Extending WildFly

Table of Contents

1 Target Audience

This document is intended for people who want to extend WildFly to introduce new capabilities.

1.1 Prerequisites

You should know how to download, install and run WildFly. If not please consult the Getting Started Guide. You should also be familiar with the management concepts from the Admin Guide, particularly the Core management concepts section and you need Java development experience to follow the example in this guide.

1.2 Examples in this guide

Most of the examples in this guide are being expressed as excerpts of the XML configuration files or by using a representation of the de-typed management model.

2 Overview

In this document we provide an example of how to extend the core functionality of WildFly via an extension and the subsystem it installs. The WildFly core is very simple and lightweight; most of the capabilities people associate with an application server are provided via extensions and their subsystems. The WildFly distribution includes many extensions and subsystems; the webserver integration is via a subsystem; the transaction manager integration is via a subsystem, the EJB container integration is via a subsystem, etc.

This document is divided into two main sections. The first is focused on learning by doing. This section will walk you through the steps needed to create your own subsystem, and will touch on most of the concepts discussed elsewhere in this guide. The second focuses on a conceptual overview of the key interfaces and classes described in the example. Readers should feel free to start with the second section if that better fits their learning style. Jumping back and forth between the sections is also a good strategy.

3 Example subsystem

Our example subsystem will keep track of all deployments of certain types containing a special marker file, and expose operations to see how long these deployments have been deployed.

3.1 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

3.2 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true.`
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

3.3 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The SubsystemExtension.initialize() method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

3.3.1 Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

It's parameter is an implementation of the ResourceDefinition interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by SubsystemDefinition.INSTANCE.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The ManagementResourceRegistration obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside registerOperations in your ResourceDefinition or by providing it in constructor of your SimpleResourceDefinition just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by `3`.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define DescriptionProvider, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Flowing example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes DefaultResourceAddDescriptionProvider and DefaultResourceRemoveDescriptionProvider that do work for you. In case you use SimpleResourceDefinition even that part is hidden from you.

resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);

For other operation handlers that are not add/remove you can use DefaultOperationDescriptionProvider that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "add-mime" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

⚠ When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

3.3.2 Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it TypeDefinition and for ease of use let it extend SimpleResourceDefinition instead just implement ResourceDefinition.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define SimpleAttributeDefinition named TICK, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation validateAndSet on AttributeDefinition that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

⚠ When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

3.4 Parsing and marshalling of the subsystem xml

JBoss AS 7 uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by `SUBSYSTEM_PATH` this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by `TYPE_PATH`) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

3.4.1 Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is add for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is add for the address `/subsystem=tracker`, and that 12345 was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command
`/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

3.5 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

3.5.1 Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

3.6 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly 8. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$AS7/domain/configuration/domain.xml`.

Now start up WildFly 8 by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly 8 console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

3.7 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

3.7.1 What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

3.7.2 How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type ModelType.EXPRESSION.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation ModelNode, and if it isn't already ModelType.EXPRESSION, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type ModelType.EXPRESSION. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

4 Working with WildFly Capabilities

An extension to WildFly will likely want to make use of services provided by the WildFly kernel, may want to make use of services provided by other subsystems, and may wish to make functionality available to other extensions. Each of these cases involves integration between different parts of the system. In releases prior to WildFly 10, this kind of integration was done on an ad-hoc basis, resulting in overly tight coupling between different parts of the system and overly weak integration contracts. For example, a service installed by subsystem A might depend on a service installed by subsystem B, and to record that dependency A's authors copy a ServiceName from B's code, or even refer to a constant or static method from B's code. The result is B's code cannot evolve without risking breaking A. And the authors of B may not even intend for other subsystems to use its services. There is no proper integration contract between the two subsystems.

Beginning with WildFly Core 2 and WildFly 10 the WildFly kernel's management layer provides a mechanism for allowing different parts of the system to integrate with each other in a loosely coupled manner. This is done via WildFly Capabilities. Use of capabilities provides the following benefits:

1. A standard way for system components to define integration contracts for their use by other system components.
2. A standard way for system components to access integration contracts provided by other system components.
3. A mechanism for configuration model referential integrity checking, such that if one component's configuration has an attribute that refers to an other component (e.g. a `socket-binding` attribute in a subsystem that opens a socket referring to that socket's configuration), the validity of that reference can be checked when validating the configuration model.

4.1 Capabilities

A capability is a piece of functionality used in a WildFly Core based process that is exposed via the WildFly Core management layer. Capabilities may depend on other capabilities, and this interaction between capabilities is mediated by the WildFly Core management layer.

Some capabilities are automatically part of a WildFly Core based process, but in most cases the configuration provided by the end user (i.e. in standalone.xml, domain.xml and host.xml) determines what capabilities are present at runtime. It is the responsibility of the handlers for management operations to register capabilities and to register any requirements those capabilities may have for the presence of other capabilities. This registration is done during the MODEL stage of operation execution

A capability has the following basic characteristics:

1. It has a name.
2. It may install an MSC service that can be depended upon by services installed by other capabilities. If it does, it provides a mechanism for discovering the name of that service.
3. It may expose some other API not based on service dependencies allowing other capabilities to integrate with it at runtime.
4. It may depend on, or **require** other capabilities.

During boot of the process, and thereafter whenever a management operation makes a change to the process' configuration, at the end of the MODEL stage of operation execution the kernel management layer will validate that all capabilities required by other capabilities are present, and will fail any management operation step that introduced an unresolvable requirement. This will be done before execution of the management operation proceeds to the RUNTIME stage, where interaction with the process' MSC Service Container is done. As a result, in the RUNTIME stage the handler for an operation can safely assume that the runtime services provided by a capability for which it has registered a requirement are available.

4.1.1 Comparison to other concepts

Capabilities vs modules

A JBoss Modules module is the means of making resources available to the classloading system of a WildFly Core based process. To make a capability available, you must package its resources in one or more modules and make them available to the classloading system. But a module is not a capability in and of itself, and simply copying a module to a WildFly installation does not mean a capability is available. Modules can include resources completely unrelated to management capabilities.

Capabilities vs Extensions

An extension is the means by which the WildFly Core management layer is made aware of manageable functionality that is not part of the WildFly Core kernel. The extension registers with the kernel new management resource types and handlers for operations on those resources. One of the things a handler can do is register or unregister a capability and its requirements. An extension may register a single capability, multiple capabilities, or possibly none at all. Further, not all capabilities are registered by extensions; the WildFly Core kernel itself may register a number of different capabilities.

4.1.2 Capability Names

Capability names are simple strings, with the dot character serving as a separator to allow namespacing.

The 'org.wildfly' namespace is reserved for projects associated with the WildFly organization on github (https://github.com/wildfly).

4.1.3 Statically vs Dynamically Named Capabilities

The full name of a capability is either statically known, or it may include a statically known base element and then a dynamic element. The dynamic part of the name is determined at runtime based on the address of the management resource that registers the capability. For example, the management resource at the address '/socket-binding-group=standard-sockets/socket-binding=web' will register a dynamically named capability named 'org.wildlfy.network.socket-binding.web'. The 'org.wildlfy.network.socket-binding' portion is the static part of the name.

All dynamically named capabilities that have the same static portion of their name should provide a consistent feature set and set of requirements.

4.1.4 Service provided by a capability

Typically a capability functions by registering a service with the WildFly process' MSC ServiceContainer, and then dependent capabilities depend on that service. The WildFly Core management layer orchestrates registration of those services and service dependencies by providing a means to discover service names.

4.1.5 Custom integration APIs provided by a capability

Instead of or in addition to providing MSC services, a capability may expose some other API to dependent capabilities. This API must be encapsulated in a single class (although that class can use other non-JRE classes as method parameters or return types).

4.1.6 Capability Requirements

A capability may rely on other capabilities in order to provide its functionality at runtime. The management operation handlers that register capabilities are also required to register their requirements.

There are three basic types of requirements a capability may have:

- Hard requirements. The required capability must always be present for the dependent capability to function.
- Optional requirements. Some aspect of the configuration of the dependent capability controls whether the depended on capability is actually necessary. So the requirement cannot be known until the running configuration is analyzed.
- Runtime-only requirements. The dependent capability will check for the presence of the depended upon capability at runtime, and if present it will utilize it, but if it is not present it will function properly without the capability. There is nothing in the dependent capability's configuration that controls whether the depended on capability must be present. Only capabilities that declare themselves as being suitable for use as a runtime-only requirement should be depended upon in this manner.

Hard and optional requirements may be for either statically named or dynamically named capabilities. Runtime-only requirements can only be for statically named capabilities, as such a requirement cannot be specified via configuration, and without configuration the dynamic part of the required capability name is unknown.

Supporting runtime-only requirements

Not all capabilities are usable as a runtime-only requirement.

Any dynamically named capability is not usable as a runtime-only requirement.

For a capability to support use as a runtime-only requirement, it must guarantee that a configuration change to a running process that removes the capability will not impact currently running capabilities that have a runtime-only requirement for it. This means:

- A capability that supports runtime-only usage must ensure that it never removes its runtime service except via a full process reload.
- A capability that exposes a custom integration API generally is not usable as a runtime-only requirement. If such a capability does support use as a runtime-only requirement, it must ensure that any functionality provided via its integration API remains available as long as a full process reload has not occurred.

4.2 Capability Contract

A capability provides a stable contract to users of the capability. The contract includes the following:

- The name of the capability (including whether it is dynamically named).
- Whether it installs an MSC Service, and if it does, the value type of the service. That value type then becomes a stable API users of the capability can rely upon.
- Whether it provides a custom integration API, and if it does, the type that represents that API. That type then becomes a stable API users of the capability can rely upon.
- Whether the capability supports use as a runtime-only requirement.

Developers can learn about available capabilities and the contracts they provide by reading the WildFly *capabilty registry*.

4.3 Capability Registry

The WildFly organization on github maintains a git repo where information about available capabilities is published.

https://github.com/wildfly/wildfly-capabilities

Developers can learn about available capabilities and the contracts they provide by reading the WildFly capabilty registry.

The README.md file at the root of that repo explains the how to find out information about the registry.

Developers of new capabilities are **strongly encouraged** to document and register their capability by submitting a pull request to the wildfly-capabilities github repo. This both allows others to learn about your capability and helps prevent capability name collisions. Capabilities that are used in the WildFly or WildFly Core code base itself **must** have a registry entry before the code referencing them will be merged.

External organizations that create capabilities should include an organization-specific namespace as part their capability names to avoid name collisions.

4.4 Using Capabilities

Now that all the background information is presented, here are some specifics about how to use WildFly capabilities in your code.

4.4.1 Basics of Using Your Own Capability

Creating your capability

A capability is an instance of the immutable `org.jboss.as.controller.capability.RuntimeCapability` class. A capability is usually registered by a resource, so the usual way to use one is to store it in constant in the resource's `ResourceDefinition`. Use a `RuntimeCapability.Builder` to create one.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    static final RuntimeCapability<Void> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo").build();


    . . .
}
```

That creates a statically named capability named `com.example.foo`.

If the capability is dynamically named, add the `dynamic` parameter to state this:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", true).build();
```

Most capabilities install a service that requiring capabilities can depend on. If your capability does this, you need to declare the service's *value type* (the type of the object returned by `org.jboss.msc.Service.getValue()`). For example, if FOO_CAPABILITY provides a `Service<javax.sql.DataSource>`:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", DataSource.class).build();
```

For a dynamic capability:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
           RuntimeCapability.Builder.of("com.example.foo", true, DataSource.class).build();
```

If the capability provides a custom integration API, you need to instantiate an instance of that API:

```
public class JTSCapability {

    static final JTSCapability INSTANCE = new JTSCapability();

    private JTSCapability() {}

    /**
     * Gets the names of the {@link org.omg.PortableInterceptor.ORBInitializer} implementations
that should be included
     * as part of the {@link org.omg.CORBA.ORB#init(String[], java.util.Properties)
initialization of an ORB}.
     *
     * @return the names of the classes implementing {@code ORBInitializer}. Will not be {@code
null}.
     */
    public List<String> getORBInitializerClasses() {
        return Collections.unmodifiableList(Arrays.asList(

"com.arjuna.ats.jts.orbspecific.jacorb.interceptors.interposition.InterpositionORBInitializerImpl"
"com.arjuna.ats.jbossatx.jts.InboundTransactionCurrentInitializer"));
    }
}
```

and provide it to the builder:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE).build();
```

For a dynamic capability:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo", true, JTSCapability.INSTANCE).build();
```

A capability can provide both a custom integration API and install a service:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE)
                .setServiceType(DataSource.class)
                .build();
```

Registering and unregistering your capability

Once you have your capability, you need to ensure it gets registered with the WildFly Core kernel when your resource is added. This is easily done simply by providing a reference to the capability to the resource's `ResourceDefinition`. This assumes your add handler is a subclass of the standard `org.jboss.as.controller.SimpleResourceDefinition`. `SimpleResourceDefinition` provides a `Parameters` class that provides a builder-style API for setting up all the data needed by your definition. This includes a `setCapabilities` method that can be used to declare the capabilities provided by resources of this type.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    . . .

    MyResourceDefinition() {
        super(new SimpleResourceDefinition.Parameters(PATH, RESOLVER)
            .setAddHandler(MyAddHandler.INSTANCE)
            .setRemoveHandler(MyRemoveHandler.INSTANCE)
            .setCapabilities(FOO_CAPABILITY)
            );
    }
}
```

Your add handler needs to extend the standard `org.jboss.as.controller.AbstractAddStepHandler` class or one of its subclasses:

```
class MyAddHandler extends AbstractAddStepHandler() {
```

`AbstractAddStepHandler`'s logic will register the capability when it executes.

Your remove handler must also extend of the standard `org.jboss.as.controller.AbstractRemoveStepHandler` or one of its subclasses.

```
class MyRemoveHandler extends AbstractRemoveStepHandler() {
```

`AbstractRemoveStepHandler`'s logic will deregister the capability when it executes.

If for some reason you cannot base your `ResourceDefinition` on `SimpleResourceDefinition` or your handlers on `AbstractAddStepHandler` and `AbstractRemoveStepHandler` then you will need to take responsibility for registering the capability yourself. This is not expected to be a common situation. See the implementation of those classes to see how to do it.

Installing, accessing and removing the service provided by your capability

If your capability installs a service, you should use the `RuntimeCapability` when you need to determine the service's name. For example in the `Stage.RUNTIME` handling of your "add" step handler. Here's an example for a statically named capability:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

If the capability is dynamically named, get the dynamic part of the name from the `OperationContext` and use that when getting the service name:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        String myName = context.getCurrentAddressValue();
        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName(myName);
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

The same patterns should be used when accessing or removing the service in handlers for `remove`, `write-attribute` and custom operations.

If you use `ServiceRemoveStepHandler` for the `remove` operation, simply provide your `RuntimeCapability` to the `ServiceRemoveStepHandler` constructor and it will automatically remove your capability's service when it executes.

4.4.2 Basics of Using Other Capabilities

When a capability needs another capability, it only refers to it by its string name. A capability should not reference the `RuntimeCapability` object of another capability.

Before a capability can look up the service name for a required capability's service, or access its custom integration API, it must first register a requirement for the capability. This must be done in Stage.MODEL, while service name lookups and accessing the custom integration API is done in Stage.RUNTIME.

Registering a requirement for a capability is simple.

Registering a hard requirement for a static capability

If your capability has a hard requirement for a statically named capability, simply declare that to the builder for your `RuntimeCapability`. For example, WildFly's JTS capability requires both a basic transaction support capability and IIOP capabilities:

```
static final RuntimeCapability<JTSCapability> JTS_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.transactions.jts", new JTSCapability())
                .addRequirements("org.wildfly.transactions", "org.wildfly.iiop.orb",
"org.wildfly.iiop.corba-naming")
                .build();
```

When your capability is registered with the system, the WildFly Core kernel will automatically register any static hard requirements declared this way.

Registering a requirement for a dynamically named capability

If the capability you require is dynamically named, usually your capability's resource will include an attribute whose value is the dynamic part of the required capability's name. You should declare this fact in the `AttributeDefinition` for the attribute using the `SimpleAttributeDefinitionBuilder.setCapabilityReference` method.

For example, the WildFly "remoting" subsystem's "org.wildfly.remoting.connector" capability has a requirement for a dynamically named socket-binding capability:

```
public class ConnectorResource extends SimpleResourceDefinition {

    . . .

    static final String SOCKET_CAPABILITY_NAME = "org.wildfly.network.socket-binding";
    static final RuntimeCapability<Void> CONNECTOR_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.remoting.connector", true)
                    .build();

    . . .

    static final SimpleAttributeDefinition SOCKET_BINDING =
            new SimpleAttributeDefinitionBuilder(CommonAttributes.SOCKET_BINDING,
ModelType.STRING, false)

.addAccessConstraint(SensitiveTargetAccessConstraintDefinition.SOCKET_BINDING_REF)
                .setCapabilityReference(SOCKET_CAPABILITY_NAME, CONNECTOR_CAPABILITY)
                .build();
```

If the "add" operation handler for your resource extends `AbstractAddStepHandler` and the handler for `write-attribute` extends `AbstractWriteAttributeHandler`, the declaration above is sufficient to ensure that the appropriate capability requirement will be registered when the attribute is modified.

Depending upon a service provided by another capability

Once the requirement for the capability is registered, your `OperationStepHandler}}s can use the {{OperationContext` to discover the name of the service provided by the required capability.

