Cloning (v0.4)

Glyn Normington

March 19, 2009

The Jersey release of the SpringSource dm Server will support *cloning*. This document defines cloning and explores the requirements that it must satisfy.

Refer to JIRA issue `ENGINE-825` for associated code changes.

Contents

1 Requirements

The requirement for cloning derives from four problems: pinning, private statics, discarded optional imports, and fragment hosts.

1.1 Pinning

When an OSGi bundle is resolved, its mandatory dependencies are satisfied and any optional dependencies that cannot be satisfied are discarded. Dependencies are satisfied by wiring them to bundles that satisfy local constraints, such as the version of an exported package belonging to the version range of a package import, and which satisfy global constraints such as the *uses* constraints between packages.

So the way in which a given resolved bundle is wired, that is the way in which its dependencies are wired, depends on the set of bundles available at the time the given bundle was resolved, the order in which these bundles were installed, which of these bundles were already resolved, and how those resolved bundles were wired.

Indeed these factors may prevent a bundle from being resolved if its dependencies cannot be met in a way which satisfies all the constraints. The most basic reason that a bundle cannot be resolved is if its local constraints cannot be met by existing bundles. This can be overcome by provisioning more bundles to satisfy those dependencies.

A more subtle reason that a bundle cannot be resolved is that its local constraints can be met by existing bundles but there is no way of satisfying the global constraints, particularly the uses constraints. In some cases, one or more bundles would have been capable of satisfying the local and global constraints if only they hadn't already been wired in a particular way. These bundles are said to be *pinned*. Note that pinning arises only from an inability to satisfy global constraints.

1.2 Private Statics

A resolved bundle uses a single class loader to load its classes and resources. Each of the classes defined by the bundle's class loader has a set of zero or more static data items.

Some scenarios require more than one set of static data items per bundle. For example, if an application uses Log4j to do logging, then the configuration of logging is held in Log4j statics. If more than one application was to use Log4j, then to avoid those applications interfering with each other's logging configuration, there would need to be a separate set of Log4j statics per application using Log4j.

In some situations, the bundle *expressing a dependency* needs to ensure there will be a private copy of the statics of the bundle satisfying the dependency. In other situations, the bundle *providing* the dependency needs to ensure its statics will always be provided as a private copy. But in yet other situations, for instance ActiveMQ, the bundle providing the dependency needs to ensure its statics will never be duplicated, for instance if it is controlling some singleton resource such as a queue manager.

1.3 Discarded Optional Imports

The discussion of pinning described how optional imports are discarded during resolution. After a bundle has been resolved, the provisioning system contents may change such that some optional imports which were discarded would no longer be discarded if the bundle was re-resolved.

Discarded optional imports could be re-instated by refreshing the bundle containing the imports. However, such a refresh may be too disruptive, for instance if a large number of other bundles depend on the bundle being refreshed.

An alternative approach is to put a private copy of the bundle inside the application. This copy can then wire its optional imports.

Fortunately, in the cases where they are significant, discarded optional imports cause a resolution error, so it is not necessary to detect specifically when a scoped application depends on a package corresponding to a dropped optional in a bundle on which the application also depends.

1.4 Fragment Hosts

When a given bundle is resolved fragment bundles which specify the given bundle as a host may be attached to the host. It is not possible to attach a fragment bundle of a scoped application to a bundle in the global scope as this would allow application content to "leak" into the global scope. It would also be impossible to define the synthetic context of the application to cover the fragment bundle but not its host.

Manually packaging a private copy of the host bundle inside the application solves this problem but is poor packaging.

2 Cloning

Cloning refers to the copying of a bundle or bundles into a scope.

Cloning can apply to bundles or libraries.

Cloning automatically driven by unsatisfied uses constraints addresses pinning and discarded optional imports.

Cloning for private statics is specified explicitly and manually in bundle manifests.

