are named 'nl.openedge.baritus.population' and 'nl.openedge.baritus.formatting' and can be found as constants in Interface `nl.openedge.baritus.LogConstants`.

To configure with Log4J [http://logging.apache.org/log4j/docs/index.html], for instance, you can add the fol-

lowing lines to your Log4J configuration file:

```
log4j.logger.nl.openedge.baritus.population=DEBUG
log4j.logger.nl.openedge.baritus.formatting=DEBUG
```

Currently, only DEBUG gives you run time logging. And, as it is quite a lot, you probably do not want to turn it on in production systems.

2.4. ExecutionParams

ExecutionParams can used to tune the flow of execution. The execution params can be tuned just for the local request, or for all subsequent requests handled by the current controller.

If you want to tune just for the current request, you can get a working copy by calling 'getExecutionParams(cctx);', where cctx is the instance of the ControllerContext. If you pass null instead of cctx, you will allways get a new copy, otherwise you will get the save params for the current request. So, with chained controlls, the excecution params will be reused for the whole server side processing. Thus, if control A chains control B (e.g. A has view '/to-b.m'), and A changes the execution params, these changed params will be used when executing control B this request.

If you want to fix/ change the execution parameters for all subsequent requests, you have to call fixExecutionParams(params) in your control. For example:

```
public void init(Element controllerNode) throws ConfigException
{
  ExecutionParams params = getExecutionParams(null);
  params.setIncludeRequestAttributes(true);
  fixExecutionParams(params);
}
```

For additional fine grained control, you can decide to override method getExectionParams instead of using the copy and the fix method.

2.4.1. Available parameters

In this section the available execution parameters will be listed. The execution parameters are properties of class ExectionParams, and can be accessed with their getters and setters.

- `noCache` (default: true). If true, HTTP response headers that indicate that this page should not be cached will be set on each request.
- `setNullForEmptyString` (default: true). Should empty strings be interpreted as null references (true) or should the empty String be interpreted as a proper empty string. The default (true) is probably what you want in most cases, as HTML forms with empty fields send empty strings.
- `includeControllerParameters` (default: false). Indicates whether the configuration parameters of the controller shoudl be used for the population process.
- `includeSessionAttributes` (default: false). Indicates whether the session attributes of the current session should be used for the population process.
- `includeRequestAttributes` (default: false). Indicates whether the attributes of the current request should be used for the population process.
- `populateAndValidate` is used as an indicator whether population and validation should be done at all. This property is especially useful when chaining controllers (that is, controller A has controller B as its view, hence both A and B are executed in the same request). It is a property of FormBeanContext instead of the

ExecutionParams to provide easier access to non-controllers.

- `doFormValidationIfFieldValidationFailed` (default: true). Indicates whether the form validators should be executed when one of the field validators failed.
- `doPerformIfPopulationFailed` (default: false). Indicates whether the perform method of the control should be executed, even if population/ validation failed.
- `reuseFormBeanContext` (default: true). Indicates whether the form bean context should be reused for multiple invocations within the same request.
- `saveReqParamsAsOverrideFieldsOnError` (default: true). If population or validation fails and this property is true, all request parameters will be saved as override values. This will give you at least the full request the client sent, and guards you for the situation where properties that normally would be loaded in the command method are not set because of the population/ validation failure.
- `strictPopulationMode` (default: true). When an unexpected error occurs, should that be interpreted as a failure, or should it just be ignored. E.g. If you have request parameter not-a-valid-property, Ognl will throw exception 'ognl.InappropriateExpressionException' In strict mode, this will have the effect that the population process is marked as failed. If not in strict mode, the exception will be ignored.
- `trimStringInputValues` (default: true). Whether string input values should be trimmed before conversion. Note that the actual trimming depends on the used populator, so setting this parameter does not garantee trimming for all parameters.

2.5. FormBeanContext

In this section, we will look at the `FormBeanContext` in more detail.