For example, the "add" handler for a remoting connector uses the `OperationContext` to find the name of the needed {{SocketBinding} service:

```
final String socketName = ConnectorResource.SOCKET_BINDING.resolveModelAttribute(context,
fullModel).asString();
        final ServiceName socketBindingName =
context.getCapabilityServiceName(ConnectorResource.SOCKET_CAPABILITY_NAME, socketName,
SocketBinding.class);
```

That service name is then used to add a dependency on the `SocketBinding` service to the remoting connector service.

If the required capability isn't dynamically named, `OperationContext` exposes an overloaded `getCapabilityServiceName` variant. For example, if a capability requires a remoting Endpoint:

```
ServiceName endpointService = context.getCapabilityServiceName("org.wildfly.remoting.endpoint",
Endpoint.class);
```

Using a custom integration API provided by another capability

In your `Stage.RUNTIME` handler, use `OperationContext.getCapabilityRuntimeAPI` to get a reference to the required capability's custom integration API. Then use it as necessary.

```
List<String> orbInitializers = new ArrayList<String>();
        . . .
        JTSCapability jtsCapability =
context.getCapabilityRuntimeAPI(IIOPExtension.JTS_CAPABILITY, JTSCapability.class);
        orbInitializers.addAll(jtsCapability.getORBInitializerClasses());
```

Runtime-only requirements

If your capability has a runtime-only requirement for another capability, that means that if that capability is present in `Stage.RUNTIME` you'll use it, and if not you won't. There is nothing about the configuration of your capability that triggers the need for the other capability; you'll just use it if it's there.

In this case, use `OperationContext.hasOptionalCapability` in your `Stage.RUNTIME` handler to check if the capability is present:

```
protected void performRuntime(final OperationContext context, final ModelNode operation, final
ModelNode model) throws OperationFailedException {

        ServiceName myServiceName = MyResource.FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> myService = createService(context, model);
        ServiceBuilder<DataSource> builder = context.getTarget().addService(myServiceName,
myService);

        // Inject a "Bar" into our "Foo" if bar capability is present
        if (context.hasOptionalCapability("com.example.bar",
MyResource.FOO_CAPABILITY.getName(), null) {
            ServiceName barServiceName = context.getCapabilityServiceName("com.example.bar",
Bar.class);
            builder.addDependency(barServiceName, Bar.class, myService.getBarInjector());
        }

        builder.install();
    }
```

The WildFly Core kernel will not register a requirement for the "com.example.bar" capability, so if a configuration change occurs that means that capability will no longer be present, that change will not be rolled back. Because of this, runtime-only requirements can only be used with capabilities that declare in their contract that they support such use.

Using a capability in a DeploymentUnitProcessor

{{DeploymentUnitProcessor}}s are likely to have a need to interact with capabilities, in order to create service dependencies from a deployment service to a capability provided service or to access some aspect of a capability's custom integration API that relates to deployments.

If a `DeploymentUnitProcessor` associated with a capability implementation needs to utilize its own capability object, the `DeploymentUnitProcessor` authors should simply provide it with a reference to the `RuntimeCapability` instance. Service name lookups or access to the capabilities custom integration API can then be performed by invoking the methods on the `RuntimeCapability`.

If you need to access service names or a custom integration API associated with a different capability, you will need to use the `org.jboss.as.controller.capability.CapabilityServiceSupport` object associated with the deployment unit. This can be found as an attachment to the `DeploymentPhaseContext`:

```
class MyDUP implements DeploymentUntiProcessor {

    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

        AttachmentKey<CapabilityServiceSupport> key =
org.jboss.as.server.deployment.Attachments.DEPLOYMENT_COMPLETE_SERVICES;
        CapabilityServiceSupport capSvcSupport = phaseContext.getAttachment(key);
```

Once you have the `CapabilityServiceSupport` you can use it to look up service names:

```
ServiceName barSvcName = capSvcSupport.getCapabilityServiceName("com.example.bar");
        // Determine what 'baz' the user specified in the deployment descriptor
        String bazDynamicName = getSelectedBaz(phaseContext);
        ServiceName bazSvcName = capSvcSupport.getCapabilityServiceName("com.example.baz",
bazDynamicName);
```

It's important to note that when you request a service name associated with a capability, the `CapabilityServiceSupport` will give you one regardless of whether the capability is actually registered with the kernel. If the capability isn't present, any service dependency your DUP creates using that service name will eventually result in a service start failure, due to the missing dependency. This behavior of not failing immediately when the capability service name is requested is deliberate. It allows deployment operations that use the `rollback-on-runtime-failure=false` header to successfully install (but not start) all of the services related to a deployment. If a subsequent operation adds the missing capability, the missing service dependency problem will then be resolved and the MSC service container will automatically start the deployment services.

You can also use the `CapabilityServiceSupport` to obtain a reference to the capability's custom integration API:

```
// We need custom integration with the baz capability beyond service injection
        BazIntegrator bazIntegrator;
        try {
            bazIntegrator = capSvcSupport.getCapabilityRuntimeAPI("com.example.baz",
bazDynamicName, BazIntegrator.class);
        } catch (NoSuchCapabilityException e) {
            //
            String msg = String.format("Deployment %s requires use of the 'bar' capability but
it is not currently registered",
                                       phaseContext.getDeploymentUnit().getName());
            throw new DeploymentUnitProcessingException(msg);
        }
```

Note that here, unlike the case with service name lookups, the `CapabilityServiceSupport` will throw a checked exception if the desired capability is not installed. This is because the kernel has no way to satisfy the request for a custom integration API if the capability is not installed. The `DeploymentUnitProcessor` will need to catch and handle the exception.

4.4.3 Detailed API

The WildFly Core kernel's API for using capabilities is covered in detail in the javadoc for the RuntimeCapability and RuntimeCapability.Builder classes and the OperationContext and CapabilityServiceSupport interfaces.

Many of the methods in `OperationContext` related to capabilities have to do with registering capabilities or registering requirements for capabilities. Typically non-kernel developers won't need to worry about these, as the abstract `OperationStepHandler` implementations provided by the kernel take care of this for you, as described in the preceding sections. If you do find yourself in a situation where you need to use these in an extension, please read the javadoc thoroughly.

5 Key Interfaces and Classes Relevant to Extension Developers

In the first major section of this guide, we provided an example of how to implement an extension to the AS. The emphasis there was learning by doing. In this section, we'll focus a bit more on the major WildFly interfaces and classes that most are relevant to extension developers. The best way to learn about these interfaces and classes in detail is to look at their javadoc. What we'll try to do here is provide a brief introduction of the key items and how they relate to each other.

Before digging into this section, readers are encouraged to read the "Core Management Concepts" section of the Admin Guide.

5.1 Extension Interface

The `org.jboss.as.controller.Extension` interface is the hook by which your extension to the core AS is able to integrate with the AS. During boot of the AS, when the `<extension>` element in the AS's xml configuration file naming your extension is parsed, the JBoss Modules module named in the element's name attribute is loaded. The standard JDK `java.lang.ServiceLoader` mechanism is then used to load your module's implementation of this interface.

The function of an `Extension` implementation is to register with the core AS the management API, xml parsers and xml marshallers associated with the extension module's subsystems. An `Extension` can register multiple subsystems, although the usual practice is to register just one per extension.

Once the `Extension` is loaded, the core AS will make two invocations upon it:

- `void initializeParsers(ExtensionParsingContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to initialize the XML parsers for this extension's subsystems and register them with the given `ExtensionParsingContext`. The parser's job when it is later called is to create `org.jboss.dmr.ModelNode` objects representing WildFly management API operations needed make the AS's running configuration match what is described in the xml. Those management operation {{ModelNode}}s are added to a list passed in to the parser.

A parser for each version of the xml schema used by a subsystem should be registered. A well behaved subsystem should be able to parse any version of its schema that it has ever published in a final release.

- `void initialize(ExtensionContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to register with the core AS the management API for its subsystems, and to register the object that is capable of marshalling the subsystem's in-memory configuration back to XML. Only one XML marshaller is registered per subsystem, even though multiple XML parsers can be registered. The subsystem should always write documents that conform to the latest version of its XML schema.

The registration of a subsystem's management API is done via the `ManagementResourceRegistration` interface. Before discussing that interface in detail, let's describe how it (and the related `Resource` interface) relate to the notion of managed resources in the AS.

5.2 WildFly Managed Resources

Each subsystem is responsible for managing one or more management resources. The conceptual characteristics of a management resource are covered in some detail in the Admin Guide; here we'll just summarize the main points. A management resource has

- An ***address*** consisting of a list of key/value pairs that uniquely identifies a resource
- Zero or more ***attributes***, the value of which is some sort of `org.jboss.dmr.ModelNode`
- Zero or more supported ***operations***. An operation has a string name and zero or more parameters, each of which is a key/value pair where the key is a string naming the parameter and the value is some sort of `ModelNode`
- Zero or ***children***, each of which in turn is a managed resource

The implementation of a managed resource is somewhat analogous to the implementation of a Java object. A managed resource will have a "type", which encapsulates API information about that resource and logic used to implement that API. And then there are actual instances of the resource, which primarily store data representing the current state of a particular resource. This is somewhat analogous to the "class" and "object" notions in Java.

A managed resource's type is encapsulated by the `org.jboss.as.controller.registry.ManagementResourceRegistration` the core AS creates when the type is registered. The data for a particular instance is encapsulated in an implementation of the `org.jboss.as.controller.registry.Resource` interface.

5.3 ManagementResourceRegistration Interface

TODO

5.4 ResourceDefinition Interface

TODO

Most commonly used implementation: `SimpleResourceDefinition`

5.4.1 ResourceDescriptionResolver

TODO

Most commonly used implementation: `StandardResourceDescriptionResolver`

5.5 AttributeDefinition Interface

TODO

Most commmonly used implementation: `SimpleAttributeDefinition`. Use `SimpleAttributeDefinitionBuilder` to build.

5.6 OperationDefinition and OperationStepHandler Interfaces

TODO

5.7 Operation Execution and the OperationContext

TODO

5.8 Resource Interface

TODO

5.9 DeploymentUnitProcessor Interface

TODO

5.10 Useful classes for implementing OperationStepHandler

TODO

6 CLI Extensibility for Layered Products

In addition to supporting the ServiceLoader extension mechanism to load command handlers coming from outside of the CLI codebase, starting from wildfly-core-1.0.0.Beta1 release the CLI running in a modular classloading environment can be extended with commands exposed in server extension modules. The CLI will look for and register extension commands when it (re-)connects to the controller by iterating through the registered by that time extensions and using the ServiceLoader mechanism on the extension modules. (Note, that this mechanism will work only for extensions available in the server installation the CLI is launched from.)

Here is an example of a simple command handler and its integration.

```
package org.jboss.as.test.cli.extensions;public class ExtCommandHandler extends
org.jboss.as.cli.handlers.CommandHandlerWithHelp {

package org.jboss.as.test.cli.extensions;
public class ExtCommandHandler extends org.jboss.as.cli.handlers.CommandHandlerWithHelp {


    public static final String NAME = "ext-command";
    public static final String OUTPUT = "hello world!";


    public CliExtCommandHandler() {
        super(NAME, false);
    }


    @Override
    protected void doHandle(CommandContext ctx) throws CommandLineException {
        ctx.printLine(OUTPUT);
    }
}
```

The command will simply print a message to the terminal. The next step is to implement the CLI CommandHandlerProvider interface.

```
package org.jboss.as.test.cli.extensions;
public class ExtCommandHandlerProvider implements org.jboss.as.cli.CommandHandlerProvider {


    @Override
    public CommandHandler createCommandHandler(CommandContext ctx) {
        return new ExtCommandHandler();
    }


    /**
     * Whether the command should be available in tab-completion.
     */
    @Override
    public boolean isTabComplete() {
        return true;
    }


    /**
     * Command name(s).
     */
    @Override
    public String[] getNames() {
        return new String[]{ExtCommandHandler.NAME};
    }
}
```

The final step is to include **META-INF/services/org.jboss.as.cli.CommandHandlerProvider** entry into the JAR file containing the classes above with value **org.jboss.as.test.cli.extensions.ExtCommandHandlerProvider**.

7 All WildFly documentation

Documentation

8 CLI extensibility for layered products

In addition to supporting the ServiceLoader extension mechanism to load command handlers coming from outside of the CLI codebase, starting from wildfly-core-1.0.0.Beta1 release the CLI running in a modular classloading environment can be extended with commands exposed in server extension modules. The CLI will look for and register extension commands when it (re-)connects to the controller by iterating through the registered by that time extensions and using the ServiceLoader mechanism on the extension modules. (Note, that this mechanism will work only for extensions available in the server installation the CLI is launched from.)

Here is an example of a simple command handler and its integration.

```
package org.jboss.as.test.cli.extensions;public class ExtCommandHandler extends
org.jboss.as.cli.handlers.CommandHandlerWithHelp {

package org.jboss.as.test.cli.extensions;
public class ExtCommandHandler extends org.jboss.as.cli.handlers.CommandHandlerWithHelp {


    public static final String NAME = "ext-command";
    public static final String OUTPUT = "hello world!";


    public CliExtCommandHandler() {
        super(NAME, false);
    }


    @Override
    protected void doHandle(CommandContext ctx) throws CommandLineException {
        ctx.printLine(OUTPUT);
    }
}
```

The command will simply print a message to the terminal. The next step is to implement the CLI CommandHandlerProvider interface.

```
package org.jboss.as.test.cli.extensions;
public class ExtCommandHandlerProvider implements org.jboss.as.cli.CommandHandlerProvider {


    @Override
    public CommandHandler createCommandHandler(CommandContext ctx) {
        return new ExtCommandHandler();
    }


    /**
     * Whether the command should be available in tab-completion.
     */
    @Override
    public boolean isTabComplete() {
        return true;
    }


    /**
     * Command name(s).
     */
    @Override
    public String[] getNames() {
        return new String[]{ExtCommandHandler.NAME};
    }
}
```

The final step is to include **META-INF/services/org.jboss.as.cli.CommandHandlerProvider** entry into the JAR file containing the classes above with value **org.jboss.as.test.cli.extensions.ExtCommandHandlerProvider**.

9 Domain Mode Subsystem Transformers

9.1 "Abstract"

A WildFly/JBoss AS 7 domain may consist of a new Domain Controller (DC) controlling slave Host Controllers (HC) running older versions. Each slave HC maintains a copy of the centralized domain configuration, which they use for controlling their own servers. In order for the slave HCs to understand the configuration from the DC, transformation is needed, whereby the DC translates the configuration and operations into something the slave HCs can understand.

9.2 Background

WildFly comes with a domain mode which allows you to have one Host Controller acting as the Domain Controller. The Domain Controller's job is to maintain the centralized domain configuration. Another term for the DC is 'Master Host Controller'. Before explaining why transformers are important and when they should be used, we will revisit how the domain configuration is used in domain mode.

The centralized domain configuration is stored in `domain.xml`. This is only ever parsed on the DC, and it has the following structure:

- `extensions` - contains:
  - `extension` - a references to a module that bootstraps the `org.jboss.as.controller.Extension` implementation used to bootstrap your subsystem parsers and initialize the resource definitions for your subsystems.
- `profiles` - contains:
  - `profile` - a named set of:
    - `subsystem` - contains the configuration for a subsystem, using the parser initialized by the subsystem's extension.
- `socket-binding-groups` - contains:
  - `socket-binding-group` - a named set of:
    - `socket-binding` - A named port on an interface which can be referenced from the `subsystem` configurations for subsystems opening sockets.
- `server-groups` - contains
  - `server-group` - this has a name and references a `profile` and a `socket-binding-group`. The HCs then reference the `server-group` name from their `<servers>` section in `host.xml`.

When the DC parses `domain.xml`, it is transformed into `add` (and in some cases `write-attribute`) operations just as explained in Parsing and marshalling of the subsystem xml. These operations build up the model on the DC.

A HC wishing to join the domain and use the DC's centralized configuration is known as a 'slave HC'. A slave HC maintains a copy of the DC's centralized domain configuration. This copy of the domain configuration is used to start its servers. This is done by asking the domain model to `describe` itself, which in turn asks the subsystems to `describe` themselves. The `describe` operation for a subsystem looks at the state of the subsystem model and produces the `add` operations necessary to create the subsystem on the server. The same mechanism also takes place on the DC (bear in mind that the DC is also a HC, which can have its own servers), although of course its copy of the domain configuration is the centralized one.

There are two steps involved in keeping the keeping the slave HC's domain configuration in sync with the centralized domain configuration.

- getting the initial domain model
- an operation changes something in the domain configuration

Let's look a bit closer at what happens in each of these steps.

9.2.1 Getting the initial domain model

When a slave HC connects to the DC it obtains a copy of the domain model from the DC. This is done in a simpler serialized format, different from the operations that built up the model on the DC, or the operations resulting from the `describe` step used to bootstrap the servers. They describe each address that exists in the DC's model, and contain the attributes set for the resource at that address. This serialized form looks like this:

```
[{
    "domain-resource-address" => [],
    "domain-resource-model" => {
        "management-major-version" => 2,
        "management-minor-version" => 0,
        "management-micro-version" => 0,
        "release-version" => "8.0.0.Beta1-SNAPSHOT",
        "release-codename" => "WildFly"
    }
},
{
    "domain-resource-address" => [("extension" => "org.jboss.as.clustering.infinispan")],
    "domain-resource-model" => {"module" => "org.jboss.as.clustering.infinispan"}
},
--SNIP - the rest of the extensions --
{
    "domain-resource-address" => [("extension" => "org.jboss.as.weld")],
    "domain-resource-model" => {"module" => "org.jboss.as.weld"}
},
{
    "domain-resource-address" => [("system-property" => "java.net.preferIPv4Stack")],
    "domain-resource-model" => {
        "value" => "true",
        "boot-time" => undefined
    }
},
{
    "domain-resource-address" => [("profile" => "full-ha")],
    "domain-resource-model" => undefined
},
{
    "domain-resource-address" => [
        ("profile" => "full-ha"),
        ("subsystem" => "logging")
    ],
    "domain-resource-model" => {}
},
{
    "domain-resource-address" => [sss|WFLY8:Example subsystem],
    "domain-resource-model" => {
        "level" => "INFO",
        "enabled" => undefined,
        "encoding" => undefined,
        "formatter" => "%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n",
        "filter-spec" => undefined,
        "autoflush" => undefined,
        "target" => undefined,
        "named-formatter" => undefined
    }
},
--SNIP---
```

The slave HC then applies these one at a time and builds up the initial domain model. It needs to do this before it can start any of its servers.