2.1 Approach

We concentrate on cloning of bundles and libraries as specified in `import-bundle` and `import-library` statements and library and bundle declarations and on auto-cloning induced by unsatisfied uses constraints.

Suppression of cloning in `import-bundle` and `import-library` statements and library and bundle declarations is also considered.

2.2 Divide and Conquer

Cloning poses three separate problems: determining the bundles to clone; creating clones and managing them subsequently; enabling Spring DM to cope with Spring framework clones.

2.2.1 Determining the Bundles to Clone

Cloning may be mandated or suppressed explicitly on `import-library`, `import-bundle`, or a library or bundle declaration. The mandated cloning operations are performed first.

Then zero or more bundles are added to the cloned set incrementally until the application resolves.

The need to re-run the deployer's metadata transformations is somewhat simplified by the observation that the set of cloned bundles is monotonically increasing, so re-scoping never requires de-scoping.

Finally, if we clone the Spring framework bundles used by the kernel, we need to clone the Spring DM extender and associated bundles such as "kernel.dm". This is discussed below.

2.2.2 Creating and Managing Clones

Clones can be created by copying bundles into an application's scope. This minimises sharing but is simpler to manage subsequently than creating sets of cloned bundles shared by more than one application.

When the original copy of a cloned bundle is refreshed, the clones must also be refreshed.

2.2.3 Spring DM and Spring Framework Clones

This is the topic of the next section.

Note that this problem could arise even without full cloning support if an application embedded the Spring framework.

3 Spring DM and Spring Framework Clones

The dm Server infrastructure is separated from the clone specific application context and deals in terms of the ModuleContext type standardised by OSGi RFC 124. This type should be fronted by a MBean for consumption by applications and third parties which do not wish to be tied to specific types.