The FormBeanContext is your interface between controllers and views and between controllers within the same request. Also, FormBeanContext has the utility methods that helps you to properly display values of the form bean. Furthermore, FormBeanContext keeps references to the form bean, the current (request scoped) locale and, in case errors occured during population/ validation, to error messages and the original input values. And lastely, FormBeanContext acts as a Map/ attribute decorator that you can use for request scoped atrributes you do not want to include in the population/ validation process.

2.5.1. References

- The form bean (property 'bean') is the object that you provided for population with method 'makeFormBean'.
- Property 'currentLocale' is the locale that will be used for this request. FormBeanCtrl gets the locale by calling it protected method 'getLocaleForRequest' and sets it as currentLocale in the FormBeanContext right after creating the formBeanContext.
- Property 'controller' provides a link to the currently active controller.

2.5.2. Attributes

- Map property 'errors' is used to store errors that occure during population and validation. You can use this map to store additional error messages as well.
- Map property 'overrideFields' is used to store the original input values when population or validation generated errors.
- Map property 'attributes' can be used to store additional attributes that should be available to the next controllers and views in the execution chain without it having any effect on population and validation, like end-user messages that are not errors. FormBeanContext acts as a wrapper for the attributes, so you should use the map methods like get(key), set(key, value) etc. This has the advantage of being able to compactely use these attributes in the view. E.g, say we have attribute with name 'myAttrib' stored as a FormBeanContxt attribute, in Velocity we could display this atribute like: '$model.myAttrib'.

2.5.3. Utility methods

The utility methods help you display properties. Allthough it is not nescesary to use these methods, as you could directly use the form bean, it is advisable to do so, as the utility methods help you look up override values and do formatting.

- `displayProperty(String propertyName)`. This method looks up the property with the provided propertyName from the form bean and returns the property value as a string using the type's default formatting. If an override value is found, this value will be returned. Baritus will try to format override values. If this fails (which is not unlikely as one of the reasons the override value was set in the first place is that conversion to the target type failed), the value will be converted to a string without using Converters.
- `displayProperty(String propertyName, String pattern)`. This method does the same as `displayProperty(String propertyName)`, but tries to use the provided pattern for formatting.
- `format(Object value)`. The provided object is formatted using the Formatter that was registered for its object type. If no formatter is found, just convert to a String using `ConvertUtils`.
- `format(Object value, String pattern)`. The provided object is formatted using the Formatter that was registered for its object type and using the provided pattern with that formatter. If no formatter is found, just convert to a String using `ConvertUtils`.
- `format(String formatterName, Object value)`. The provided object is formatted using the Formatter that was registered for the provided formatterName. If no formatter is found, just convert to a String using `ConvertUtils`.
- `format(String formatterName, Object value, pattern)`. The provided object is formatted using the Formatter that was registered for the provided formatterName and using the provided pattern with that formatter. If no formatter is found, just convert to a String using `ConvertUtils`.

With all formatting the locale for the current request will be taken into account. If localized formatters were registered, these will be used instead of non-localized formatters.