9.2.2 An operation changes something in the domain configuration

Once a domain is up and running we can still change things in the domain configuration. These changes must happen when connected to the DC, and are then propagated to the slave HCs, which then in turn propagate the changes to any servers running in a server group affected by the changes made. In this example:

```
[disconnected /] connect
[domain@localhost:9990 /]
/profile=full/subsystem=datasources/data-source=ExampleDS:write-attribute(name=enabled,value=false
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {
        "slave" => {"server-one" => {"response" => {
            "outcome" => "success",
            "result" => undefined,
            "response-headers" => {
                "operation-requires-restart" => true,
                "process-state" => "restart-required"
            }
        }}},
        "master" => {
            "server-one" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }},
            "server-two" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }}
        }
    }}}
}
```

the DC propagates the changes to itself `host=master`, which in turn propagates it to its two servers belonging to `main-server-group` which uses the `full` profile. More interestingly, it also propagates it to `host=slave` which updates its local copy of the domain model, and then propagates the change to its `server-one` which belongs to `main-server-group` which uses the `full` profile.

9.3 Versions and backward compatibility

A HC and its servers will always be the same version of WildFly (they use the same module path and jars). However, the DC and the slave HCs do not necessarily need to be the same version. One of the points in the original specification for WildFly is that

Important

A Doman Controller should be able to manage slave Host Controllers older than itself.

This means that for example a WildFly 8 DC should be able to work with slave HCs running JBoss AS 7.1.2. The opposite is not true, the DC must be the same or the newest version in the domain.

9.3.1 Versioning of subsystems

To help with being able to know what is compatible we have versions within the subsystems, this is stored in the subsystem's extension. When registering the subsystem you will typically see something like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    /**
     * {@inheritDoc}
     * @see
org.jboss.as.controller.Extension#initialize(org.jboss.as.controller.ExtensionContext)
     */
    @Override
    public void initialize(ExtensionContext context) {

        // IMPORTANT: Management API version != xsd version! Not all Management API changes
result in XSD changes
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);

        //Register the resource definitions
        ....
    }
    ....
}
```

Which sets the `ModelVersion` of the subsystem.

Important

Whenever something changes in the subsystem, such as:

- an attribute is added or removed from a resource
- a attribute is renamed in a resource
- an attribute has its type changed
- an attribute or operation parameter's nillable or allows expressions is changed
- an attribute or operation parameter's default value changes
- a child resource type is added or removed
- an operation is added or removed
- an operation has its parameters changed

and the current version of the subsystem has been part of a Final release of WildFly, we **must** bump the version of the subsystem.

Once it has been increased you can of course make more changes until the next Final release without more version bumps. It is also worth noting that a new WildFly release does not automatically mean a new version for the subsystem, the new version is only needed if something was changed. For example the `jaxrs` subsystem has remained on 1.0.0 for all versions of WildFly and JBoss AS 7.

You can find the `ModelVersion` of a subsystem by querying its extension:

```
domain@localhost:9990 /]
/extension=org.jboss.as.clustering.infinispan:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "module" => "org.jboss.as.clustering.infinispan",
        "subsystem" => {"infinispan" => {
            "management-major-version" => 2,
            "management-micro-version" => 0,
            "management-minor-version" => 0,
            "xml-namespaces" => [jboss:domain:infinispan:1.0",
                "urn:jboss:domain:infinispan:1.1",
                "urn:jboss:domain:infinispan:1.2",
                "urn:jboss:domain:infinispan:1.3",
                "urn:jboss:domain:infinispan:1.4",
                "urn:jboss:domain:infinispan:2.0"]
        }}
    }
}
```

9.4 The role of transformers

Now that we have mentioned the slave HCs registration process with the DC, and know about ModelVersions, it is time to mention that when registering with the DC, the slave HC will send across a list of all its subsystem ModelVersions. The DC maintains this information in a registry for each slave HC, so that it knows which transformers (if any) to invoke for a legacy slave. We will see how to write and register transformers later on in How do I write a transformer. Slave HCs from version 7.2.0 onwards will also include a list of resources that they ignore (see Ignoring resources on legacy hosts), and the DC will maintain this information in its registry. The DC will not send across any resources that it knows a slave ignores during the initial domain model transfer. When forwarding operations onto the slave HCs, the DC will skip forwarding those to slave HCs ignoring those resources.

There are two kinds of transformers:

- resource transformers
- operation transformers

The main function of transformers is to transform a subsystem to something that the legacy slave HC can understand, or to aggressively reject things that the legacy slave HC will not understand. Rejection, in this context, essentially means, that the resource or operation cannot safely be transformed to something valid on the slave, so the transformation fails. We will see later how to reject attributes in Rejecting attributes, and child resources in Reject child resource.

Both resource and operation transformers are needed, but take effect at different times. Let us use the `weld` subsystem, which is relatively simple, as an example. In JBoss AS 7.2.0 and lower it had a ModelVersion of 1.0.0, and its resource description was as follows:

```
{
                    "description" => "The configuration of the weld subsystem.",
                    "attributes" => {},
                    "operations" => {
                        "remove" => {
                            "operation-name" => "remove",
                            "description" => "Operation removing the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        },
                        "add" => {
                            "operation-name" => "add",
                            "description" => "Operation creating the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        }
                    },
                    "children" => {}
                },
```

In WildFly 8, it has a ModelVersion of 2.0.0 and has added two attributes, `require-bean-descriptor` and `non-portable` mode:

```
{
        "description" => "The configuration of the weld subsystem.",
        "attributes" => {
            "require-bean-descriptor" => {
                "type" => BOOLEAN,
                "description" => "If true then implicit bean archives without bean descriptor
file (beans.xml) are ignored by Weld",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "non-portable-mode" => {
                "type" => BOOLEAN,
                "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
CDI 1.1.",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            }
        },
        "operations" => {
            "remove" => {
                "operation-name" => "remove",
                "description" => "Operation removing the weld subsystem.",
                "request-properties" => {},
                "reply-properties" => {}
            },
            "add" => {
                "operation-name" => "add",
                "description" => "Operation creating the weld subsystem.",
                "request-properties" => {
                    "require-bean-descriptor" => {
                        "type" => BOOLEAN,
                        "description" => "If true then implicit bean archives without bean
descriptor file (beans.xml) are ignored by Weld",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    },
                    "non-portable-mode" => {
                        "type" => BOOLEAN,
                        "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
CDI 1.1.",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    }
                },
                "reply-properties" => {}
            }
        },
        "children" => {}
    }
```

In the rest of this section we will assume that we are running a DC running WildFly 8 so it will have ModelVersion 2.0.0 of the weld subsystem, and that we are running a slave using ModelVersion 1.0.0 of the weld subsystem.

Important

Transformation always takes place on the Domain Controller, and is done when sending across the initial domain model AND forwarding on operations to legacy slave HCs.

9.4.1 Resource transformers

When copying over the centralized domain configuration as mentioned in Getting the initial domain model, we need to make sure that the copy of the domain model is something that the servers running on the legacy slave HC understand. So if the centralized domain configuration had any of the two new attributes set, we would need to reject the transformation in the transformers. One reason for this is to keep things consistent, it doesn't look good if you connect to the slave HC and find attributes and/or child resources when doing `:read-resource` which are not there when you do `:read-resource-description`. Also, to make life easier for subsystem writers, most instances of the `describe` operation use a standard implementation which would include these attributes when creating the `add` operation for the server, which could cause problems there.

Another, more concrete example from the logging subsystem is that it allows a '`%K{...}`' in the pattern formatter which makes the formatter use colour:

```
<pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>
```

This '`%K{...}`' however was introduced in JBoss AS < 7.1.3 (ModelVersion 1.2.0), so if that makes it across to a slave HC running an older version, the servers **will** fail to start up. So the logging extension registers transformers to strip out the '`%K{...}`' from the attribute value (leaving '`%-5p`

`[%c]`

`(%t) %s%E%n"`') so that the old slave HC's servers can understand it.

Rejection in resource transformers

Only slave HCs from JBoss AS 7.2.0 and newer inform the DC about their ignored resources (see Ignoring resources on legacy hosts). This means that if a transformer on the DC rejects transformation for a legacy slave HC, exactly what happens to the slave HC depends on the version of the slave HC. If the slave HC is:

- *older than 7.2.0* - the DC has no means of knowing if the slave HC has ignored the resource being rejected or not. So we log a warning on the DC, and send over the serialized part of that model anyway. If the slave HC has ignored the resource in question, it does not apply it. If the slave HC has not ignored the resource in question, it will apply it, but no failure will happen until it tries to start a server which references this bad configuration.
- *7.2.0 or newer* - If a resource is ignored on the slave HC, the DC knows about this, and will not attempt to transform or send the resource across to the slave HC. If the resource transformation is rejected, we know the resource was not ignored on the slave HC and so we can aggressively fail the transformation, which in turn will cause the slave HC to fail to start up.

9.4.2 Operation transformers

When An operation changes something in the domain configuration the operation gets sent across to the slave HCs to update their domain models. The slave HCs then forward this operation onto the affected servers. The same considerations as in Resource transformers are true, although operation transformers give you quicker 'feedback' if something is not valid. If you try to execute:

```
/profile=full/subsystem=weld:write-attribute(name=require-bean-descriptor, value=false)
```

This will fail on the legacy slave HC since its version of the subsystem does not contain any such attribute. However, it is best to aggressively reject in such cases.

Rejection in operation transformers

For transformed operations we can always know if the operation is on an ignored resource in the legacy slave HC. In 7.2.0 onwards, we know this through the DC's registry of ignored resources on the slave. In older versions of slaves, we send the operation across to the slave, which tries to invoke the operation. If the operation is against an ignored resource we inform the DC about this fact. So as part of the transformation process, if something gets rejected we can (and do!) fail the transformation aggressively. If the operation invoked on the DC results in the operation being sent across to 10 slave HCs and one of them has a legacy version which ends up rejecting the transformation, we rollback the operation across the whole domain.

9.4.3 Different profiles for different versions

Now for the `weld` example we have been using there is a slight twist. We have the new `require-bean-descriptor` and `non-portable-mode` attributes. These have been added in WildFly 8 which supports Java EE 7, and thus CDI 1.1. JBoss AS 7.x supports Java EE 6, and thus CDI 1.0. In CDI 1.1 the values of these attributes are tweakable, so they can be set to either `true` or `false`. The default behaviour for these in CDI 1.1, if not set, is that they are `false`. However, for CDI 1.0 these were not tweakable, and with the way the subsystem in JBoss AS 7.x worked is similar to if they are set to `true`.

The above discussion implies that to use the weld subsystem on a legacy slave HC, the `domain.xml` configuration for it must look like:

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="true"
      non-portable-mode="true"/>
```

We will see the exact mechanics for how this is actually done later but in short when pushing this to a legacy slave DC we register transformers which reject the transformation if these attributes are not set to `true` since that implies some behaviour not supported on the legacy slave DC. If they are `true`, all is well, and the transformers discard, or remove, these attributes since they don't exist in the legacy model. This removal is fine since they have the values which would result in the behaviour assumed on the legacy slave HC.

That way the older slave HCs will work fine. However, we might also have WildFly 8 slave HCs in our domain, and they are missing out on the new features introduced by the attributes introduced in ModelVersion 2.0.0. If we do

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="false"
      non-portable-mode="false"/>
```

then it will fail when doing transformation for the legacy controller. The solution is to put these in two different profiles in `domain.xml`

```
<domain>
....
  <profiles>
    <profile name="full">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="false"
        non-portable-mode="false"/>
      ...
    </profile>
    <profile name="full-legacy">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="true"
        non-portable-mode="true"/>
      ...
    </profile>
  </profiles>
  ...
  <server-groups>
    <server-group name="main-server-group" profile="full">
      ....
    <server-group>
    <server-group name="main-server-group-legacy" profile="full-legacy">
      ....
    <server-group>
  </server-groups>
</domain>
```

Then have the HCs using WildFly 8 make their servers reference the `main-server-group` server group, and the HCs using older versions of WildFly 8 make their servers reference the `main-server-group-legacy` server group.

Ignoring resources on legacy hosts

Booting the above configuration will still cause problems on legacy slave HCs, especially if they are JBoss AS 7.2.0 or later. The reason for this is that when they register themselves with the DC it lets the DC know which `ignored resources` they have. If the DC comes to transform something it should reject for a slave HC and it is not part of its ignored resources it will aggressively fail the transformation. Versions of JBoss AS older than 7.2.0 still have this ignored resources mechanism, but don't let the DC know about what they have ignored so the DC cannot reject aggressively - instead it will log some warnings. However, it is still good practice to ignore resources you are not interested in regardless of which legacy version the slave HC is running.

To ignore the profile we cannot understand we do the following in the legacy slave HC's `host.xml`

```
<host xmlns="urn:jboss:domain:1.3" name="slave">
...
    <domain-controller>
       <remote host="${jboss.test.host.master.address}" port="${jboss.domain.master.port:9999}"
security-realm="ManagementRealm">
            <ignored-resources type="profile">
                <instance name="full-legacy"/>
            </ignored-resources>
       </remote>
    </domain-controller>
....
</host>
```

Important

Any top-level resource type can be ignored `profile`, `extension`, `server-group` etc. Ignoring a resource instance ignores that resource, and all its children.

9.5 How do I know what needs to be transformed?

There is a set of related classes in the `org.wildfly.legacy.util` package to help you determine this. These now live at
https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/java/org/wildfly/legacy/util.
They are all runnable in your IDE, just start the WildFly or JBoss AS 7 instances as described below.

9.5.1 Getting data for a previous version

https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/resources/legacy-models contains the output for the previous WildFly/JBoss AS 7 versions, so check if the files for the version you want to check backwards compatibility are there yet. If not, then you need to do the following to get the subsystem definitions:

1. Start the **old** version of WildFly/JBoss AS 7 using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.GrabModelVersionsUtil`, which will output the subsystem versions to `target/standalone-model-versions-running.dmr`
3. Run `org.wildfly.legacy.util.DumpStandaloneResourceDefinitionUtil` which will output the full resource definition to `target/standalone-resource-definition-running.dmr`
4. Stop the running version of WildFly/JBoss AS 7

9.5.2 See what changed

To do this follow the following steps

1. Start the **new** version of WildFly using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.CompareModelVersionsUtil` and answer the following questions"
   1. Enter Legacy AS version:
      - If it is known version in the `tools/src/test/resources/legacy-models` folder, enter the version number.
      - If it is a not known version, and you got the data yourself in the last step, enter '`running`'
   2. Enter type:
      - Answer 'S'
   3. Read from target directory or from the legacy-models directory:
      - If it is known version in the `controller/src/test/resources/legacy-models` folder, enter '`l`'.
      - If it is a not known version, and you got the data yourself in the last step, enter '`t`'
   4. Report on differences in the model when the management versions are different?:
      - Answer 'y'

Here is some example output, as a subsystem developer you can ignore everything down to `====== Comparing subsystem models ======`:

```
Enter legacy AS version: 7.2.0.Final
Using target model: 7.2.0.Final
Enter type [S](standalone)/H(host)/D(domain)/F(domain + host):S
Read from target directory or from the legacy-models directory - t/[l]:
Report on differences in the model when the management versions are different? y/[n]: y
Reporting on differences in the model when the management versions are different
Loading legacy model versions for 7.2.0.Final....
Loaded legacy model versions
Loading model versions for currently running server...
Oct 01, 2013 6:26:03 PM org.xnio.Xnio <clinit>
INFO: XNIO version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 4.0.0.Beta1
Loaded current model versions
Loading legacy resource descriptions for 7.2.0.Final....
Loaded legacy resource descriptions
Loading resource descriptions for currently running STANDALONE...
Loaded current resource descriptions
Starting comparison of the current....