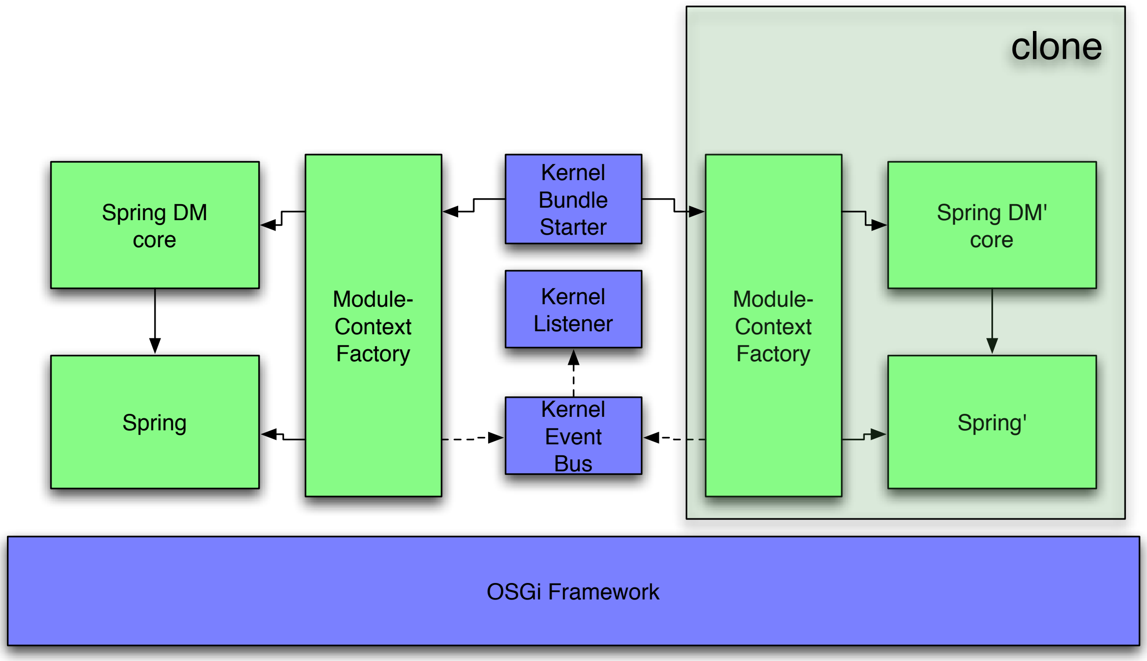

Figure 1: Reusing Spring DM Core and Conforming to a Standard API

The Wight release of the server extended the Spring DM application context type. Although this is necessary to support Spring Web applications, the personality SPI should offer a set of specific contracts, for example to allow beans to be added to an application context. *Need to consider resource loading and context class loading and whether these have any implications for cloning.*

This design also removes a number of issues associated with the BundleStarter having to track the creation of application contexts by the Spring DM extender by introducing a much more direct ModuleContextFactory interface implemented by a clone specific instance.

4 Z Specification

4.1 Introduction

An abstract model is constructed which should be just sufficient to reason about cloning.

4.2 Types

Bundles are uniquely identified by bundle id. Bundles are uniquely identified at any point in their lifecycle by bundle symbolic name and version, although these may change if the bundle is updated. Bundles also have content that is not modelled further here.

$$[BundleId, BSN, BV, BundleContent]$$

Bundles have a bundle symbolic name, a bundle version, and some content which is not further described.

$$\begin{array}{|l}
\textit{Bundle} \\ \hline
bsn : BSN \\
bv : BV \\
content : BundleContent \\ \hline
\end{array}$$

An OSGi framework identifies each bundle by id. Each bundle is also uniquely identified by the combination of its bundle symbolic name and bundle version.

$$\begin{array}{|l}
\textit{Framework} \\ \hline
bundles : BundleId \mathrel{⤔} Bundle \\
nv : BSN \times BV \mathrel{⤔} Bundle \\
bid : BSN \times BV \mathrel{⤔} BundleId \\ \hline
nv = \{ b : \operatorname{ran} bundles \bullet (b.bsn, b.bv) \mapsto b \} \\
bid = bundles^{\sim} \circ nv \\ \hline
\end{array}$$

Successfully installing a bundle adds it to the bundles in the framework. Since *bundles* is an injection, the installed bundle's symbolic name and version must not identify another bundle in the framework. Since *nv* is an injection, the bundle id that identifies the installed bundle was not previously in use.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{FInstall} \quad}} \\
\Delta \mathit{Framework} \\
\mathit{bundleData}? : \mathit{Bundle} \\
\hline
\exists\, \mathit{id} : \mathit{BundleId} \bullet \mathit{bundles}' = \mathit{bundles} \cup \{\mathit{id} \mapsto \mathit{bundleData}?\} \\
\hline
\end{array}$$

Successfully updating a bundle with a given id replaces the bundle in the framework and therefore requires that the updated bundle's symbolic name and version do not clash with those of another bundle in the framework.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{FUpdate} \quad}} \\
\Delta \mathit{Framework} \\
\mathit{id}? : \mathit{BundleId} \\
\mathit{bundleData}? : \mathit{Bundle} \\
\hline
\mathit{id}? \in \operatorname{dom} \mathit{bundles} \\
\mathit{bundles}' = \mathit{bundles} \oplus \{\mathit{id}? \mapsto \mathit{bundleData}?\} \\
\hline
\end{array}$$

It is also possible to update a bundle with a given symbolic name and version. Again the updated bundle's symbolic name and version must not clash with those of another bundle in the framework.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{FUpdateNV} \quad}} \\
\Delta \mathit{Framework} \\
\mathit{bsn}? : \mathit{BSN} \\
\mathit{bv}? : \mathit{BV} \\
\mathit{bundleData}? : \mathit{Bundle} \\
\hline
\exists\, \mathit{id}? : \mathit{BundleId} \mid \mathit{id}? = \mathit{bid}(\mathit{bsn}?, \mathit{bv}?) \bullet \mathit{FUpdate} \\
\hline
\end{array}$$

Any installed bundle may be uninstalled.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{FUninstall} \quad}} \\
\Delta \mathit{Framework} \\
\mathit{id}? : \mathit{BundleId} \\
\hline
\mathit{id}? \in \operatorname{dom} \mathit{bundles} \\
\mathit{bundles}' = \{\mathit{id}?\} \lhd \mathit{bundles} \\
\hline
\end{array}$$

4.3 Scope

Informally, cloning involves copying a bundle into a scope. So we make the notion of scope precise.