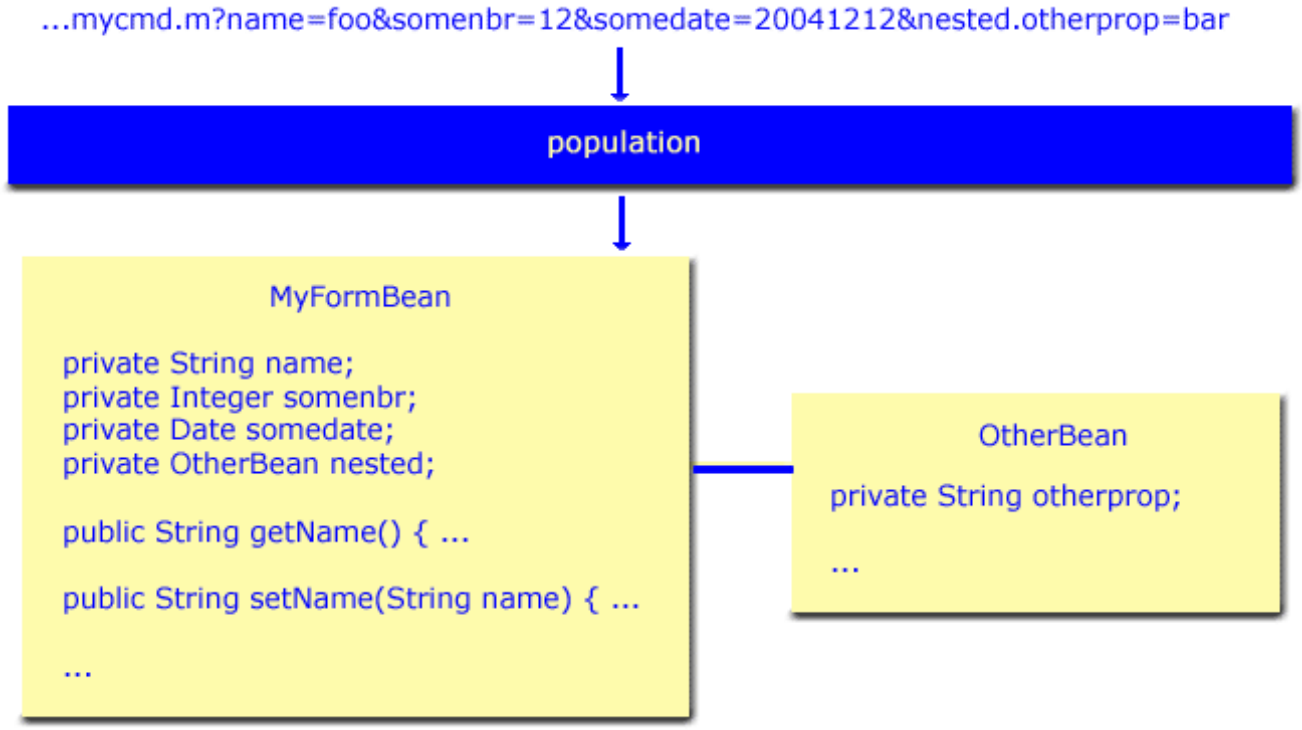
Chapter 3. Population

This chapter explains the population process.

3.1. Population process

When building web applications with Java, one of the commons things you want to do is to translate user input, i.e. request parameters, into server side objects, like objects of your domain model. Population is the process of matching request parameters (and optionally configuration parameters, session attributes and request attributes) with properties of the form bean, converting the request parameters to the target property type and setting the converted values on the form bean.

By default, Ognl [http://www.ognl.org] is used to get and set properties. It is possible to use a custom populator delegate instead of the default delegate 'nl.openedge.baritus.population.OgnlFieldPopulator'. See chapter six for the details, and this BeanUtils populator [BeanUtilsFieldPopulator.java.txt] as an example.



3.2. Conversion

One of the tasks to be done when populating a java bean from an HTTP request is the conversion of string parameters to Java types (Integers, Dates, etc.). `Converters` are used for this purpose. `Converters` are globally registered with the `ConverterRegistry`.

There are two types of converters, normal converters and locale-sensitive converters. Locale-sensitive converters override normal converters that are registered for the same type. Thus, if both a normal and a locale-sensitive converter are registered for type `java.util.Date`, the locale-sensitive converter will be used.

Allthough the ConverterRegistry has sensible defaults, you can register other Converters like this:

```
ConverterRegistry reg = ConverterRegistry.getInstance();
```

```
reg.register(new FallbackDateConverter(), Date.class);
reg.register(new FallbackDateConverter(), java.sql.Date.class);
reg.register(new FallbackDateConverter(), Timestamp.class);
```

Converters must implement either `org.apache.commons.beanutils.Converter` or `org.apache.commons.beanutils.locale.LocaleConverter`.