====== Comparing core models ======
-- SNIP --

====== Comparing subsystem models ======
-- SNIP --
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
-- SNIP --
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]

Done comparison of STANDALONE!
```

So we can see that for the `remoting` subsystem, we have added a child type called `http-connector`, and we have added an attribute called `protocol` (they are missing in legacy).
in the `weld` subsystem, we have added the `require-bean-descriptor` and `non-portable-mode` attributes in the current version. It will also point out other issues like changed attribute types, changed defaults etc.

Warning

Note that CompareModelVersionsUtil simply inspects the raw resource descriptions of the specified legacy and current models. Its results show the differences between the two. They do not take into account whether one or more transformers have already been written for those versions differences. You will need to check that transformers are not already in place for those versions.

One final point to consider are that some subsystems register runtime-only resources and operations. For example the `modcluster` subsystem has a `stop` method. These do not get registered on the `DC`, e.g. there is no `/profile=full-ha/subsystem=modcluster:stop` operation, it only exists on the servers, for example `/host=xxx/server=server-one/subsystem=modcluster:stop`. What this means is that you don't have to transform such operations and resources. The reason is they are not callable on the DC, and so do not need propagation to the servers in the domain, which in turn means no transformation is needed.

9.6 How do I write a transformer?

There are two APIs available to write transformers for a resource. There is the original low-level API where you register transformers directly, the general idea is that you get hold of a `TransformersSubRegistration` for each level and implement the `ResourceTransformer`, `OperationTransformer` and `PathAddressTransformer` interfaces directly. It is, however, a pretty complex thing to do, so we recommend the other approach. For completeness here is the entry point to handling transformation in this way.

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    static void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_1_0(subsystem);
        registerTransformers_1_2_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private static void registerTransformers_1_1_0(final SubsystemRegistration subsystem) {
        final ModelVersion version = ModelVersion.create(1, 1, 0);

        //The default resource transformer forwards all operations
        final TransformersSubRegistration registration =
subsystem.registerModelTransformers(version, ResourceTransformer.DEFAULT);
        final TransformersSubRegistration child =
registration.registerSubResource(PathElement.pathElement("child"));
        //We can do more things on the TransformersSubRegistation instances


        registerRelayTransformers(stack);
    }
```

Having implemented a number of transformers using the above approach, we decided to simplify things, so we introduced the `org.jboss.as.controller.transform.description.ResourceTransformationDescriptionBu` API. It is a lot simpler and avoids a lot of the duplication of functionality required by the low-level API approach. While it doesn't give you the full power that the low-level API does, we found that there are very few places in the WildFly codebase where this does not work, so we will focus on the `ResourceTransformationDescriptionBuilder` API here. (If you come across a problem where this does not work, get in touch with someone from the WildFly Domain Management Team and we should be able to help). The builder API makes all the nasty calls to `TransformersSubRegistration` for you under the hood. It also allows you to fall back to the low-level API in places, although that will not be covered in the current version of this guide. The entry point for using the builder API here is taken from the WeldExtension (in current WildFly this has ModelVersion 2.0.0).

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

Here we register a `discard check` and a `reject check`. As mentioned in Attribute transformation lifecycle all attributes are inspected for whether they should be discarded first. Then all attributes which were not discarded are checked for if they should be rejected. We will dig more into what this code means in the next few sections, but in short it means that we discard the `require-bean-descriptor` and `non-portable` attributes on the `weld` subsystem resource if they have the value `true`. If they have any other value, they will not get discarded and so reach the reject check, which will reject the transformation of the attributes if they have any other value.

Here we are saying that we should discard the `require-bean-descriptor` and `non-portable-mode` attributes on the `weld` subsystem resource if they are undefined, and reject them if they are defined. So that means that if the weld subsystem looks like

```
{
        "non-portable-mode" => false,
        "require-bean-descriptor" => false
    }
```

or

```
{
            "non-portable-mode" => undefined,
            "require-bean-descriptor" => undefined
      }
```

or any other combination (the default values for these attributes if undefined is `false`) we will reject the transformation for the slave legacy HC.

If the resource has true for these attributes:

```
{
            "non-portable-mode" => true,
            "require-bean-descriptor" => true
      }
```

they both get discarded (i.e. removed), so they will not get inspected for rejection, and an empty model not containing these attributes gets sent to the legacy HC.

Here we will discuss this API a bit more, to outline the most important features/most commonly needed tasks.

9.6.1 ResourceTransformationDescriptionBuilder

The `ResourceTransformationDescriptionBuilder` contains transformations for a resource type. The initial one is for the subsystem, obtained by the following call:

```
ResourceTransformationDescriptionBuilder subsystemBuilder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
```

The `ResourceTransformationDescriptionBuilder` contains functionality for how to handle child resources, which we will look at in this section. It is also the entry point to how to handle transformation of attributes as we will see in AttributeTransformationDescriptionBuilder. Also, it allows you to further override operation transformation as discussed in OperationTransformationOverrideBuilder. When we have finished with our builder, we register it with the `SubsystemRegistration` against the target ModelVersion.

```
TransformationDescription.Tools.register(subsystemBuilder.build(), subsystem,
ModelVersion.create(1, 0, 0));
```

Important

If you have several old ModelVersions you could be transforming to, you need a separate builder for each of those.

Silently discard child resources

To make the `ResourceTransformationDescriptionBuilder` do something, we need to call some of its methods. For example, if we want to silently discard a child resource, we can do

```
subsystemBuilder.discardChildResource(PathElement.pathElement("child", "discarded"));
```

This means that any usage of `/subsystem=my-subsystem/child=discarded` never make it to the legacy slave HC running ModelVersion 1.0.0. During the initial domain model transfer, that part of the serialized domain model is stripped out, and any operations on this address are not forwarded on to the legacy slave HCs running that version of the subsystem. (For brevity this section will leave out the leading `/profile=xxx` part used in domain mode, and use `/subsystem=my-subsystem` as the 'top-level' address).

Warning

Note that discarding, although the simplest option in theory, is **rarely the right thing to do**.

The presence of the defined child normally implies some behaviour on the DC, and that behaviour is not available on the legacy slave HC, so normally rejection is a better policy for those cases. Remember we can have different profiles targeting different groups of versions of legacy slave HCs.

Reject child resource

If we want to reject transformation if a child resource exists, we can do

```
subsystemBuilder.rejectChildResource(PathElement.pathElement("child", "reject"));
```

Now, if there are any legacy slaves running ModelVersion 1.0.0, any usage of `/subsystem=my-subsystem/child=reject` will get rejected for those slaves. Both during the initial domain model transfer, and if any operations are invoked on that address. For example the `remoting` subsystem did not have a `http-connector=*` child until ModelVersion 2.0.0, so it is set up to reject that child when transforming to legacy HCs for all previous ModelVersions (1.1.0, 1.2.0 and 1.3.0). (See Rejection in resource transformers and Rejection in operation transformers for exactly what happens when something is rejected).

Redirect address for child resource

Sometimes we rename the addresses for a child resource between model versions. To do that we use one of the `addChildRedirection()` methods, note that these also return a builder for the child resource (since we are not rejecting or discarding it), we can do this for all children of a given type:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild"),
PathElement.pathElement("oldChild");
```

Now, in the initial domain transfer `/subsystem=my-subsystem/newChild=test` becomes `/subsystem=my-subsystem/oldChild=test`. Similarly all operations against the former address get mapped to the latter when executing operations on the DC before sending them to the legacy slave HC running ModelVersion 1.1.0 of the subsystem.

We can also rename a specific named child:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild", "newName"),
PathElement.pathElement("oldChild", "oldName");
```

Now, `/subsystem=my-subsystem/newChild=newName` becomes `/subsystem=my-subsystem/oldChild=oldName` both in the initial domain transfer, and when mapping operations to the legacy slave. For example, under the web subsystem `ssl=configuration` got renamed to `configuration=ssl` in later versions, meaning we need a redirect from `configuration=ssl` to `ssl=configuration` in its transformers.

Getting a child resource builder

Sometimes we don't want to transform the subsystem resource, but we want to transform something in one of its child resources. Again, since we are not discarding or rejecting, we get a reference to the builder for the child resource.

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    //We don't actually want to transform anything in /subsystem-my-subsystem/some-child=*
either :-)
    //We are interested in /subsystem-my-subsystem/some-child=*/another-level
    ResourceTransformationDescriptionBuilder anotherBuilder =
       childBuilder.addChildResource(PathElement.pathElement("another-level"));

    //Use anotherBuilder to add child-resource and/or attribute transformation
    ....
```

9.6.2 AttributeTransformationDescriptionBuilder

To transform attributes you call `ResourceTransformationDescriptionBuilder.getAttributeBuilder()` which returns you a `AttributeTransformationDescriptionBuilder` which is used to define transformation for the resource's attributes. For example this gets the attribute builder for the subsystem resource:

```
AttributeTransformationDescriptionBuilder attributeBuilder =
subSystemBuilder.getAttributeBuilder();
```

or we could get it for one of the child resources:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    AttributeTransformationDescriptionBuilder attributeBuilder =
childBuilder.getAttributeBuilder();
```

The attribute transformations defined by the `AttributeTransformationDescriptionBuilder` will also impact the parameters to all operations defined on the resource. This means that if you have defined the `example` attribute of `/subsystem=my-subsystem/some-child=*` to reject transformation if its value is `true`, the inital domain transfer will reject if it is `true`, also the transformation of the following operations will reject:

```
/subsystem=my-subsystem/some-child=test:add(example=true)
    /subsystem=my-subsystem:write-attribute(name=example, value=true)
    /subsystem=my-subsystem:custom-operation(example=true)
```

The following operations will pass in this example, since the `example` attribute is not getting set to `true`

```
/subsystem=my-subsystem/some-child=test:add(example=false)
    /subsystem=my-subsystem/some-child=test:add()            //Here it 'example' is simply left
undefined
    /subsystem=my-subsystem:write-attribute(name=example, value=false)
    /subsystem=my-subsystem:undefine-attribute(name=example)  //Again this makes 'example'
undefined
    /subsystem=my-subsystem:custom-operation(example=false)
```

For the rest of the examples in this section we assume that the `attributeBuilder` is for `/subsystem=my-subsystem`

Attribute transformation lifecycle

There is a well defined lifecycle used for attribute transformation that is worth explaining before jumping into specifics. Transformation is done in the following phases, in the following order:

1. `discard` - All attributes in the domain model transfer or invoked operation that have been registered for a discard check, are checked to see if the attribute should be discarded. If an attribute should be discarded, it is removed from the resource's attributes/operation's parameters and it does not get passed to the next phases. Once discarded it does not get sent to the legacy slave HC.
2. `reject` - All attributes that have been registered for a reject check (and which not have been discarded) are checked to see if the attribute should be rejected. As explained in Rejection in resource transformers and Rejection in operation transformers exactly what happens when something is rejected varies depending on whether we are transforming a resource or an operation, and the version of the legacy slave HC we are transforming for. If a transformer rejects an attribute, all other reject transformers still get invoked, and the next phases also get invoked. This is because we don't know in all cases what will happen if a reject happens. Although this might sound cumbersome, in practice it actually makes it easier to write transformers since you only need one kind regardless of if it is a resource, an operation, and legacy slave HC version. However, as we will see in Common transformation use-cases, it means some extra checks are needed when writing reject and convert transformers.
3. `convert` - All attributes that have been registered for conversion are checked to see if the attribute should be converted. If the attribute does not exist in the original operation/resource it may be introduced. This is useful for setting default values for the target legacy slave HC.
4. `rename` - All attributes registered for renaming are renamed.

Next, let us have a look at how to register attributes for each of these phases.

Discarding attributes

The general idea behind a discard is that we remove attributes which do not exist in the legacy slave HC's model. However, as hopefully described below, we normally can't simply discard everything, we need to check the values first.

To discard an attribute we need an instance of `org.jboss.as.controller.transform.description.DiscardAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
DiscardAttributeChecker discardCheckerA = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1", "attr2");
```

As shown, you can register the `DiscardAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `DiscardAttributeChecker` instances for different attributes:

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerA, "attr2");
```

Note that you can only have one `DiscardAttributeChecker` per attribute, so the following would cause an error (if running with assertions enabled, otherwise `discardCheckerB` will overwrite `discardCheckerA`):

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerB, "attr1");
```

The DiscardAttributeChecker interface

`org.jboss.as.controller.transform.description.DiscardAttributeChecker` contains both the `DiscardAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple:

```
public interface DiscardAttributeChecker {

    /**
     * Returns {@code true} if the attribute should be discarded if expressions are used
     *
     * @return whether to discard if expressions are used
     */
    boolean isDiscardExpressions();
```

Return `true` here to discard the attribute if it is an expression. If it is an expression, and this method returns `true`, the `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Returns {@code true} if the attribute should be discarded if it is undefined
     *
     * @return whether to discard if the attribute is undefined
     */
    boolean isDiscardUndefined();
```

Return `true` here to discard the attribute if it is `undefined`. If it is `undefined`, and this method returns `true`, the `isDiscardExpressions`, `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Gets whether the given operation parameter can be discarded
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter.
     * @param attributeValue the value of the operation parameter.
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     *
     * @return {@code true} if the operation parameter value should be discarded, {@code false}
otherwise.
     */
    boolean isOperationParameterDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to discard the operation.

```
/**
     * Gets whether the given attribute can be discarded
     *
     * @param address the address of the resource
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context the context of the transformation
     *
     * @return {@code true} if the attribute value should be discarded, {@code false} otherwise.
     */
    boolean isResourceAttributeDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
}
```

DiscardAttributeChecker helper classes/implementations

`DiscardAttributeChecker` contains a few helper implementations for the most common cases to save you writing the same stuff again and again.

DiscardAttributeChecker.DefaultDiscardAttributeChecker

`DiscardAttributeChecker.DefaultDiscardAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `isResourceAttributeDiscardable()` and `isOperationParameterDiscardable()` methods call the following method.

```
protected abstract boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context);
```

All you loose, in the case of an operation transformation, is the name of the transformed operation. The constructor of `DiscardAttributeChecker.DefaultDiscardAttributeChecker` also allows you to define values for `isDiscardExpressions()` and `isDiscardUndefined()`.

DiscardAttributeChecker.DiscardAttributeValueChecker

This is another convenience class, which allows you to discard an attribute if it has one or more values. Here is a real-world example from the `jpa` subsystem:

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                   new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                   JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

We will come back to the reject checks in the Rejecting attributes section. We are saying that we should discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` attribute if it has the value `deep`. The reasoning here is that this attribute did not exist in the old model, but the legacy slave HCs *implied behaviour* is that this was `deep`. In the current version we added the possibility to toggle this setting, but only `deep` is consistent with what is available in the legacy slave HC. In this case we are using the constructor for `DiscardAttributeChecker.DiscardAttributeValueChecker` which says don't discard if it uses expressions, and discard if it is `undefined`. If it is `undefined` in the current model, looking at the default value of `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE`, it is `deep`, so a discard is in line with the implied legacy behaviour. If an expression is used, we cannot discard since we have no idea what the expression will resolve to on the slave HC.

DiscardAttributeChecker.ALWAYS

`DiscardAttributeChecker.ALWAYS` will always discard an attribute. Use this sparingly, since normally the presence of an attribute in the current model implies some behaviour should be turned on, and if that does not exist in the legacy model it implies that that behaviour does not exist in the legacy slave HC and its servers. Normally the legacy slave HC's subsystem has some implied behaviour which is better checked for by using a `DiscardAttributeChecker.DiscardAttributeValueChecker`. One valid use for `DiscardAttributeChecker.ALWAYS` can be found in the `ejb3` subsystem:

```
private static void registerTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance()
                .getAttributeBuilder()
                 ...
                // We can always discard this attribute, because it's meaningless without the
security-manager subsystem, and
                // a legacy slave can't have that subsystem in its profile.
                .setDiscard(DiscardAttributeChecker.ALWAYS,
EJB3SubsystemRootResourceDefinition.DISABLE_DEFAULT_EJB_PERMISSIONS)
   ...
```

As the comment says, this attribute only makes sense with the security-manager susbsystem, which does not exist on legacy slaves running ModelVersion 1.1.0 of the `ejb3` subsystem.

DiscardAttributeChecker.UNDEFINED

`DiscardAttributeChecker.UNDEFINED` will discard an attribute if it is `undefined`. This is normally safer than `DiscardAttributeChecker.ALWAYS` since the attribute is not set in the current model, we don't need to send it to the legacy model. However, you should check that this attribute not existing in the legacy slave HC, implies the same functionality as being undefined in the current DC.

Rejecting attributes

The next step is to check attributes and values which we know for sure will not work on the target legacy slave HC.

To reject an attribute we need an instance of `org.jboss.as.controller.transform.description.RejectAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
RejectAttributeChecker rejectCheckerA = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1", "attr2");
```

As shown you can register the `RejectAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `RejectAttributeChecker` instances for different attributes:

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr2");
```

You can also register several `RejectAttributeChecker` instances per attribute

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr1, "attr2");
```

In this case `attr1` gets both `rejectCheckerA` and `rejectCheckerB`. For attributes with several `RejectAttributeChecker` registered, they get processed in the order that they have been added. So when checking `attr1` for rejection, `rejectCheckerA` gets run before `rejectCheckerB`. As mentioned in Attribute transformation lifecycle, if an attribute is rejected, we still invoke the rest of the reject checkers.

The RejectAttributeChecker interface

`org.jboss.as.controller.transform.description.RejectAttributeChecker` contains both the `RejectAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple, and its main methods are similar to `DiscardAttributeChecker`:

```
public interface RejectAttributeChecker {
    /**
     * Determines whether the given operation parameter value is not understandable by the
target process and needs
     * to be rejected.
     *
     * @param address        the address of the operation
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param operation      the operation executed. This is unmodifiable.
     * @param context        the context of the transformation
     * @return {@code true} if the parameter value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to reject the operation.

```
/**
     * Gets whether the given resource attribute value is not understandable by the target
process and needs
     * to be rejected.
     *
     * @param address        the address of the resource
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context        the context of the transformation
     * @return {@code true} if the attribute value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
/**
     * Returns the log message id used by this checker. This is used to group it so that all
attributes failing a type of rejction
     * end up in the same error message
     *
     * @return the log message id
     */
    String getRejectionLogMessageId();
```

Here we need a unique id for the log message from the `RejectAttributeChecker`. It is used to group rejected attributes by their log message. A typical implementation will contain {{return getRejectionLogMessage(Collections.<String, ModelNode>emptyMap());}

```
/**
     * Gets the log message if the attribute failed rejection
     *
     * @param attributes a map of all attributes failed in this checker and their values
     * @return the formatted log message
     */
    String getRejectionLogMessage(Map<String, ModelNode> attributes);
```

Here we return a message saying why the attributes were rejected, with the possiblity to format the message to include the names of all the rejected attributes and the values they had.

```
}
```

RejectAttributeChecker helper classes/implementations

`RejectAttributeChecker` contains a few helper classes for the most common scenarios to save you from writing the same stuff again and again.