A scope is a collection of bundles, indexed by their symbolic names and versions, some of which are clones.

$$\begin{array}{|l}
\textit{Scope} \\
\hline
\textit{snv} : \textit{BSN} \times \textit{BV} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \textit{Bundle} \\
\textit{clones} : \mathbb{P}(\textit{BSN} \times \textit{BV}) \\
\hline
\textit{snv} = \{ b : \operatorname{ran} \textit{snv} \bullet (b.\textit{bsn}, b.\textit{bv}) \mapsto b \} \\
\textit{clones} \subseteq \operatorname{dom} \textit{snv} \\
\hline
\end{array}$$

A bundle may be added to a scope.

$$\begin{array}{|l}
\textit{AddBundle} \\
\hline
\Delta \textit{Scope} \\
\textit{bundleData}? : \textit{Bundle} \\
\hline
\operatorname{ran} \textit{snv}' = \operatorname{ran} \textit{snv} \cup \{ \textit{bundleData}? \} \\
\hline
\end{array}$$

$$\textit{Add} \mathrel{\widehat=} [\textit{AddBundle} \mid \textit{clones}' = \textit{clones}]$$

A clone of a bundle may also be added to a scope.

$$\textit{AddClone} \mathrel{\widehat=} [\textit{AddBundle} \mid \textit{clones}' = \textit{clones} \cup \{(\textit{bundleData}?.\textit{bsn}, \textit{bundleData}?.\textit{bv})\}]$$

A bundle may be updated in a scope.

$$\begin{array}{|l}
\textit{UpdateNVInScope} \\
\hline
\Delta \textit{Scope} \\
\textit{bsn}? : \textit{BSN} \\
\textit{bv}? : \textit{BV} \\
\textit{bundleData}? : \textit{Bundle} \\
\hline
(\textit{bsn}?, \textit{bv}?) \in \operatorname{dom} \textit{snv} \\
\textit{snv}' = \textit{snv} \oplus \{ (\textit{bsn}?, \textit{bv}?) \mapsto \textit{bundleData}? \} \\
\textit{clones}' = \textit{clones} \\
\hline
\end{array}$$

Any clone of a bundle may be updated in a scope.

UpdateCloneNVInScope

$\Delta Scope$
$bsn? : BSN$
$bv? : BV$
$bundleData? : Bundle$

$bsn? = bundleData?.bsn$
$bv? = bundleData?.bv$
$(bsn?, bv?) \in clones \Rightarrow snv' = snv \oplus \{(bsn?, bv?) \mapsto bundleData?\}$
$(bsn?, bv?) \notin clones \Rightarrow snv' = snv$
$clones' = clones$

The above operation is undefined if the bundle symbolic name or bundle version are changed by the update. In such circumstances, the implementation escalates the update for the named scope by uninstalling and then re-installing it.

A bundle may be removed from a scope.

Remove

$\Delta Scope$
$b? : Bundle$

$snv' = snv \rhd \{b?\}$
$clones' = clones \setminus \{snv^{\sim}\ b?\}$

4.4 Kernel

In a kernel, certain scopes are named.

$[ScopeName]$

We define a utility function for extracting the *snv* function from a scope.

$scopesnv : Scope \rightarrow (BSN \times BV \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow Bundle)$

$scopesnv = (\lambda\, Scope \bullet snv)$

A kernel has an OSGi framework, a single global scope, and a collection of named scopes. The scopes partition the bundles in the framework. The global scope does not have any clones.