ConverterRegistry is a global registry. Hence, registered converters (and formatters) are shared amongst all controllers.

3.3. Formatting

Besides converting string values to Java types, you often want to format values. Two mechanisms are available for formatting.

- Using converters. Converters that implement interface `nl.openedge.baritus.converters.Formatter` are used automatically for formatting values of the type that the converter was registered for.
- Other formatters. Formatters (that implement interface `nl.openedge.baritus.converters.Formatter`) can registered on name. For example:

```
public class InverseFormatter implements Formatter
{
        /** reverse input */
        public String format(Object object, String pattern)
        {
                String formatted = null;
                if(object != null)
                {
                        StringBuffer b = new StringBuffer(String.valueOf(object));
                        formatted = b.reverse();
                }
                return formatted;
        }
}
```

```
...
ConverterRegistry reg = ConverterRegistry.getInstance();
reg.register(new InverseFormatter(), "*REVERSE");
...
```

```
#set( $myString = "toBeReversed" )
$!{model.format($myString, "*REVERSE")}
```

or, if the form bean has property 'myProperty':

```
$!{model.displayProperty("myProperty", "*REVERSE")}
```

Note that displayProperty not only looks up the property value, and checks if there is an override value registered for that property, but also tries to format the value using one of the registered formatters.

The algoritm to find a formatter is: first see if there's a formatter registered on fieldName (e.g. 'myProperty'). If not found, see if there's a formatter registered with the provided pattern (e.g. '*REVERSE). If not found, see if the converter that is registered for the property type and (the optional) locale, implements Formatter. If so, use the converter for formatting. If not found, just convert to a plain string (using ConvertUtils).

3.4. Custom population

The population process can be customized to suit your requirements. By default, class `nl.openedge.baritus.population.DefaultFieldPopulator` is used for population. `DefaultFieldPopulator` uses introspection to set the properties. You can however, create custom populators by creating implementations of `nl.openedge.baritus.population.FieldPopulator`. `nl.openedge.baritus.population.AbstractFieldPopulator` can be used as a base class. You can register custom populators in two ways:

- Register the populator with a field name like:

```
addPopulator("myProperty",  new MySpecialPopulator());
```

  E.g. with:

```
<form action="${request.contextType}/mycmd.m">
        <input type="text" name="myProperty" value="">
        <input type="text" name="someOtherProperty" value="">
        <input type="submit" value="submit">
</form>
```

  In the above example, `MySpecialPopulator` will be used to handle request parameter 'myProperty', and `DefaultFieldPopulator` will be used to handle 'someOtherProperty'.
- Register the populator with a regexp like:

```
// block property by regex pattern
addPopulator(Pattern.compile("(.)*ByRegex$"), new IgnoreFieldPopulator());
```

  E.g. with:

```
<form action="${request.contextType}/mycmd.m">
        <input type="text" name="myPropertyByRegex" value="">
        <input type="text" name="anotherPropertyByRegex" value="">
        <input type="text" name="someOtherProperty" value="">
        <input type="submit" value="submit">
</form>
```

  In this example, `IgnoreFieldPopulator` will be used to handle request parameters 'myPropertyByRegex' and 'anotherPropertyByRegex', and `DefaultFieldPopulator` will be used to handle 'someOtherProperty'. `IgnoreFieldPopulator` is actually shipped with Baritus as it has a common use. When working with pojo that come for instance from your domain model, you potentially introduce a security hazard. A common case is that clients should never be able to change id's directly from the request. Using `IgnoreFieldPopulator` you can, for instance, block all request parameters that end with '.id'.

Custom populators are registered with a controller. Hence, registering a custom populator for controller A does not have effect on the population for controller B.

Chapter 4. Validation

This chapter explains validation.