RejectAttributeChecker.DefaultRejectAttributeChecker

`RejectAttributeChecker.DefaultRejectAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `rejectOperationParameter()` and `rejectResourceAttribute()` methods call the following method.

```
protected abstract boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

RejectAttributeChecker.DEFINED

`RejectAttributeChecker.DEFINED` is used to reject any attribute that has a defined value. Normally this is because the attribute does not exist on the target legacy slave HC. A typical use case for these is for the *implied behaviour* example we looked at in the `jpa` subsystem in DiscardAttributeChecker.DiscardAttributeValueChecker

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                   new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                   JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

So we discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` value if it is not an expression, and also has the value `deep`. Now if it was not discarded, it would will still be defined so we reject it.

Important

Reject and discard often work in pairs.

RejectAttributeChecker.SIMPLE_EXPRESSIONS

`RejectAttributeChecker.SIMPLE_EXPRESSIONS` can be used to reject an attribute that contains expressions. This was used a lot for transformations to subsystems in JBoss AS 7.1.x, since we had not fully realized the importance of where to support expressions until JBoss AS 7.2.0 was released, so a lot of attributes in earlier versions were missing expressions support.

RejectAttributeChecker.ListRejectAttributeChecker

The `RejectAttributeChecker}}s we have seen so far work on simple attributes, i.e. where the attribute has a ModelType which is one of the primitives. We also have a {{RejectAttributeChecker.ListRejectAttributeChecker` which allows you to define a checker for the elements of a list, when the type of an attribute is `ModelType.LIST`.

```
attributeBuilder
            .addRejectCheck(new ListRejectAttributeChecker(RejectAttributeChecker.EXPRESSIONS),
"attr1");
```

For `attr1` it will check each element of the list and run `RejectAttributeChecker.EXPRESSIONS` to check that each element is not an expression. You can of course pass in another kind of `RejectAttributeChecker` to check the elements as well.

RejectAttributeChecker.ObjectFieldsRejectAttributeChecker

For attributes where the type is `ModelType.OBJECT` we have `RejectAttributeChecker.ObjectFieldsRejectAttributeChecker` which allows you to register different reject checkers for the different fields of the registered object.

```
Map<String, RejectAttributeChecker> fieldRejectCheckers = new HashMap<String,
RejectAttributeChecker>();
    fieldRejectCheckers.put("time", RejectAttributeChecker.SIMPLE_EXPRESSIONS);
    fieldRejectCheckers.put("unit", "Lunar Month");
    attributeBuilder
            .addRejectCheck(new ObjectFieldsRejectAttributeChecker(fieldRejectCheckers),
"attr1");
```

Now if `attr1` is a complex type where `attr1.get("time").getType() == ModelType.EXPRESSION` or `attr1.get("unit").asString().equals("Lunar Month")` we reject the attribute.

Converting attributes

To convert an attribute you register an `org.jboss.as.controller.transform.description.AttributeConverter` instance against the attributes you want to convert:

```
AttributeConverter converterA = ...;
    AttributeConverter converterB = ...;
    attributeBuilder
            .setValueConverter(converterA, "attr1", "attr2");
    attributeBuilder
            .setValueConverter(converterB, "attr3");
```

Now if `attr1` and `attr2` get converted with `converterA`, while `attr3` gets converted with `converterB`.

The AttributeConverter interface

The `AttributeConverter` interface gets called for each attribute for which the `AttributeConverter` has been registered

```
public interface AttributeConverter {

    /**
     * Converts an operation parameter
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter
     * @param attributeValue the value of the operation parameter to be converted
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     */
    void convertOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter for which the con. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a write-attribute operation. To change the attribute value, you modify the `attributeValue`.

```
/**
     * Converts a resource attribute
     *
     * @param address the address of the operation
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute to be converted
     * @param context the context of the transformation
     */
    void convertResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. To change the attribute value, you modify the `attributeValue`.

```
}
```

A hypothetical example is if the current and legacy subsystems both contain an attribute called `timeout`. In the legacy model this was specified to be milliseconds, however in the current model it has been changed to be seconds, hence we need to convert the value when sending it to slave HCs using the legacy model:

```
AttributeConverter secondsToMs = new AttributeConverter.DefaultAttributeConverter() {
                  @Override
                  protected void convertAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                           TransformationContext context) {
                      if (attributeValue.isDefined()) {
                           int seconds = attributeValue.asInt();
                           int milliseconds = seconds * 1000;
                           attributeValue.set(milliseconds);
                      }
                  }
          };

     attributeBuilder.
          .setValueConverter(secondsToMs , "timeout")
```

We need to be a bit careful here. If the `timeout` attribute is an expression our nice conversion will not work, so we need to add a reject check to make sure it is not an expression as well:

```
attributeBuilder.
          .addRejectCheck(SIMPLE_EXPRESSIONS, "timeout")
          .setValueConverter(secondsToMs , "timeout")
```

Now it should be fine.

`AttributeConverter.DefaultAttributeConverter` is is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the convertOperationParameter() and convertResourceAttribute() methods call the following method.

```
protected abstract void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker` and `DefaultRejectAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

Introducing attributes during transformation

Say both the current and the legacy models have an attribute called `port`. In the legacy version this attribute had to be specified, and the default xml configuration had `1234` for its value. In the current version this attribute has been made optional with a default value of `1234` so that it does not need to be specified. When transforming to a slave HC using the old version we will need to introduce this attribute if the new model does not contain it:

```
attributeBuilder.
         setValueConverter(AttributeConverter.Factory.createHardCoded(new ModelNode(1234) true),
"port");
```

So what this factory method does is to create an implementation of `AttributeConverter.DefaultAttributeConverter` where in `convertAttribute()` we set `attributeValue` to have the value `1234` if it is `undefined`. As long as `attributeValue` gets set in that method it will get set in the model, regardless of if it existed already or not.

Renaming attributes

To rename an attribute, you simply do

```
attributeBuilder.addRename("my-name", "legacy-name");
```

Now, in the initial domain transfer to the legacy slave HC, we rename `/subsystem=my-subsystem`'s `my-name` attribute to `legacy-name`. Also, the operations involving this attribute are affected, so

```
/subsystem=my-subsystem/:add(my-name=true)  ->
         /subsystem=my-subsystem/:add(legacy-name=true)
    /subsystem=my-subsystem:write-attribute(name=my-name, value=true) ->
         /subsystem=my-subsystem:write-attribute(name=legacy-name, value=true)
    /subsystem=my-subsystem:undefine-attribute(name=my-name) ->
         /subsystem=my-subsystem:undefine-attribute(name=legacy-name)
```

9.6.3 OperationTransformationOverrideBuilder

All operations on a resource automatically get the same transformations on their parameters as set up by the `AttributeTransformationDescriptionBuilder`. In some cases you might want to change this, so you can use the `OperationTransformationOverrideBuilder`, which is got from:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
```

In this case the operation will now no longer inherit the attribute/operation parameter transformations, so they are effectively turned off. In other cases you might want to include them by calling `inheritResourceAttributeDefinitions()`, and to include some more checks (the `OperationTransformationBuilder` interface has all the methods found in `AttributeTransformationBuilder`:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.inheritResourceAttributeDefinitions();
    operationBuilder.setValueConverter(AttributeConverter.Factory.createHardCoded(new
ModelNode(1234) true), "port");
```

You can also rename operations, in this case the operation `some-operation` gets renamed to `legacy-operation` before getting sent to the legacy slave HC.

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.rename("legacy-operation");
```

9.7 Evolving transformers with subsystem ModelVersions

Say you have a subsystem with ModelVersions 1.0.0 and 1.1.0. There will (hopefully!) already be transformers in place for 1.1.0 to 1.0.0 transformations. Let's say that the transformers registration looks like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 1;
    private static final int MANAGEMENT_API_MINOR_VERSION = 1;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now say we want to do a new version of the model. This new version contains a new attribute called 'new-attr' which cannot be defined when transforming to 1.1.0, we bump the model version to 2.0.0:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }
```

There are a few ways to evolve your transformers:

- The old way
- Chained transformers

9.7.1 The old way

This is the way that has been used up to WildFly 8.x. However, in WildFly 9, it is strongly recommended to migrate to what is mentioned in Chained transformers

Now we need some new transformers from the current ModelVersion to 1.1.0 where we reject any defined occurrances of our new attribute `new-attr`:

```
private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
        registerTransformers_1_1_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private void registerTransformers_1_1_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 1, 0));
    }
```

So that is all well and good, however we also need to take into account that `new-attr` **does not exist in ModelVersion 1.0.0 either**, so we need to extend our transformer for 1.0.0 to reject it there as well. As you can see 1.0.0 also rejects a defined 'attr1' in addition to the 'new-attr'(which is rejected in both versions).

```
/**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1", "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now `new-attr` will be rejected if defined for all previous model versions.

9.7.2 Chained transformers

Since 'The old way' had a lot of duplication of code, in WildFly 9 we now have chained transformers. You obtain a `ChainedTransformationDescriptionBuilder` which is a different entry point to the `ResourceTransformationDescriptionBuilder` we have seen earlier. Each `ResourceTransformationDescriptionBuilder` deals with transformation across one version delta.

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ModelVersion version1_1_0 = ModelVersion.create(1, 1, 0);
        ModelVersion version1_0_0 = ModelVersion.create(1, 0, 0);

        ChainedTransformationDescriptionBuilder chainedBuilder =

TransformationDescriptionBuilder.Factory.createChainedSubystemInstance(subsystem.getSubsystemVersi
//Differences between the current version and 1.1.0
        ResourceTransformationDescriptionBuilder builder110 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_1_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();

        //Differences between the 1.1.0 and 1.0.0
        ResourceTransformationDescriptionBuilder builder100 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_0_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();

        chainedBuilder.buildAndRegister(subsystem, new ModelVersion[]{version1_0_0,
version1_1_0});
```

The `buildAndRegister(ModelVersion[]... chains)` method registers a chain consisting of the built `builder110` and `builder100` for transformation to 1.0.0, and a chain consisting of the built `builder110` for transformation to 1.1.0. It allows you to specify more than one chain.

Now when transforming from the current version to 1.0.0, the resource is first transformed from the current version to 1.1.0 (which rejects a defined `new-attr`) and then it is transformed from 1.1.0 to 1.0.0 (which rejects a defined `attr1`). So when evolving transformers you should normally only need to add things to the last version delta. The full current-to-1.1.0 transformation is run before the 1.1.0-to-1.0.0 transformation is run.

One thing worth pointing out that the value returned by `TransformationContext.readResource(PathAddress address)` and `TransformationContext.readResourceFromRoot(PathAddress address)` which you can use from your custom `RejectAttributeChecker`, `DiscardAttributeChecker` and `AttributeConverter` behaves slightly differently depending on if you are transforming an operation or a resource.

During *resource transformation* this will be the latest model, so in our above example, in the current-to-1.1.0 transformation it will be the original model. In the 1.1.0-to-1.0.0 transformation, it will be the result of the current-to-1.1.0 transformation.

During *operation transformation* these methods will always return the original model (we are transforming operations, not resources!).

In WildFly 9 we are now less aggressive about transforming to all previous versions of WildFly, however we still have a lot of good tests for running against 7.1.x, 8. Also, for Red Hat employees we have tests against EAP versions. These tests no longer get run by default, to run them you need to specify some system properties when invoking maven. They are:

- `-Djboss.test.transformers.subsystem.old` - enables the non-default subsystem tests.
- -Djboss.test.transformers.eap - (Red Hat developers only), enables the eap tests, but only the ones run by default. If run in conjunction with `-Djboss.test.transformers.subsystem.old` you get all the possible subsystem tests run.
- -Djboss.test.transformers.core.old - enables the non-default core model tests.

9.8 Testing transformers

To test transformation you need to extend `org.jboss.as.subsystem.test.AbstractSubsystemTest` or `org.jboss.as.subsystem.test.AbstractSubsystemBaseTest`. Then, in order to have the best test coverage possible, you should test the fullest configuration that will work, and you should also test configurations that don't work if you have rejecting transformers registered. The following example is from the threads subsystem, and I have only included the tests against 7.1.2 - there are more! First we need to set up our test:

```
public class ThreadsSubsystemTestCase extends AbstractSubsystemBaseTest {
    public ThreadsSubsystemTestCase() {
        super(ThreadsExtension.SUBSYSTEM_NAME, new ThreadsExtension());
    }

    @Override
    protected String getSubsystemXml() throws IOException {
        return readResource("threads-subsystem-1_1.xml");
    }
```

So we say that this test is for the `threads` subsystem, and that it is implemented by `ThreadsExtension`. This is the same test framework as we use in Example subsystem#Testing the parsers, but we will only talk about the parts relevant to transformers here.

9.8.1 Testing a configuration that works

To test a configuration xxx

```
@Test
    public void testTransformerAS712() throws Exception {
        testTransformer_1_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }
    /**
     * Tests transformation of model from 1.1.0 version into 1.0.0 version.
     *
     * @throws Exception
     */
    private void testTransformer_1_0(ModelTestControllerVersion controllerVersion) throws
Exception {
        String subsystemXml = "threads-transform-1_0.xml";   //This has no expressions not
understood by 1.0
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0); //The old model version
        //Use the non-runtime version of the extension which will happen on the HC
        KernelServicesBuilder builder =
createKernelServicesBuilder(AdditionalInitialization.MANAGEMENT)
                .setSubsystemXmlResource(subsystemXml);

        final PathAddress subsystemAddress =
PathAddress.pathAddress(PathElement.pathElement(SUBSYSTEM, mainSubsystemName));

        // Add legacy subsystems
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
                .addOperationValidationResolve("add",
subsystemAddress.append(PathElement.pathElement("thread-factory")))
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(modelVersion);
        Assert.assertNotNull(legacyServices);
        checkSubsystemModelTransformation(mainServices, modelVersion);
    }
```

What this test does is get the builder to configure the test controller using `threads-transform-1_0.xml`. This main builder works with the current subsystem version, and the jars in the WildFly checkout.

Next we configure a 'legacy' controller. This will run the version of the core libraries (e.g the `controller` module) as found in the targeted legacy version of JBoss AS/Wildfly), and the subsystem. We need to pass in that it is using the core AS version 7.1.2.Final (i.e. the `ModelTestControllerVersion.V7_1_2_FINAL` part) and that that version is ModelVersion 1.0.0. Next we have some `addMavenResourceURL()` calls passing in the Maven GAVs of the old version of the subsystem and any dependencies it has needed to boot up. Normally, specifying just the Maven GAV of the old version of the subsystem is enough, but that depends on your subsystem. In this case the old subsystem GAV is enough. When booting up the legacy controller the framework uses the parsed operations from the main controller and transforms them using the 1.0.0 transformer in the threads subsystem. The `addOperationValidationResolve()` and `excludeFromParent()` calls are not normally necessary, see the javadoc for more examples.

The call to `KernelServicesBuilder.build()` will build both the main controller and the legacy controller. As part of that it also boots up a second copy of the main controller using the transformed operations to make sure that the 'old' ops to boot our subsystem will still work on the current controller, which is important for backwards compatibility of CLI scripts. To tweak how that is done if you see failures there, see `LegacyKernelServicesInitializer.skipReverseControllerCheck()` and `LegacyKernelServicesInitializer.configureReverseControllerCheck()`. The `LegacyKernelServicesInitializer` is what gets returned by `KernelServicesBuilder.createLegacyKernelServicesBuilder()`.

Finally we call `checkSubsystemModelTransformation()` which reads the full legacy subsystem model. The legacy subsystem model will have been built up from the transformed boot operations from the parsed xml. The operations get transformed by the operation transformers. Then it takes the model of the current subsystem and transforms that using the resource transformers. Then it compares the two models, which should be the same. In some rare cases it is not possible to get those two models exactly the same, so there is a version of this method that takes a `ModelFixer` to make adjustments. The `checkSubsystemModelTransformation()` method also makes sure that the legacy model is valid according to the legacy subsystem's resource definition.

The legacy subsystem resource definitions are read on demand from the legacy controller when the tests run. In some older versions of subsystems (before we converted everything to use ResourceDefinition, and DescriptionProvider implementations were coded by hand) there were occasional problems with the resource definitions and they needed to be touched up. In this case you can generate a new one, touch it up and store the result in a file in the test resources under `/same/package/as/the/test/class/{{subsystem-name-model-version`. This will then prefer the file read from the file system to the one read at runtime. To generate the .dmr file, you need to generate it by adding a temporary test (make sure that you adjust `controllerVersion` and `modelVersion` to what you want to generate):

```
@Test
    public void deleteMeWhenDone() throws Exception {
        ModelTestControllerVersion controllerVersion = ModelTestControllerVersion.V7_1_2_FINAL;
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0);
        KernelServicesBuilder builder = createKernelServicesBuilder(null);

        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
            .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion());
        KernelServices services = builder.build();

        generateLegacySubsystemResourceRegistrationDmr(services, modelVersion);
    }
```

Now run the test and delete it. The legacy .dmr file should be in `target/test-classes/org/jboss/as/subsystem/test/<your-subsystem-name>-<your-vers` . Copy this .dmr file to the correct location in your project's test resources.