The schema defines two components of the state in terms of the others. *isGlobal* is the set of bundles in the global scope. *scopeName* gives the name of the scope containing each bundle which is not in the global scope.

$$
\begin{array}{|l}
\textit{Kernel} \\
\hline
\textit{Framework} \\
g : \textit{Scope} \\
s : \textit{ScopeName} \rightarrowtail\!\!\!\!\!\!\leftrightarrow \textit{Scope} \\
\textit{isGlobal} : \mathbb{P}(\textit{BSN} \times \textit{BV}) \\
\textit{scopeName} : \textit{BSN} \times \textit{BV} \mathrel{+\!\!\!\!\rightarrow} \textit{ScopeName} \\
\hline
\textit{scopesnv} \circ s \ \mathsf{partition} \ \textit{nv} \setminus g.\textit{snv} \\
g.\textit{clones} = \emptyset \\
\textit{isGlobal} = \operatorname{dom} g.\textit{snv} \\
\operatorname{dom} \textit{scopeName} = \operatorname{dom} \textit{nv} \setminus \textit{isGlobal} \\
\forall \textit{bsn} : \textit{BSN};\ \textit{bv} : \textit{BV} \mid (\textit{bsn}, \textit{bv}) \in \operatorname{dom} \textit{scopeName} \bullet \\
\qquad (\textit{bsn}, \textit{bv}) \in \operatorname{dom}((\textit{scopesnv} \circ s \circ \textit{scopeName})(\textit{bsn}, \textit{bv})) \\
\hline
\end{array}
$$

We define some promotion schemas.

Some operations are directed at the global scope but may also affect named scopes.

$$
\begin{array}{|l}
\textit{GlobalScopeOp} \\
\hline
\Delta \textit{Kernel} \\
\Delta \textit{Scope} \\
\hline
g = \theta \textit{Scope} \\
g' = \theta \textit{Scope}' \\
\hline
\end{array}
$$

Some operations directed at the global scope do not affect named scopes.

$$
\begin{array}{|l}
\textit{GlobalOnlyScopeOp} \\
\hline
\textit{GlobalScopeOp} \\
\hline
s' = s \\
\hline
\end{array}
$$

Some operations are directed at a specific named scope but may also affect the global scope and other named scopes.

$$
\begin{array}{|l}
\textit{NamedScopeOp} \\
\Delta \textit{Kernel} \\
\Delta \textit{Scope} \\
sn? : \textit{ScopeName} \\
\hline
sn? \in \operatorname{dom} s \\
sn? \in \operatorname{dom} s' \\
s\ sn? = \theta \textit{Scope} \\
s'\ sn? = \theta \textit{Scope}' \\
\hline
\end{array}
$$

Some operations are directed at a specific named scope and do not affect the global scope or any other named scopes.

$$
\begin{array}{|l}
\textit{NamedOnlyScopeOp} \\
\Delta \textit{Kernel} \\
\Delta \textit{Scope} \\
sn? : \textit{ScopeName} \\
\hline
sn? \in \operatorname{dom} s \\
g' = g \\
s\ sn? = \theta \textit{Scope} \\
s' = s \oplus \{ sn? \mapsto \theta \textit{Scope}' \} \\
\hline
\end{array}
$$

4.4.1 Installing Bundles

A bundle may be installed in the global scope. The bundle is installed in the OSGi framework and added to the global scope.

$$\textit{InstallGlobal} \mathrel{\widehat{=}} \exists\, \Delta \textit{Scope} \bullet \textit{GlobalOnlyScopeOp} \land \textit{FInstall} \land \textit{Add}$$

Bundles may also be installed in a named scope. The bundle is installed in the OSGi framework and added to the named scope. Unfortunately, this does not have the desired scoping effect: the operation is not defined for a bundle whose symbolic name and version clash with those of an existing bundle.