4.1. After population

The first pass in the population process is the conversion of request parameters etc. to Java objects and populating the provided form bean with those objects. If this is successful, you know that inputs for integers, dates, etc. conform to a valid form (e.g. 'foo' is not a valid integer, but '12' is).

Next, you might want to check if, for instance, a provided parameter not only is a valid integer, but is also greater than say 10. Or that a parameter not only is a valid date, but that this date also is after today. If we talk about validation, we mean this kind of checks.

Validation can be performed on two levels: field and form level. Field level validation is related to one input field (e.g. a request parameter) at a time. Form level validation is executed regardles the provided parameters.

Furthermore, whether a validation action is executed can be made dependend of a certain state (like the value a parameter) using `nl.openedge.baritus.validation.ValidationActivationRules`.

4.2. An example

Let us start with an example that displays some usages of the validation mechanism of Baritus. In the following example, we want to validate that not only myInteger is a valid integer, but also that the integer allways has a value that is greater than ten.

```
public class FooForm
{
        private Integer myInteger;

        public Integer getMyInteger() { return myInteger; }
        public void setMyInteger(Integer i) { myInteger = i; }
}
```

The validator:

```
public class BarValidator extends AbstractFieldValidator
{
        public boolean isValid(
                ControllerContext cctx,
                FormBeanContext formBeanContext,
                String fieldName,
                Object value)
        {
                // note that type conversion is allready done by the population process
                Integer val = (Integer)value;
                // the value can still be null though
                boolean valid = (val != null && val.intValue() > 10);

                if(!valid)
                {
                        String errorMessage = "input " + value +
                                " is not valid; please provide a number larger than 10";
                        formBeanContext.setError(fieldName, errorMessage);
                }
```

```
                return valid;
        }

}
```

Now, register it with your control:

```

public class MyCtrl extends FormBeanCtrl
{
        ...

        public void init(Element controllerNode) throws ConfigException
        {
                addValidator("myInteger", new BarValidator());
                ...
        }

        ...
}
```

And a crude example of how you can use it with a velocity template:

```

<input type="text" name="myInteger"
        #if( $model.errors.get('myInteger') ) class="fielderror" #else class="field" #end
        value="$!{model.displayProperty('myInteger')}">

<br>
#if( $model.errors.get('myInteger')  )
        <span class="error"> ${model.errors.get('myInteger') } </span>
#end
```

4.3. Mapped and indexed properties

When using mapped or indexed properties, you have two options of registering field validators, on target name and/ or on flat name. Let's illustratie this by example:

```

public class FormWithMap
{
        private Map foo = new HashMap();

        public Integer getfoo() { return foo; }
        public void setfoo(Map m) { foo = m; }
}
```

Not only can we register validators with the whole target name, including the keys or indexes like this:

```

...
        addValidator("foo['bar']", myValidator);
        addValidator("foo['someOtherKey']", myOtherValidator);
...
```

But in case the validator should be executed for all keys (or indexes), the 'flat name' (name without markup for map/ index navigation) as well like:

```

...
        addValidator("foo", myValidatorForAllKeys);
...
```

4.4. ValidationActivationRules

Whether validators are actually used a given situation can be directed through the use of ValidationActivationRules (`nl.openedge.baritus.validation.ValidationActivationRule`).

For example:

```
public class MyValidationRule implements ValidationActivationRule
{
        public boolean allowValidation(
                ControllerContext cctx,
                FormBeanContext formBeanContext)
        {
                return "true".equals(cctx.getRequest().getParameter("validate"));
        }
}
```

```
FieldValidator validator = new RequiredFieldValidator();
validator.setValidationRule(new MyValidationRule());
addValidator("myField", validator);
```

In the above example the validator will only be called when request parameter 'validate' is provided and has value 'true'.

You can register only one validation activation rule with a validator at a time. It is not difficult though, to stack validator activation rules, for example by using `nl.openedge.baritus.validation.impl.NestedValidationActivationRule`.