9.8.2 Testing a configuration that does not work

The `threads` subsystem (like several others) did not support the use of expression values in the version that came with JBoss AS 7.1.2.Final. So we have a test that attempts to use expressions, and then fixes each resource and attribute where expressions were not allowed.

```
@Test
    public void testRejectExpressionsAS712() throws Exception {
        testRejectExpressions_1_0_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }

    private void testRejectExpressions_1_0_0(ModelTestControllerVersion controllerVersion)
throws Exception {
        // create builder for current subsystem version
        KernelServicesBuilder builder =
createKernelServicesBuilder(createAdditionalInitialization());

        // create builder for legacy subsystem version
        ModelVersion version_1_0_0 = ModelVersion.create(1, 0, 0);
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, version_1_0_0)
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(version_1_0_0);

        Assert.assertNotNull(legacyServices);
        Assert.assertTrue("main services did not boot", mainServices.isSuccessfulBoot());
        Assert.assertTrue(legacyServices.isSuccessfulBoot());

        List<ModelNode> xmlOps = builder.parseXmlResource("expressions.xml");

        ModelTestUtils.checkFailedTransformedBootOperations(mainServices, version_1_0_0, xmlOps,
getConfig());
    }
```

Again we boot up a current and a legacy controller. However, note in this case that they are both empty, no xml was parsed on boot so there are no operations to boot up the model. Instead once the controllers have been booted, we call `KernelServicesBuilder.parseXmlResource()` which gets the operations from `expressions.xml`. `expressions.xml` uses expressions in all the places they were not allowed in 7.1.2.Final. For each resource `ModelTestUtils.checkFailedTransformedBootOperations()` will check that the `add` operation gets rejected, and then correct one attribute at a time until the resource has been totally corrected. Once the `add` operation is totally correct, it will check that the add operation no longer is rejected. The configuration for this is the `FailedOperationTransformationConfig` returned by the `getConfig()` method:

```
private FailedOperationTransformationConfig getConfig() {
        PathAddress subsystemAddress = PathAddress.pathAddress(ThreadsExtension.SUBSYSTEM_PATH);
        FailedOperationTransformationConfig.RejectExpressionsConfig allowedAndKeepalive =
                new
FailedOperationTransformationConfig.RejectExpressionsConfig(PoolAttributeDefinitions.ALLOW_CORE_TI
PoolAttributeDefinitions.KEEPALIVE_TIME);
...
        return new FailedOperationTransformationConfig()

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BLOCKING_BOUN
allowedAndKeepalive)

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BOUNDED_QUEUE_
allowedAndKeepalive)
    }
```

So what this means is that we expect the `allow-core-timeout` and `keepalive-time` attributes for the `blocking-bounded-queue-thread-pool=*` and `bounded-queue-thread-pool=*` add operations to use expressions in the parsed xml. We then expect them to fail since there should be transformers in place to reject expressions, and correct them one at a time until the add operation should pass. As well as doing the add operations the `ModelTestUtils.checkFailedTransformedBootOperations()` method will also try calling `write-attribute` for each attribute, correcting as it goes along. As well as allowing you to test rejection of expressions `FailedOperationTransformationConfig` also has some helper classes to help testing rejection of other scenarios.

9.9 Common transformation use-cases

Most transformations are quite similar, so this section covers some of the actual transformation patterns found in the WildFly codebase. We will look at the output of CompareModelVersionsUtil, and see what can be done to transform for the older slave HCs. The examples come from the WildFly codebase but are stripped down to focus solely on the use-case being explained in an attempt to keep things as clear/simple as possible.

9.9.1 Child resource type does not exist in legacy model

Looking at the model comparison between WildFly and JBoss AS 7.2.0, there is a change to the `remoting` subsystem. The relevant part of the output is:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
```

So our current model has added a child type called `http-connector` which was not there in 7.2.0. This is configurable, and adds new behaviour, so it can not be part of a configuration sent across to a legacy slave running version 1.2.0. So we add the following to `RemotingExtension` to reject all instances of that child type against ModelVersion 1.2.0.

```
@Override
    public void initialize(ExtensionContext context) {
        ....
        if (context.isRegisterTransformers()) {
            registerTransformers_1_1(registration);
            registerTransformers_1_2(registration);
        }
    }

    private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.rejectChildResource(HttpConnectorResource.PATH);

        return builder.build();
    }
```

Since this child resource type also does not exist in ModelVersion 1.1.0 we need to reject it there as well using a similar mechanism.

9.9.2 Attribute does not exist in the legacy subsystem

Default value of the attribute is the same as legacy implied behaviour

This example also comes from the `remoting` subsystem, and is probably the most common type of transformation. The comparison tells us that there is now an attribute under `/subsystem=remoting/remote-outbound-connection=*` called `protocol` which did not exist in the older version:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
....
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
```

This difference also affects the `add` operation. Looking at the current model the valid values for the `protocol` attribute are `remote`, `http-remoting` and `https-remoting`. The last two are new protocols introduced in WildFly 8, meaning that the *implied behaviour* in JBoss 7.2.0 and earlier is the `remote` protocol. Since this attribute does not exist in the legacy model we want to discard this attribute if it is `undefined` or if it has the value `remote`, both of which are in line with what the legacy slave HC is hardwired to use. Also we want to reject it if it has a value different from `remote`. So what we need to do when registering transformers against ModelVersion 1.2.0 to handle this attribute:

```
private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();

protocolTransform(builder.addChildResource(RemoteOutboundConnectionResourceDefinition.ADDRESS)
                .getAttributeBuilder());
        return builder.build();
    }

    private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(Protocol.REMOTE.toString())), RemoteOutboundConnectionResourceDefinition.PROTOCOL)
                .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
        return builder;
    }
```

So the first thing to happens is that we register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which discards the attribute if it is either `undefined` (the default value in the current model is `remote`), or `defined` and has the value `remote`. Remembering that the `discard` phase always happens before the `reject` phase, the reject checker checks that the `protocol` attribute is defined, and rejects it if it is. The only reason it would be `defined` in the reject check, is if it was not discarded by the discard check. Hopefully this example shows that the discard and reject checkers often work in pairs.

An alternative way to write the `protocolTransform()` method would be:

```
private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DefaultDiscardAttributeChecker() {
                    @Override
                    protected boolean isValueDiscardable(final PathAddress address, final String
attributeName, final ModelNode attributeValue, final TransformationCon
                        return !attributeValue.isDefined() ||
attributeValue.asString().equals(Protocol.REMOTE.toString());
                    }
                }, RemoteOutboundConnectionResourceDefinition.PROTOCOL)
         .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
         return builder;
```

The reject check remains the same, but we have implemented the discard check by using `DiscardAttributeChecker.DefaultDiscardAttributeChecker` instead. However, the effect of the discard check is exactly the same as when we used `DiscardAttributeChecker.DiscardAttributeValueChecker`.

Default value of the attribute is different from legacy implied behaviour

We touched on this in the weld subsystem example we used earlier in this guide, but let's take a more thorough look. Our comparison tells us that we have two new attributes `require-bean-descriptor` and `non-portable-mode`:

```
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]
```

Now when we look at this we see that the default value for both of the attributes in the current model is `false`, which allows us more flexible behaviour introduced in CDI 1.1 (which was introduced with this version of the subsystem). The old model does not have these attributes, and implements CDI 1.0, which under the hood (using our weld subsystem expertise knowledge) implies the values `true` for both of these. So our transformer must reject anything that is not `true` for these attributes. Let us look at the transformer registered by the WeldExtension:

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

This looks a bit more scary than the previous transformer we have seen, but isn't actually too bad. The first thing we do is register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which will discard the attribute if it has the value `true`. It will not discard if it is `undefined` since that defaults to `false`. This is registered for both attributes.

If the attributes had the value `true` they will get discarded we will not hit the reject checker since discarded attributes never get checked for rejection. If on the other hand they were an expression (since we are interested in the actual value, but cannot evaluate what value an expression will resolve to on the target from the DC running the transformers), `false`, or `undefined` (which will then default to `false`) they will not get discarded and will need to be rejected. So our `RejectAttributeChecker.DefaultRejectAttributeChecker.rejectAttribute()` method will return `true` (i.e. reject) if the attribute value is `undefined` (since that defaults to `false`) or if it is defined and 'not equal to `true`'. It is better to check for 'not equal to `true`' than to check for 'equal to `false`' since if an expression was used we still want to reject, and only the 'not equal to `true`' check would actually kick in in that case.

The other thing we need in our `DiscardAttributeChecker.DiscardAttributeValueChecker` is to override the `getRejectionLogMessage()` method to get the message to be displayed when rejecting the transformation. In this case it says something along the lines "These attributes must be 'true' for use with CDI 1.0 '%s'", with the names of the attributes having been rejected substituting the `%s`.

9.9.3 Attribute has a different default value

– TODO

9.9.4 Attribute has a different type

Here the example comes from the `capacity` parameter some way into the `modcluster` subsystem, and the legacy version is AS 7.1.2.Final. There are quite a few differences, so I am only showing the ones relevant for this example:

```
====== Resource root address: ["subsystem" => "modcluster"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
...
--- Problems for relative address to root ["mod-cluster-config" =>
"configuration","dynamic-load-provider" => "configuration","custom-load-m
etric" => "*"]:
Different 'type' for attribute 'capacity'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for attribute 'capacity'. Current: true; legacy: false
...
Different 'type' for parameter 'capacity' of operation 'add'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for parameter 'capacity' of operation 'add'. Current: true;
legacy: false
```

So as we can see expressions are not allowed for the `capacity` attribute, and the current type is `double` while the legacy subsystem is `int`. So this means that if the value is for example `2.0` we can convert this to `2`, but `2.5` cannot be converted. The way this is solved in the ModClusterExtension is to register the following some other attributes are registered here, but hopefully it is clear anyway:

```
dynamicLoadProvider.addChildResource(LOAD_METRIC_PATH)
                    .getAttributeBuilder()
                        .addRejectCheck(RejectAttributeChecker.SIMPLE_EXPRESSIONS, TYPE, WEIGHT,
CAPACITY, PROPERTY)
                        .addRejectCheck(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        .setValueConverter(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        ...
                        .end();
```

So we register that we should reject expressions, and we also register the `CapacityCheckerAndConverter` for `capacity`. `CapacityCheckerAndConverter` extends the convenience class `DefaultCheckersAndConverter` which implements the `DiscardAttributeChecker`, `RejectAttributeChecker`, and `AttributeConverter` interfaces. We have seen `DiscardAttributeChecker` and `RejectAttributeChecker` in previous examples. Since we now need to convert a value we need an instance of `AttributeConverter`.

```
static class CapacityCheckerAndConverter extends DefaultCheckersAndConverter {

        static final CapacityCheckerAndConverter INSTANCE = new CapacityCheckerAndConverter();
```

We should not discard so `isValueDiscardable()` from `DiscardAttributeChecker` always returns `false`:

```
@Override
        protected boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context) {
            //Not used for discard
            return false;
        }

        @Override
        public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
            return
ModClusterMessages.MESSAGES.capacityIsExpressionOrGreaterThanIntegerMaxValue(attributes.get(CAPACI
}
```

Now we check to see if we can convert the attribute to an `int` and reject if not. Note that if it is an expression, we have no idea what its value will resolve to on the target host, so we need to reject it. Then we try to change it into an `int`, and reject if that was not possible:

```
@Override
        protected boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            if (checkForExpression(attributeValue)
                    || (attributeValue.isDefined() &&
!isIntegerValue(attributeValue.asDouble()))) {
                return true;
            }
            Long converted = convert(attributeValue);
            return (converted != null && (converted > Integer.MAX_VALUE || converted <
Integer.MIN_VALUE));
        }
```

And then finally we do the conversion:

```
@Override
        protected void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            Long converted = convert(attributeValue);
            if (converted != null && converted <= Integer.MAX_VALUE && converted >=
Integer.MIN_VALUE) {
                attributeValue.set((int)converted.longValue());
            }
        }


        private Long convert(ModelNode attributeValue) {
            if (attributeValue.isDefined() && !checkForExpression(attributeValue)) {
                double raw = attributeValue.asDouble();
                if (isIntegerValue(raw)) {
                    return Math.round(raw);
                }
            }
            return null;
        }

        private boolean isIntegerValue(double raw) {
            return raw == Double.valueOf(Math.round(raw)).doubleValue();
        }

    }
```

10 Example subsystem

Our example subsystem will keep track of all deployments of certain types containing a special marker file, and expose operations to see how long these deployments have been deployed.

10.1 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

10.2 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true.`
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

10.3 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The SubsystemExtension.initialize() method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

10.3.1 Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

It's parameter is an implementation of the ResourceDefinition interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by SubsystemDefinition.INSTANCE.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The ManagementResourceRegistration obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside registerOperations in your ResourceDefinition or by providing it in constructor of your SimpleResourceDefinition just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by `3`.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define DescriptionProvider, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Flowing example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes DefaultResourceAddDescriptionProvider and DefaultResourceRemoveDescriptionProvider that do work for you. In case you use SimpleResourceDefinition even that part is hidden from you.

resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);

For other operation handlers that are not add/remove you can use DefaultOperationDescriptionProvider that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "add-mime" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

10.3.2 Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it TypeDefinition and for ease of use let it extend SimpleResourceDefinition instead just implement ResourceDefinition.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define SimpleAttributeDefinition named TICK, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation validateAndSet on AttributeDefinition that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

10.4 Parsing and marshalling of the subsystem xml

JBoss AS 7 uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by SUBSYSTEM_PATH this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by TYPE_PATH) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

10.4.1 Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is add for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is add for the address `/subsystem=tracker`, and that 12345 was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical
ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command
`/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

10.5 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

10.5.1 Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

10.6 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly 8. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$AS7/domain/configuration/domain.xml`.

Now start up WildFly 8 by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly 8 console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

10.7 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

10.7.1 What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

10.7.2 How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type ModelType.EXPRESSION.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation ModelNode, and if it isn't already ModelType.EXPRESSION, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type ModelType.EXPRESSION. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

10.8 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

10.8.1 Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

10.9 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true.`
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

10.10 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

10.11 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The SubsystemExtension.initialize() method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

10.11.1 Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

It's parameter is an implementation of the ResourceDefinition interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by SubsystemDefinition.INSTANCE.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The ManagementResourceRegistration obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside registerOperations in your ResourceDefinition or by providing it in constructor of your SimpleResourceDefinition just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by `3`.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define DescriptionProvider, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Flowing example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes DefaultResourceAddDescriptionProvider and DefaultResourceRemoveDescriptionProvider that do work for you. In case you use SimpleResourceDefinition even that part is hidden from you.

resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);

For other operation handlers that are not add/remove you can use DefaultOperationDescriptionProvider that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "add-mime" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

10.11.2 Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it TypeDefinition and for ease of use let it extend SimpleResourceDefinition instead just implement ResourceDefinition.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define SimpleAttributeDefinition named TICK, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation validateAndSet on AttributeDefinition that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

⚠ When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

10.12 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

10.12.1 What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

10.12.2 How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type ModelType.EXPRESSION.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation ModelNode, and if it isn't already ModelType.EXPRESSION, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type ModelType.EXPRESSION. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

10.13 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly 8. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$AS7/domain/configuration/domain.xml`.

Now start up WildFly 8 by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly 8 console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

10.14 Parsing and marshalling of the subsystem xml

JBoss AS 7 uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by SUBSYSTEM_PATH this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by TYPE_PATH) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

10.14.1 Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is `add` for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is `add` for the address `/subsystem=tracker`, and that `12345` was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command
`/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

11 Key Interfaces and Classes Relevant to Extension Developers

In the first major section of this guide, we provided an example of how to implement an extension to the AS. The emphasis there was learning by doing. In this section, we'll focus a bit more on the major WildFly interfaces and classes that most are relevant to extension developers. The best way to learn about these interfaces and classes in detail is to look at their javadoc. What we'll try to do here is provide a brief introduction of the key items and how they relate to each other.

Before digging into this section, readers are encouraged to read the "Core Management Concepts" section of the Admin Guide.

11.1 Extension Interface

The `org.jboss.as.controller.Extension` interface is the hook by which your extension to the core AS is able to integrate with the AS. During boot of the AS, when the `<extension>` element in the AS's xml configuration file naming your extension is parsed, the JBoss Modules module named in the element's name attribute is loaded. The standard JDK `java.lang.ServiceLoader` mechanism is then used to load your module's implementation of this interface.

The function of an `Extension` implementation is to register with the core AS the management API, xml parsers and xml marshallers associated with the extension module's subsystems. An `Extension` can register multiple subsystems, although the usual practice is to register just one per extension.

Once the `Extension` is loaded, the core AS will make two invocations upon it:

- `void initializeParsers(ExtensionParsingContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to initialize the XML parsers for this extension's subsystems and register them with the given `ExtensionParsingContext`. The parser's job when it is later called is to create `org.jboss.dmr.ModelNode` objects representing WildFly management API operations needed make the AS's running configuration match what is described in the xml. Those management operation {{ModelNode}}s are added to a list passed in to the parser.

A parser for each version of the xml schema used by a subsystem should be registered. A well behaved subsystem should be able to parse any version of its schema that it has ever published in a final release.

- `void initialize(ExtensionContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to register with the core AS the management API for its subsystems, and to register the object that is capable of marshalling the subsystem's in-memory configuration back to XML. Only one XML marshaller is registered per subsystem, even though multiple XML parsers can be registered. The subsystem should always write documents that conform to the latest version of its XML schema.