$$\textit{InstallInNamedScopeRaw} \mathrel{\widehat{=}} \exists\, \Delta \textit{Scope} \bullet \textit{NamedOnlyScopeOp} \land \textit{FInstall} \land \textit{Add}$$

In order to prevent a bundle in a named scope from clashing with a bundle with the same symbolic name and version in another scope, the bundle's symbolic name is transformed by prefixing it with the scope name.

$$| \; \mathit{prefix} : \mathit{ScopeName} \times \mathit{BSN} \rightarrowtail \mathit{BSN}$$

A prefixing schema is defined for convenience. The bundle's symbolic name is prefixed with the scope name but the bundle version and content are unchanged.

$$\begin{array}{|l}
\textit{PrefixBundle} \\
\hline
b, pb : \mathit{Bundle} \\
sn? : \mathit{ScopeName} \\
\hline
pb.bsn = \mathit{prefix}(sn?, b.bsn) \\
pb.bv = b.bv \\
pb.content = b.content \\
\hline
\end{array}$$

PrefixBundle ignores content which refers to symbolic names of bundles in the scope. Modelling how such content is scoped would bloat the specification and add little value.

A bundle is installed in a named scope by transforming its symbolic name and then installing the transformed bundle in the OSGi framework and adding the transformed bundle to the named scope.

$$\mathit{InstallInNamedScope} \mathrel{\widehat=} (\mathit{InstallInNamedScopeRaw}[pb/bundleData?] \land \\ \mathit{PrefixBundle}[bundleData?/b]) \setminus (pb)$$

4.4.2 Cloning Bundles

Bundles may also be cloned into a named scope. The bundle is installed in the OSGi framework and cloned into the named scope. Again, this does not have the desired scoping effect: the operation is not defined for a bundle whose symbolic name and version clash with those of an existing bundle.

$$\mathit{CloneRaw} \mathrel{\widehat=} \exists \Delta \mathit{Scope} \bullet \mathit{NamedOnlyScopeOp} \land \mathit{FInstall} \land \mathit{AddClone}$$

A bundle is cloned into a named scope by transforming its symbolic name and then installing the transformed bundle in the OSGi framework and cloning the transformed bundle into the named scope.

$$\mathit{Clone} \mathrel{\widehat=} (\mathit{CloneRaw}[pb/bundleData?] \land \\ \mathit{PrefixBundle}[bundleData?/b]) \setminus (pb)$$

4.4.3 Updating Bundles

A bundle in the global scope may be updated. Any clones of the bundle are also updated.

$$
\begin{array}{|l}
\hline
\mathit{UpdateGbl} \\
\mathit{GlobalScopeOp} \\
\mathit{FUpdate} \\
\mathit{UpdateNVInScope} \\
\hline
\mathit{bsn?} = (\mathit{bundles}\ \mathit{id?}).\mathit{bsn} \\
\mathit{bv?} = (\mathit{bundles}\ \mathit{id?}).\mathit{bv} \\
(\mathit{bsn?}, \mathit{bv?}) \in \mathit{isGlobal} \\
\mathrm{dom}\, s' = \mathrm{dom}\, s \\
\forall\, \mathit{sn?} : \mathrm{dom}\, s \bullet \\
\quad \mathit{NamedScopeOp} \land \\
\quad (\exists\, \mathit{pb} : \mathit{Bundle} \bullet \\
\quad\quad \mathit{UpdateCloneNVInScope}[\mathit{pb}/\mathit{bundleData?}] \land \\
\quad\quad \mathit{PrefixBundle}[\mathit{bundleData?}/b]) \\
\hline
\end{array}
$$

$$\mathit{UpdateGlobal} \mathrel{\widehat{=}} \exists\, \Delta \mathit{Scope};\ \mathit{bsn?} : \mathit{BSN};\ \mathit{bv?} : \mathit{BV} \bullet \mathit{UpdateGbl}$$