4.5. Escaping validation

By default, if population failed and/ or if you registered validators that failed, the 'peform' method of your controller will not be called. Instead, the errors and override fields are saved, and the error page is shown.

Earlier, you have seen that the actual firing of Validators can be made conditional by registering ValidatorActivationRules with your validators. This works well for higher level validation that is dependent of e.g. the value of one of the other request parameters.

There are two ways of escaping the default mechanism of not performing the controller command method. The first way is to skip population and validation all together. In order to achieve this, you can set property 'populateAndValidate' of the ExecutionParams to false. The second way is to set property 'doPerformIfPopulationFailed' to true (false by default). In this case, population and validation is performed as usually, but now the perform method is allways executed, regardless of the population/ validation outcome. This option should be used with care.

4.6. Validator implementations

Package `nl.openedge.baritus.validation.impl` is reserved for implementations of Validators.

- `MaximumFieldLengthValidator`. This validator checks on maximum length. If the type of the value is a

String, the string length is checked. If the type of the value is a Number, the actual number is used. E.g. if property maxLength is 4, "hello" will fail, but "hi" will pass, and number 5 will fail, but 2 will pass.

- `MinimumFieldLengthValidator`. This validator checks on minimum length. If the type of the value is a String, the string length is checked. If the type of the value is a Number, the actual number is used. E.g. if property maxLength is 4, "hello" will pass, but "hi" will fail, and number 5 will pass, but 2 will fail.
- `PropertyNotNullFormValidator`. Checks whether the form contains a non null property with the name of property propertyName.
- `RegexValidator`. Tests for a match against a regex pattern. if property 'mode' is MODE_VALID_IF_MATCHES (which is the default), isValid returns true if the input matches the pattern. If property mode is MODE_INVALID_IF_MATCHES (i.e. else), isValid returns false if the input matches the pattern.
- `RequiredFieldValidator`. Checks for a non-EMPTY input. Use this for fields that should have a not null (empty string) input. Note that as this validator is a field validator, and thus is registered for a single field, it is only fired if a field (e.g. a request parameter) is actually provided. In other words: if an instance of a required field validator was registered for field name 'myprop', but 'myprop' is not part of the request parameters, this validator never fires. Hence, if you want to be REALLY sure that a property of the form is not null, use a form validator (PropertyNotNullValidator). RequiredFieldValidator works fine for most cases where you have a HTML form with a field that should have a non empty value, but that - if a user fools around - does not seriously break anything when a value is not provided (e.g. you probably have not null constraint in you database as well).
- `RequiredSessionAttributeValidator`. Checks whether a session attribute exists with the key that was set for property sessionAttributeKey.

Chapter 5. Interceptors

The subject of this chapter, Interceptors, allow you to override/ extend the default behaviour of Baritus.

5.1. Interception

Interceptors provide a means to encapsulate cross-cutting code that is executed on pre-defined points in the line of execution. Interceptors are classes that implement one or more interfaces from the package 'nl.openedge.baritus.interceptors'.

Interceptors can be used to decorate the normal execution. Also, by throwing FlowExceptions, interceptors can alter the flow of execution. An interceptor can throw a FlowException if it wants Baritus to stop normal processing and go to the given declared view (using ReturnNowFlowException) such as 'error', or dispatch to an arbitrary - non declared - URL (using DispatchNowFlowException) location.

5.2. Interceptors

The following interceptors are available.

- `BeforeMakeFormBeanInterceptor`. Registered instances will have their command method executed before the method makeFormBean is called.
- `BeforePopulationInterceptor`. Registered instances will have their command method executed before population and validation is done.
- `PopulationErrorInterceptor`. Registered instances will have their command method executed if the model failed to populate, or did not pass validation.
- `AfterPopulationInterceptor`. Registered instances will have their command method executed before the normal action execution took place, but after the form bean was populated.
- `PerformExceptionInterceptor`. Registered instances will have their command method executed if an unhandeld exception occured while executing the perform method.
- `AfterPerformInterceptor`. Registered instances will have their command method executed after the normal action execution took place. That means that makeFormBean was called, the form was populated and - if that population was succesfull - the command method was called prior to this execution.