The registration of a subsystem's management API is done via the `ManagementResourceRegistration` interface. Before discussing that interface in detail, let's describe how it (and the related `Resource` interface) relate to the notion of managed resources in the AS.

11.2 WildFly Managed Resources

Each subsystem is responsible for managing one or more management resources. The conceptual characteristics of a management resource are covered in some detail in the Admin Guide; here we'll just summarize the main points. A management resource has

- An ***address*** consisting of a list of key/value pairs that uniquely identifies a resource
- Zero or more ***attributes***, the value of which is some sort of `org.jboss.dmr.ModelNode`
- Zero or more supported ***operations***. An operation has a string name and zero or more parameters, each of which is a key/value pair where the key is a string naming the parameter and the value is some sort of `ModelNode`
- Zero or ***children***, each of which in turn is a managed resource

The implementation of a managed resource is somewhat analogous to the implementation of a Java object. A managed resource will have a "type", which encapsulates API information about that resource and logic used to implement that API. And then there are actual instances of the resource, which primarily store data representing the current state of a particular resource. This is somewhat analogous to the "class" and "object" notions in Java.

A managed resource's type is encapsulated by the `org.jboss.as.controller.registry.ManagementResourceRegistration` the core AS creates when the type is registered. The data for a particular instance is encapsulated in an implementation of the `org.jboss.as.controller.registry.Resource` interface.

11.3 ManagementResourceRegistration Interface

TODO

11.4 ResourceDefinition Interface

TODO

Most commonly used implementation: `SimpleResourceDefinition`

11.4.1 ResourceDescriptionResolver

TODO

Most commonly used implementation: `StandardResourceDescriptionResolver`

11.5 AttributeDefinition Interface

TODO

Most commmonly used implementation: `SimpleAttributeDefinition`. Use `SimpleAttributeDefinitionBuilder` to build.

11.6 OperationDefinition and OperationStepHandler Interfaces

TODO

11.7 Operation Execution and the OperationContext

TODO

11.8 Resource Interface

TODO

11.9 DeploymentUnitProcessor Interface

TODO

11.10 Useful classes for implementing OperationStepHandler

TODO

12 Transformers

This guide is work in progress

12.1 What are transformers

Transformers are mechanism used in domain mode, to support scenarios where you have host controllers (HC) joined to domain controller (DC) running different version of subsystem (server).

When HC registers itself against DC and it is running different version of some subsystem that is present on DC, DC performs transformation of model (ModelTransformer) and sends it back to HC as configuration it has too run.

Also when operations are performed on DC, for example call ADD for some resource, DC also transforms that operation (OperationTransformer) into target version of HC.

Transformers are primarily used for upgrade migration paths. General idea behind this migration period is that there are numerous HC and one DC deployed. First, DC is upgraded. Afterwards, HC are upgraded in steps as needed without breaking configuration.

12.1.1 When are they invoked

When Host controller registers to Domain Controller. HC does not know about conversions, they are done on DC as follows:

1. HC -> connect -> DC
2. HC <- list of subsystems to sent supported versions for <- DC
3. HC -> list of supported versions for requested subsystems -> DC
4. **transformations happen on DC**
5. HC <- complete model in supported model versions <- DC

12.1.2 When should they be implemented

When model in your subsystem has changed in incompatible way.

Rule of thumb would be if current model is applied to previous version of subsystem model would it work? If not, model version must be increased and transformers implemented.

12.1.3 Model Transformer

Model transformer, works on top of current model and transforms its model to target version.

12.1.4 Operation Transformer

Operation transformer works on top of operations instead of model.

12.1.5 Testing transformers

There is testing framework in place for testing proper behavior of transformers

13 WildFly 9 JNDI Implementation

13.1 Introduction

This page proposes a reworked WildFly JNDI implementation, and new/updated APIs for WildFly subsystem and EE deployment processors developers to bind new resources easier.

To support discussion in the community, the content includes a big focus on comparing WildFly 8 JNDI implementation with the new proposal, and should later evolve to the prime guide for WildFly developers needing to interact with JNDI at subsystem level.

13.2 Architecture

WildFly relies on MSC to provide the data source for the JNDI tree. Each resource bound in JNDI is stored in a MSC service (BinderService), and such services are installed as children of subsystem/deployment services, for an automatically unbound as consequence of uninstall of the parent services.

Since there is the need to know what entries are bound, and MSC does not provides that, there is also the (ServiceBased)NamingStore concept, which internally manage the set of service names bound. There are multiple naming stores in every WildFly instance, serving different JNDI namespaces:

- java:comp - the standard EE namespace for entries scoped to a specific component, such as an EJB
- java:module - the standard EE namespace for entries scoped to specific module, such as an EJB jar, and shared by all components in it
- java:app - the standard EE namespace for entries scoped to a specific application, i.e. EAR, and shared by all modules in it
- java:global - the standard EE namespace for entries shared by all deployments
- java:jboss - a proprietary namespace "global" namespace
- java:jboss/exported - a proprietary "global" namespace which entries are exposed to remote JNDI
- java: - any entries not in the other namespaces

One particular implementation choice, to save resources, is that JNDI contexts by default are not bound, the naming stores will search for any entry bound with a name that is a child of the context name, if found then its assumed the context exists.

The reworked implementation introduces shared/global java:comp, java:module and java:app namespaces. Any entry bound on these will automatically be available to every EE deployment scoped instance of these namespaces, what should result in a significant reduction of binder services, and also of EE deployment processors. Also, the Naming subsystem may now configure bind on these shared contexts, and these contexts will be available when there is no EE component in the invocation, which means that entries such as java:comp/DefaultDatasource will always be available.

13.3 Binding APIs

WildFly Naming subsystem exposes high level APIs to bind new JNDI resources, there is no need to deal with the low level BinderService type anymore.

13.3.1 Subsystem

At the lowest level a JNDI entry is bound by installing a BinderService to a ServiceTarget:

```
    /**
      * Binds a new entry to JNDI.
      * @param serviceTarget the binder service's target
      * @param name the new JNDI entry's name
      * @param value the new JNDI entry's value
      */
     private ServiceController<?> bind(ServiceTarget serviceTarget, String name, Object value) {

 // the bind info object provides MSC service names to use when creating the binder service
         final ContextNames.BindInfo bindInfo = ContextNames.bindInfoFor(name);
         final BinderService binderService = new BinderService(bindInfo.getBindName());

 // the entry's value is provided by a managed reference factory,
         // since the value may need to be obtained on lookup (e.g. EJB reference)
         final ManagedReferenceFactory managedReferenceFactory = new
ImmediateManagedReferenceFactory(value);

 return serviceTarget
                 // add binder service to specified target
                 .addService(bindInfo.getBinderServiceName(), binderService)
                 // when started the service will be injected with the factory
                 .addInjection(binderService.getManagedObjectInjector(), managedReferenceFactory)
                 // the binder service depends on the related naming store service,
                 // and on start/stop will add/remove its service name
                 .addDependency(bindInfo.getParentContextServiceName(),
                         ServiceBasedNamingStore.class,
                         binderService.getNamingStoreInjector())
                 .install();
     }
```

But the example above is the simplest usage possible, it may become quite complicated if the entry's value is not immediately available, for instance it is a value in another MSC service, or is a value in another JNDI entry. It's also quite easy to introduce bugs when working with the service names, or incorrectly assume that other MSC functionality, such as alias names, may be used.

Using the new high level API, it's as simple as:

```
// bind an immediate value
ContextNames.bindInfoFor("java:comp/ORB").bind(serviceTarget, this.orb);


// bind value from another JNDI entry (an alias/linkref)
ContextNames.bindInfoFor(“java:global/x").bind(serviceTarget, new JndiName(“java:jboss/x"));


// bind value obtained from a MSC service
ContextNames.bindInfoFor(“java:global/z").bind(serviceTarget, serviceName);
```

If there is the need to access the binder's service builder, perhaps to add a service verification handler or simply not install the binder service right away:

```
ContextNames.bindInfoFor("java:comp/ORB").builder(serviceTarget, verificationHandler,
ServiceController.Mode.ON_DEMAND).installService(this.orb);
```

13.3.2 EE Deployment

With respect to EE deployments, the subsystem API should not be used, since bindings may need to be discarded/overridden, thus a EE deployment processor should add a new binding in the form of a BindingConfiguration, to the EeModuleDescription or ComponentDescription, depending if the bind is specific to a component or not. An example of a deployment processor adding a binding:

```
public class ModuleNameBindingProcessor implements DeploymentUnitProcessor {

    // jndi name objects are immutable
    private static final JndiName JNDI_NAME_java_module_ModuleName = new
JndiName("java:module/ModuleName");

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

 final DeploymentUnit deploymentUnit = phaseContext.getDeploymentUnit();
        // skip deployment unit if it's the top level EAR
        if (DeploymentTypeMarker.isType(DeploymentType.EAR, deploymentUnit)) {
            return;
        }

 // the module's description is in the DUs attachments
        final EEModuleDescription moduleDescription = deploymentUnit
                .getAttachment(org.jboss.as.ee.component.Attachments.EE_MODULE_DESCRIPTION);
        if (moduleDescription == null) {
            return;
        }

 // add the java:module/ModuleName binding
        // the value's injection source for an immediate available value
        final InjectionSource injectionSource = new
ImmediateInjectionSource(moduleDescription.getModuleName());

 // add the binding configuration to the module's description bindings configurations
        moduleDescription.getBindingConfigurations()
                .addDeploymentBinding(new BindingConfiguration(JNDI_NAME_java_module_ModuleName,
injectionSource));
    }

    //...
}
```

When adding the binding configuration use:

- addDeploymentBinding() for a binding that may not be overriden, such as the ones found in xml descriptors
- addPlatformBinding() for a binding which may be overriden by a deployment descriptor bind or annotation, for instance java:comp/DefaultDatasource

A deployment processor may now also add a binding configuration to all components in a module:

```
moduleDescription.getBindingConfigurations().addPlatformBindingToAllComponents(bindingConfiguratio
```

⚠ In the reworked implementation there is now no need to behave differently considering the deployment type, for instance if deployment is a WAR or app client, the Module/Component BindingConfigurations objects handle all of that. The processor should simply go for the 3 use cases: module binding, component binding or binding shared by all components.

⚠ All deployment binding configurations MUST be added before INSTALL phase, this is needed because on such phase, when the bindings are actually done, there must be a final set of deployment binding names known, such information is need to understand if a resource injection targets entries in the global or scoped EE namespaces.

Most cases for adding bindings to EE deployments are in the context of a processor deploying a XML descriptor, or scanning deployment classes for annotations, and there abstract types, such as the AbstractDeploymentDescriptorBindingsProcessor, which simplifies greatly the processor code for such use cases.

One particular use case is the parsing of EE Resource Definitions, and the reworked implementation provides high level abstract deployment processors for both XML descriptor and annotations, an example for each:

```
/**
 * Deployment processor responsible for processing administered-object deployment descriptor
elements
 *
 * @author Eduardo Martins
 */
public class AdministeredObjectDefinitionDescriptorProcessor extends
ResourceDefinitionDescriptorProcessor {

    @Override
    protected void processEnvironment(RemoteEnvironment environment,
ResourceDefinitionInjectionSources injectionSources) throws DeploymentUnitProcessingException {
        final AdministeredObjectsMetaData metaDatas = environment.getAdministeredObjects();
        if (metaDatas != null) {
            for(AdministeredObjectMetaData metaData : metaDatas) {

injectionSources.addResourceDefinitionInjectionSource(getResourceDefinitionInjectionSource(metaDat
}
        }
    }

    private ResourceDefinitionInjectionSource getResourceDefinitionInjectionSource(final
AdministeredObjectMetaData metaData) {
        final String name = metaData.getName();
        final String className = metaData.getClassName();
        final String resourceAdapter = metaData.getResourceAdapter();
        final AdministeredObjectDefinitionInjectionSource resourceDefinitionInjectionSource =
new AdministeredObjectDefinitionInjectionSource(name, className, resourceAdapter);
        resourceDefinitionInjectionSource.setInterface(metaData.getInterfaceName());
        if (metaData.getDescriptions() != null) {

resourceDefinitionInjectionSource.setDescription(metaData.getDescriptions().toString());
        }
        resourceDefinitionInjectionSource.addProperties(metaData.getProperties());
        return resourceDefinitionInjectionSource;
    }

}
```

and

```
/**
 * Deployment processor responsible for processing {@link
javax.resource.AdministeredObjectDefinition} and {@link
javax.resource.AdministeredObjectDefinitions}.
 *
 * @author Jesper Pedersen
 * @author Eduardo Martins
 */
public class AdministeredObjectDefinitionAnnotationProcessor extends
ResourceDefinitionAnnotationProcessor {

    private static final DotName ANNOTATION_NAME =
DotName.createSimple(AdministeredObjectDefinition.class.getName());
    private static final DotName COLLECTION_ANNOTATION_NAME =
DotName.createSimple(AdministeredObjectDefinitions.class.getName());

    @Override
    protected DotName getAnnotationDotName() {
        return ANNOTATION_NAME;
    }

    @Override
    protected DotName getAnnotationCollectionDotName() {
        return COLLECTION_ANNOTATION_NAME;
    }

    @Override
    protected ResourceDefinitionInjectionSource processAnnotation(AnnotationInstance
annotationInstance) throws DeploymentUnitProcessingException {
        final String name = AnnotationElement.asRequiredString(annotationInstance,
AnnotationElement.NAME);
        final String className = AnnotationElement.asRequiredString(annotationInstance,
"className");
        final String ra = AnnotationElement.asRequiredString(annotationInstance,
"resourceAdapter");
        final AdministeredObjectDefinitionInjectionSource
directAdministeredObjectInjectionSource =
                new AdministeredObjectDefinitionInjectionSource(name, className, ra);

directAdministeredObjectInjectionSource.setDescription(AnnotationElement.asOptionalString(annotati
AdministeredObjectDefinitionInjectionSource.DESCRIPTION));

directAdministeredObjectInjectionSource.setInterface(AnnotationElement.asOptionalString(annotation
AdministeredObjectDefinitionInjectionSource.INTERFACE));

directAdministeredObjectInjectionSource.addProperties(AnnotationElement.asOptionalStringArray(anno
AdministeredObjectDefinitionInjectionSource.PROPERTIES));
        return directAdministeredObjectInjectionSource;
    }

}
```

The abstract processors with respect to Resource Definitions are already submitted through WFLY-3292's PR.

13.4 Resource Ref Processing

TODO for now no changes on this in the reworked WildFly Naming.

14 Working with WildFly Capabilities

An extension to WildFly will likely want to make use of services provided by the WildFly kernel, may want to make use of services provided by other subsystems, and may wish to make functionality available to other extensions. Each of these cases involves integration between different parts of the system. In releases prior to WildFly 10, this kind of integration was done on an ad-hoc basis, resulting in overly tight coupling between different parts of the system and overly weak integration contracts. For example, a service installed by subsystem A might depend on a service installed by subsystem B, and to record that dependency A's authors copy a ServiceName from B's code, or even refer to a constant or static method from B's code. The result is B's code cannot evolve without risking breaking A. And the authors of B may not even intend for other subsystems to use its services. There is no proper integration contract between the two subsystems.

Beginning with WildFly Core 2 and WildFly 10 the WildFly kernel's management layer provides a mechanism for allowing different parts of the system to integrate with each other in a loosely coupled manner. This is done via WildFly Capabilities. Use of capabilities provides the following benefits:

1. A standard way for system components to define integration contracts for their use by other system components.
2. A standard way for system components to access integration contracts provided by other system components.
3. A mechanism for configuration model referential integrity checking, such that if one component's configuration has an attribute that refers to an other component (e.g. a `socket-binding` attribute in a subsystem that opens a socket referring to that socket's configuration), the validity of that reference can be checked when validating the configuration model.

14.1 Capabilities

A capability is a piece of functionality used in a WildFly Core based process that is exposed via the WildFly Core management layer. Capabilities may depend on other capabilities, and this interaction between capabilities is mediated by the WildFly Core management layer.

Some capabilities are automatically part of a WildFly Core based process, but in most cases the configuration provided by the end user (i.e. in standalone.xml, domain.xml and host.xml) determines what capabilities are present at runtime. It is the responsibility of the handlers for management operations to register capabilities and to register any requirements those capabilities may have for the presence of other capabilities. This registration is done during the MODEL stage of operation execution

A capability has the following basic characteristics:

1. It has a name.
2. It may install an MSC service that can be depended upon by services installed by other capabilities. If it does, it provides a mechanism for discovering the name of that service.
3. It may expose some other API not based on service dependencies allowing other capabilities to integrate with it at runtime.
4. It may depend on, or **require** other capabilities.

During boot of the process, and thereafter whenever a management operation makes a change to the process' configuration, at the end of the MODEL stage of operation execution the kernel management layer will validate that all capabilities required by other capabilities are present, and will fail any management operation step that introduced an unresolvable requirement. This will be done before execution of the management operation proceeds to the RUNTIME stage, where interaction with the process' MSC Service Container is done. As a result, in the RUNTIME stage the handler for an operation can safely assume that the runtime services provided by a capability for which it has registered a requirement are available.

14.1.1 Comparison to other concepts

Capabilities vs modules

A JBoss Modules module is the means of making resources available to the classloading system of a WildFly Core based process. To make a capability available, you must package its resources in one or more modules and make them available to the classloading system. But a module is not a capability in and of itself, and simply copying a module to a WildFly installation does not mean a capability is available. Modules can include resources completely unrelated to management capabilities.