A bundle in a named scope is updated by updating it in the OSGi framework and in the named scope.

$$
\begin{array}{|l}
\hline
\mathit{UpdateInNS} \\
\mathit{NamedOnlyScopeOp} \\
\mathit{FUpdate} \\
\mathit{UpdateNVInScope} \\
\hline
\mathit{bsn?} = (\mathit{bundles}\ \mathit{id?}).\mathit{bsn} \\
\mathit{bv?} = (\mathit{bundles}\ \mathit{id?}).\mathit{bv} \\
(\mathit{bsn?}, \mathit{bv?}) \notin \mathit{isGlobal} \\
\mathit{sn?} = \mathit{scopeName}(\mathit{bsn?}, \mathit{bv?}) \\
\hline
\end{array}
$$

$$\mathit{UpdateInNamedScope} \mathrel{\widehat{=}} \exists\, \Delta \mathit{Scope};\ \mathit{bsn?} : \mathit{BSN};\ \mathit{bv?} : \mathit{BV};\ \mathit{sn?} : \mathit{ScopeName} \bullet \mathit{UpdateInNS}$$

The update operation need not distinguish between the global and named scope cases.

$$\mathit{Update} \mathrel{\widehat{=}} \mathit{UpdateGlobal} \lor \mathit{UpdateInNamedScope}$$

There is a corresponding update operation which takes a symbolic name and version.

$$\begin{array}{|l}
\text{\_\_ } \mathit{UpdateNV} \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{Kernel} \\
\mathit{bsn}? : \mathit{BSN} \\
\mathit{bv}? : \mathit{BV} \\
\mathit{bundleData}? : \mathit{Bundle} \\
\hline
\exists\, \mathit{id}? : \mathit{BundleId} \mid \mathit{id}? = \mathit{bid}(\mathit{bsn}?, \mathit{bv}?) \bullet \mathit{Update} \\
\hline
\end{array}$$

4.4.4 Uninstalling Bundles

A bundle is uninstalled from the global scope by uninstalling it from the OSGi framework and removing it from the global scope.

$$\begin{array}{|l}
\text{\_\_ } \mathit{UninstallGbl} \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{GlobalOnlyScopeOp} \\
\mathit{FUninstall} \\
\mathit{Remove} \\
\hline
b? = \mathit{bundles}\ \mathit{id}? \\
\hline
\end{array}$$

$$\mathit{UninstallGlobal} \mathrel{\widehat{=}} \exists\, \Delta \mathit{Scope};\ b? : \mathit{Bundle} \bullet \mathit{UninstallGbl}$$

Note that the above operation does not uninstall any clones of the bundle being uninstalled.

A bundle is uninstalled from a named scope by uninstalling it from the OSGi framework and removing it from the named scope.

$$\begin{array}{|l}
\text{\_\_ } \mathit{UninstallFromNS} \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{NamedOnlyScopeOp} \\
\mathit{FUninstall} \\
\mathit{Remove} \\
\hline
b? = \mathit{bundles}\ \mathit{id}? \\
\hline
\end{array}$$

$$\mathit{UninstallFromNamedScope} \mathrel{\widehat{=}} \exists\, \Delta \mathit{Scope};\ b? : \mathit{Bundle} \bullet \mathit{UninstallFromNS}$$

Note that the above operation *can* be used to uninstall a cloned bundle.

The uninstall operation need not distinguish between the global and named scope cases.

$$\mathit{Uninstall} \mathrel{\widehat{=}} \mathit{UninstallGlobal} \lor \mathit{UninstallFromNamedScope}$$

5 Alternative Designs

Figures 2 and 3 show alternative designs which were considered and rejected.

Both designs reproduce the Spring DM extender inside each clone which causes complications for listening. Cloned extenders need to ignore events outside their 'scope' and the non-cloned extender needs to ignore events associated with clones. So it seems preferable to initiate the building of application contexts at the level of deploying a bundle and delegate building the application context to a clone-specific standard delegate.