  You cannot be sure that the form was populated successfully. Therefore it's dangerous and generally bad practice to rely on form properties that are populated from the http request. A good usage example: a lot of views need data to fill their dropdown lists etc. In this method, you could load that data and save it in the form (or as a request attribute if that's your style). As this method is allways executed, you have a guaranteed data delivery to your view, regardless the normal execution outcome of the control.

5.3. Throwing FlowExceptions

An interceptor can alter the line of execution by thowing a FlowException. An example of when to do this is: if you expect a session variable to be set, but there is none, you might want to redirect to a search page or an error page. Another example: on certain non handled exceptions that are thrown in the perform method, you might want to redirect to a specific error page that is different from the error page that is shown when population errors occured.

There are two types of FlowExceptions that can be thrown by interceptors.

- `ReturnNowFlowException`. The property 'view' in this exception is used to return to Maverick as the view to display. In effect, this view must correspond to a declared view (like 'error' or 'detail') in your Maverick configuration.
- `DispatchNowFlowException`. The property 'dispatchPath' is used to dispatch to directly (like 'errors/critical.jsp', 'mymacro.vm' or 'http://www.myserver.com/support'.

Finally, with FlowException property 'executeOtherInterceptors' you can set on the exception that indicates whether (some of the) other interceptors should be executed. FlowExceptions thrown by these interceptors are ignored.

5.4. And example

Here is an example of how to use interceptors. In part of our imaginary application, we want to be sure that an object was saved in the session, and after we checked it is, we want to set that object as one of our form bean properties. If our object is not found however, we want to cancel the normal Baritus behaviour and do a redirect (or actually dispatch) to another location right away. Because we set that object in the form bean just before population takes place (but after the makeFormBean method was called), it can be populated by the request paremeters etc. as well.

First, there is our Interceptor.

```
public class LoadFooFromSessionInterceptor implements BeforePopulationInterceptor {

    /**
     * Before population takes place, we get our Foo object from the session, check
     * it's actually there (is not null) and set it in our form, so that it can be
     * populated as well.
     */
    public void doBeforePopulation(
        ControllerContext cctx, FormBeanContext formBeanContext)
        throws ServletException, DispatchNowFlowException, ReturnNowFlowException
    {
        HttpServletRequest request = cctx.getRequest();
        HttpSession session = request.getSession();
        Foo ourFoo = (Foo) session.getAttribute(Constants.SESSION_KEY_FOO_OBJECT);

        if (ourFoo == null)
        { // ourFoo does not exist in the session... redirect to 'home.m'
            throw new DispatchNowFlowException("home.m");
        }
        else
        { //  we're ok, set in form and exit method normally
            FooBarForm form = (FooBarForm) formBeanContext.getBean();
            form.setFoo(ourFoo);
        }
    }
}
```

Then, we should register the interceptor in the controllers we want to use it.

```
    public void init(Element controllerNode) throws ConfigException
    {
        addInterceptor(new LoadFooFromSessionInterceptor());
    }
```

Baritus will now call this interceptor each time this control is used right after calling makeFormBean and just before population and validation takes place.

5.5. Stacking interceptors

Just like you can do with validators, you can stack interceptors as well.

```
public void init(Element controllerNode) throws ConfigException
{
    addInterceptor(new FooInterceptor());
    addInterceptor(new BarInterceptor());
    addInterceptor(new MooInterceptor());
}
```

In the above example, all interceptors can be of the same type(s), and will be called in order of registration. Thus, FooInterceptor is called first, BarInterceptor after that and MooInterceptor is called last.

Chapter 6. Extending Baritus

This section gives some examples of how to use and extend Baritus.