Capabilities vs Extensions

An extension is the means by which the WildFly Core management layer is made aware of manageable functionality that is not part of the WildFly Core kernel. The extension registers with the kernel new management resource types and handlers for operations on those resources. One of the things a handler can do is register or unregister a capability and its requirements. An extension may register a single capability, multiple capabilities, or possibly none at all. Further, not all capabilities are registered by extensions; the WildFly Core kernel itself may register a number of different capabilities.

14.1.2 Capability Names

Capability names are simple strings, with the dot character serving as a separator to allow namespacing.

The 'org.wildfly' namespace is reserved for projects associated with the WildFly organization on github (https://github.com/wildfly).

14.1.3 Statically vs Dynamically Named Capabilities

The full name of a capability is either statically known, or it may include a statically known base element and then a dynamic element. The dynamic part of the name is determined at runtime based on the address of the management resource that registers the capability. For example, the management resource at the address '/socket-binding-group=standard-sockets/socket-binding=web' will register a dynamically named capability named 'org.wildlfy.network.socket-binding.web'. The 'org.wildlfy.network.socket-binding' portion is the static part of the name.

All dynamically named capabilities that have the same static portion of their name should provide a consistent feature set and set of requirements.

14.1.4 Service provided by a capability

Typically a capability functions by registering a service with the WildFly process' MSC ServiceContainer, and then dependent capabilities depend on that service. The WildFly Core management layer orchestrates registration of those services and service dependencies by providing a means to discover service names.

14.1.5 Custom integration APIs provided by a capability

Instead of or in addition to providing MSC services, a capability may expose some other API to dependent capabilities. This API must be encapsulated in a single class (although that class can use other non-JRE classes as method parameters or return types).

14.1.6 Capability Requirements

A capability may rely on other capabilities in order to provide its functionality at runtime. The management operation handlers that register capabilities are also required to register their requirements.

There are three basic types of requirements a capability may have:

- Hard requirements. The required capability must always be present for the dependent capability to function.
- Optional requirements. Some aspect of the configuration of the dependent capability controls whether the depended on capability is actually necessary. So the requirement cannot be known until the running configuration is analyzed.
- Runtime-only requirements. The dependent capability will check for the presence of the depended upon capability at runtime, and if present it will utilize it, but if it is not present it will function properly without the capability. There is nothing in the dependent capability's configuration that controls whether the depended on capability must be present. Only capabilities that declare themselves as being suitable for use as a runtime-only requirement should be depended upon in this manner.

Hard and optional requirements may be for either statically named or dynamically named capabilities. Runtime-only requirements can only be for statically named capabilities, as such a requirement cannot be specified via configuration, and without configuration the dynamic part of the required capability name is unknown.

Supporting runtime-only requirements

Not all capabilities are usable as a runtime-only requirement.

Any dynamically named capability is not usable as a runtime-only requirement.

For a capability to support use as a runtime-only requirement, it must guarantee that a configuration change to a running process that removes the capability will not impact currently running capabilities that have a runtime-only requirement for it. This means:

- A capability that supports runtime-only usage must ensure that it never removes its runtime service except via a full process reload.
- A capability that exposes a custom integration API generally is not usable as a runtime-only requirement. If such a capability does support use as a runtime-only requirement, it must ensure that any functionality provided via its integration API remains available as long as a full process reload has not occurred.

14.2 Capability Contract

A capability provides a stable contract to users of the capability. The contract includes the following:

- The name of the capability (including whether it is dynamically named).
- Whether it installs an MSC Service, and if it does, the value type of the service. That value type then becomes a stable API users of the capability can rely upon.
- Whether it provides a custom integration API, and if it does, the type that represents that API. That type then becomes a stable API users of the capability can rely upon.
- Whether the capability supports use as a runtime-only requirement.

Developers can learn about available capabilities and the contracts they provide by reading the WildFly *capabilty registry*.

14.3 Capability Registry

The WildFly organization on github maintains a git repo where information about available capabilities is published.

https://github.com/wildfly/wildfly-capabilities

Developers can learn about available capabilities and the contracts they provide by reading the WildFly capabilty registry.

The README.md file at the root of that repo explains the how to find out information about the registry.

Developers of new capabilities are **strongly encouraged** to document and register their capability by submitting a pull request to the wildfly-capabilities github repo. This both allows others to learn about your capability and helps prevent capability name collisions. Capabilities that are used in the WildFly or WildFly Core code base itself **must** have a registry entry before the code referencing them will be merged.

External organizations that create capabilities should include an organization-specific namespace as part their capability names to avoid name collisions.

14.4 Using Capabilities

Now that all the background information is presented, here are some specifics about how to use WildFly capabilities in your code.

14.4.1 Basics of Using Your Own Capability

Creating your capability

A capability is an instance of the immutable `org.jboss.as.controller.capability.RuntimeCapability` class. A capability is usually registered by a resource, so the usual way to use one is to store it in constant in the resource's `ResourceDefinition`. Use a `RuntimeCapability.Builder` to create one.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    static final RuntimeCapability<Void> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo").build();


    . . .
}
```

That creates a statically named capability named `com.example.foo`.

If the capability is dynamically named, add the `dynamic` parameter to state this:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", true).build();
```

Most capabilities install a service that requiring capabilities can depend on. If your capability does this, you need to declare the service's *value type* (the type of the object returned by `org.jboss.msc.Service.getValue()`). For example, if FOO_CAPABILITY provides a `Service<javax.sql.DataSource>`:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", DataSource.class).build();
```

For a dynamic capability:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
           RuntimeCapability.Builder.of("com.example.foo", true, DataSource.class).build();
```

If the capability provides a custom integration API, you need to instantiate an instance of that API:

```
public class JTSCapability {

    static final JTSCapability INSTANCE = new JTSCapability();

    private JTSCapability() {}

    /**
     * Gets the names of the {@link org.omg.PortableInterceptor.ORBInitializer} implementations
that should be included
     * as part of the {@link org.omg.CORBA.ORB#init(String[], java.util.Properties)
initialization of an ORB}.
     *
     * @return the names of the classes implementing {@code ORBInitializer}. Will not be {@code
null}.
     */
    public List<String> getORBInitializerClasses() {
        return Collections.unmodifiableList(Arrays.asList(

"com.arjuna.ats.jts.orbspecific.jacorb.interceptors.interposition.InterpositionORBInitializerImpl"
"com.arjuna.ats.jbossatx.jts.InboundTransactionCurrentInitializer"));
    }
}
```

and provide it to the builder:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE).build();
```

For a dynamic capability:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo", true, JTSCapability.INSTANCE).build();
```

A capability can provide both a custom integration API and install a service:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE)
                .setServiceType(DataSource.class)
                .build();
```

Registering and unregistering your capability

Once you have your capability, you need to ensure it gets registered with the WildFly Core kernel when your resource is added. This is easily done simply by providing a reference to the capability to the resource's `ResourceDefinition`. This assumes your add handler is a subclass of the standard `org.jboss.as.controller.SimpleResourceDefinition`. `SimpleResourceDefinition` provides a `Parameters` class that provides a builder-style API for setting up all the data needed by your definition. This includes a `setCapabilities` method that can be used to declare the capabilities provided by resources of this type.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    . . .

    MyResourceDefinition() {
        super(new SimpleResourceDefinition.Parameters(PATH, RESOLVER)
            .setAddHandler(MyAddHandler.INSTANCE)
            .setRemoveHandler(MyRemoveHandler.INSTANCE)
            .setCapabilities(FOO_CAPABILITY)
            );
    }
}
```

Your add handler needs to extend the standard `org.jboss.as.controller.AbstractAddStepHandler` class or one of its subclasses:

```
class MyAddHandler extends AbstractAddStepHandler() {
```

`AbstractAddStepHandler`'s logic will register the capability when it executes.

Your remove handler must also extend of the standard `org.jboss.as.controller.AbstractRemoveStepHandler` or one of its subclasses.

```
class MyRemoveHandler extends AbstractRemoveStepHandler() {
```

`AbstractRemoveStepHandler`'s logic will deregister the capability when it executes.

If for some reason you cannot base your `ResourceDefinition` on `SimpleResourceDefinition` or your handlers on `AbstractAddStepHandler` and `AbstractRemoveStepHandler` then you will need to take responsibility for registering the capability yourself. This is not expected to be a common situation. See the implementation of those classes to see how to do it.

Installing, accessing and removing the service provided by your capability

If your capability installs a service, you should use the `RuntimeCapability` when you need to determine the service's name. For example in the `Stage.RUNTIME` handling of your "add" step handler. Here's an example for a statically named capability:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

If the capability is dynamically named, get the dynamic part of the name from the `OperationContext` and use that when getting the service name:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        String myName = context.getCurrentAddressValue();
        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName(myName);
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

The same patterns should be used when accessing or removing the service in handlers for `remove`, `write-attribute` and custom operations.

If you use `ServiceRemoveStepHandler` for the `remove` operation, simply provide your `RuntimeCapability` to the `ServiceRemoveStepHandler` constructor and it will automatically remove your capability's service when it executes.

14.4.2 Basics of Using Other Capabilities

When a capability needs another capability, it only refers to it by its string name. A capability should not reference the `RuntimeCapability` object of another capability.

Before a capability can look up the service name for a required capability's service, or access its custom integration API, it must first register a requirement for the capability. This must be done in Stage.MODEL, while service name lookups and accessing the custom integration API is done in Stage.RUNTIME.

Registering a requirement for a capability is simple.

Registering a hard requirement for a static capability

If your capability has a hard requirement for a statically named capability, simply declare that to the builder for your `RuntimeCapability`. For example, WildFly's JTS capability requires both a basic transaction support capability and IIOP capabilities:

```
static final RuntimeCapability<JTSCapability> JTS_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.transactions.jts", new JTSCapability())
                .addRequirements("org.wildfly.transactions", "org.wildfly.iiop.orb",
"org.wildfly.iiop.corba-naming")
                .build();
```

When your capability is registered with the system, the WildFly Core kernel will automatically register any static hard requirements declared this way.

Registering a requirement for a dynamically named capability

If the capability you require is dynamically named, usually your capability's resource will include an attribute whose value is the dynamic part of the required capability's name. You should declare this fact in the `AttributeDefinition` for the attribute using the `SimpleAttributeDefinitionBuilder.setCapabilityReference` method.

For example, the WildFly "remoting" subsystem's "org.wildfly.remoting.connector" capability has a requirement for a dynamically named socket-binding capability:

```
public class ConnectorResource extends SimpleResourceDefinition {

    . . .

    static final String SOCKET_CAPABILITY_NAME = "org.wildfly.network.socket-binding";
    static final RuntimeCapability<Void> CONNECTOR_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.remoting.connector", true)
                    .build();

    . . .

    static final SimpleAttributeDefinition SOCKET_BINDING =
            new SimpleAttributeDefinitionBuilder(CommonAttributes.SOCKET_BINDING,
ModelType.STRING, false)

.addAccessConstraint(SensitiveTargetAccessConstraintDefinition.SOCKET_BINDING_REF)
                .setCapabilityReference(SOCKET_CAPABILITY_NAME, CONNECTOR_CAPABILITY)
                .build();
```

If the "add" operation handler for your resource extends `AbstractAddStepHandler` and the handler for `write-attribute` extends `AbstractWriteAttributeHandler`, the declaration above is sufficient to ensure that the appropriate capability requirement will be registered when the attribute is modified.

Depending upon a service provided by another capability

Once the requirement for the capability is registered, your `OperationStepHandler}}s can use the {{OperationContext` to discover the name of the service provided by the required capability.

For example, the "add" handler for a remoting connector uses the `OperationContext` to find the name of the needed {{SocketBinding} service:

```
final String socketName = ConnectorResource.SOCKET_BINDING.resolveModelAttribute(context,
fullModel).asString();
        final ServiceName socketBindingName =
context.getCapabilityServiceName(ConnectorResource.SOCKET_CAPABILITY_NAME, socketName,
SocketBinding.class);
```

That service name is then used to add a dependency on the `SocketBinding` service to the remoting connector service.

If the required capability isn't dynamically named, `OperationContext` exposes an overloaded `getCapabilityServiceName` variant. For example, if a capability requires a remoting Endpoint:

```
ServiceName endpointService = context.getCapabilityServiceName("org.wildfly.remoting.endpoint",
Endpoint.class);
```

Using a custom integration API provided by another capability

In your `Stage.RUNTIME` handler, use `OperationContext.getCapabilityRuntimeAPI` to get a reference to the required capability's custom integration API. Then use it as necessary.

```
List<String> orbInitializers = new ArrayList<String>();
        . . .
        JTSCapability jtsCapability =
context.getCapabilityRuntimeAPI(IIOPExtension.JTS_CAPABILITY, JTSCapability.class);
        orbInitializers.addAll(jtsCapability.getORBInitializerClasses());
```

Runtime-only requirements

If your capability has a runtime-only requirement for another capability, that means that if that capability is present in `Stage.RUNTIME` you'll use it, and if not you won't. There is nothing about the configuration of your capability that triggers the need for the other capability; you'll just use it if it's there.

In this case, use `OperationContext.hasOptionalCapability` in your `Stage.RUNTIME` handler to check if the capability is present:

```
protected void performRuntime(final OperationContext context, final ModelNode operation, final
ModelNode model) throws OperationFailedException {

        ServiceName myServiceName = MyResource.FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> myService = createService(context, model);
        ServiceBuilder<DataSource> builder = context.getTarget().addService(myServiceName,
myService);

        // Inject a "Bar" into our "Foo" if bar capability is present
        if (context.hasOptionalCapability("com.example.bar",
MyResource.FOO_CAPABILITY.getName(), null) {
            ServiceName barServiceName = context.getCapabilityServiceName("com.example.bar",
Bar.class);
            builder.addDependency(barServiceName, Bar.class, myService.getBarInjector());
        }

        builder.install();
    }
```

The WildFly Core kernel will not register a requirement for the "com.example.bar" capability, so if a configuration change occurs that means that capability will no longer be present, that change will not be rolled back. Because of this, runtime-only requirements can only be used with capabilities that declare in their contract that they support such use.

Using a capability in a DeploymentUnitProcessor

{{DeploymentUnitProcessor}}s are likely to have a need to interact with capabilities, in order to create service dependencies from a deployment service to a capability provided service or to access some aspect of a capability's custom integration API that relates to deployments.

If a `DeploymentUnitProcessor` associated with a capability implementation needs to utilize its own capability object, the `DeploymentUnitProcessor` authors should simply provide it with a reference to the `RuntimeCapability` instance. Service name lookups or access to the capabilities custom integration API can then be performed by invoking the methods on the `RuntimeCapability`.

If you need to access service names or a custom integration API associated with a different capability, you will need to use the `org.jboss.as.controller.capability.CapabilityServiceSupport` object associated with the deployment unit. This can be found as an attachment to the `DeploymentPhaseContext`:

```
class MyDUP implements DeploymentUntiProcessor {

    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

        AttachmentKey<CapabilityServiceSupport> key =
org.jboss.as.server.deployment.Attachments.DEPLOYMENT_COMPLETE_SERVICES;
        CapabilityServiceSupport capSvcSupport = phaseContext.getAttachment(key);
```

Once you have the `CapabilityServiceSupport` you can use it to look up service names:

```
ServiceName barSvcName = capSvcSupport.getCapabilityServiceName("com.example.bar");
        // Determine what 'baz' the user specified in the deployment descriptor
        String bazDynamicName = getSelectedBaz(phaseContext);
        ServiceName bazSvcName = capSvcSupport.getCapabilityServiceName("com.example.baz",
bazDynamicName);
```

It's important to note that when you request a service name associated with a capability, the `CapabilityServiceSupport` will give you one regardless of whether the capability is actually registered with the kernel. If the capability isn't present, any service dependency your DUP creates using that service name will eventually result in a service start failure, due to the missing dependency. This behavior of not failing immediately when the capability service name is requested is deliberate. It allows deployment operations that use the `rollback-on-runtime-failure=false` header to successfully install (but not start) all of the services related to a deployment. If a subsequent operation adds the missing capability, the missing service dependency problem will then be resolved and the MSC service container will automatically start the deployment services.

You can also use the `CapabilityServiceSupport` to obtain a reference to the capability's custom integration API:

```
// We need custom integration with the baz capability beyond service injection
        BazIntegrator bazIntegrator;
        try {
            bazIntegrator = capSvcSupport.getCapabilityRuntimeAPI("com.example.baz",
bazDynamicName, BazIntegrator.class);
        } catch (NoSuchCapabilityException e) {
            //
            String msg = String.format("Deployment %s requires use of the 'bar' capability but
it is not currently registered",
                                       phaseContext.getDeploymentUnit().getName());
            throw new DeploymentUnitProcessingException(msg);
        }
```

Note that here, unlike the case with service name lookups, the `CapabilityServiceSupport` will throw a checked exception if the desired capability is not installed. This is because the kernel has no way to satisfy the request for a custom integration API if the capability is not installed. The `DeploymentUnitProcessor` will need to catch and handle the exception.

14.4.3 Detailed API

The WildFly Core kernel's API for using capabilities is covered in detail in the javadoc for the RuntimeCapability and RuntimeCapability.Builder classes and the OperationContext and CapabilityServiceSupport interfaces.

Many of the methods in `OperationContext` related to capabilities have to do with registering capabilities or registering requirements for capabilities. Typically non-kernel developers won't need to worry about these, as the abstract `OperationStepHandler` implementations provided by the kernel take care of this for you, as described in the preceding sections. If you do find yourself in a situation where you need to use these in an extension, please read the javadoc thoroughly.