Figure 2 forces much kernel and server code to be cloned with corresponding overhead. This makes it much more difficult to integrate with dm Server wide infrastructure such as management and third party code as this also would need cloning. In the limit, this design clones most of the kernel and server and would be better implemented by hosting each clone in its own server instance. Although it should be possible to host each clone on its own server instance, this should not be forced on the users. Especially during development, it is convenient to deploy multiple versions of applications and the libraries they use concurrently.

Figure 3 on the other hand requires a complex mapping of cloned to standard types. The whole mapping is somewhat wasteful given that there is no need to generate cloned events in the first place (except perhaps to support legacy Spring DM applications).

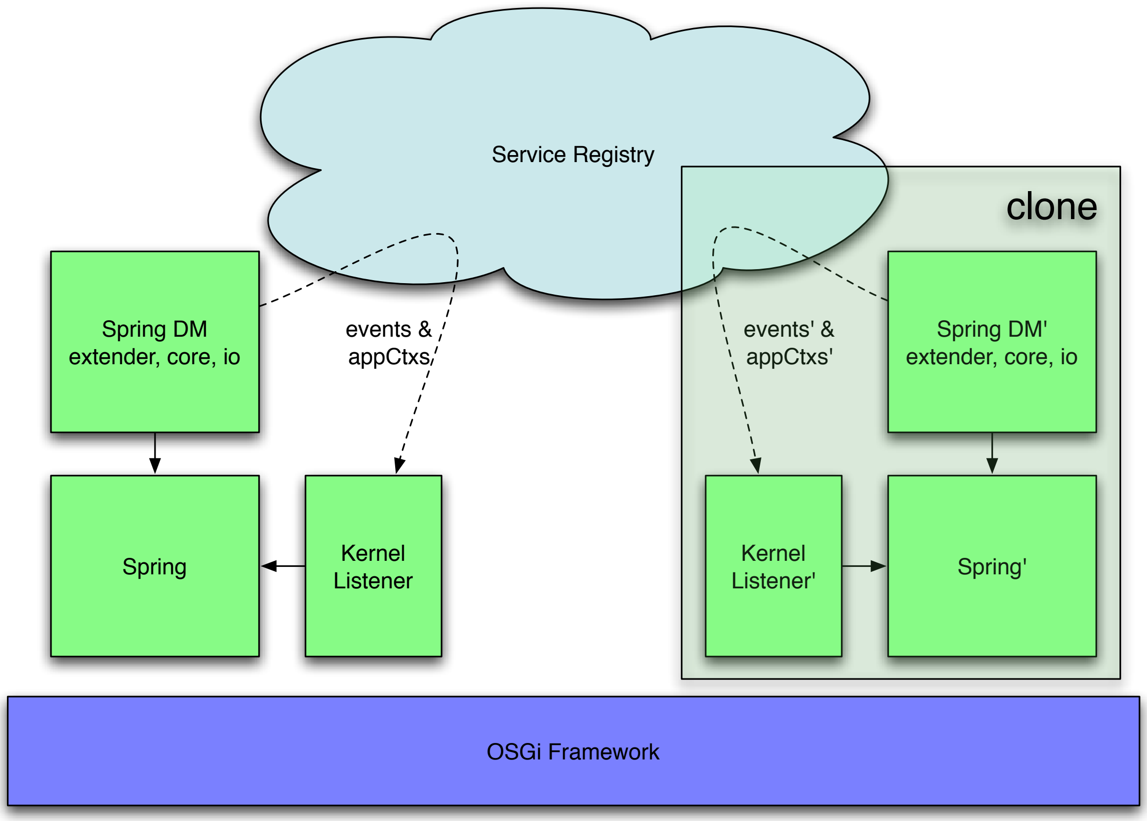

Figure 2: Reusing All of Spring DM and Cloning the Kernel

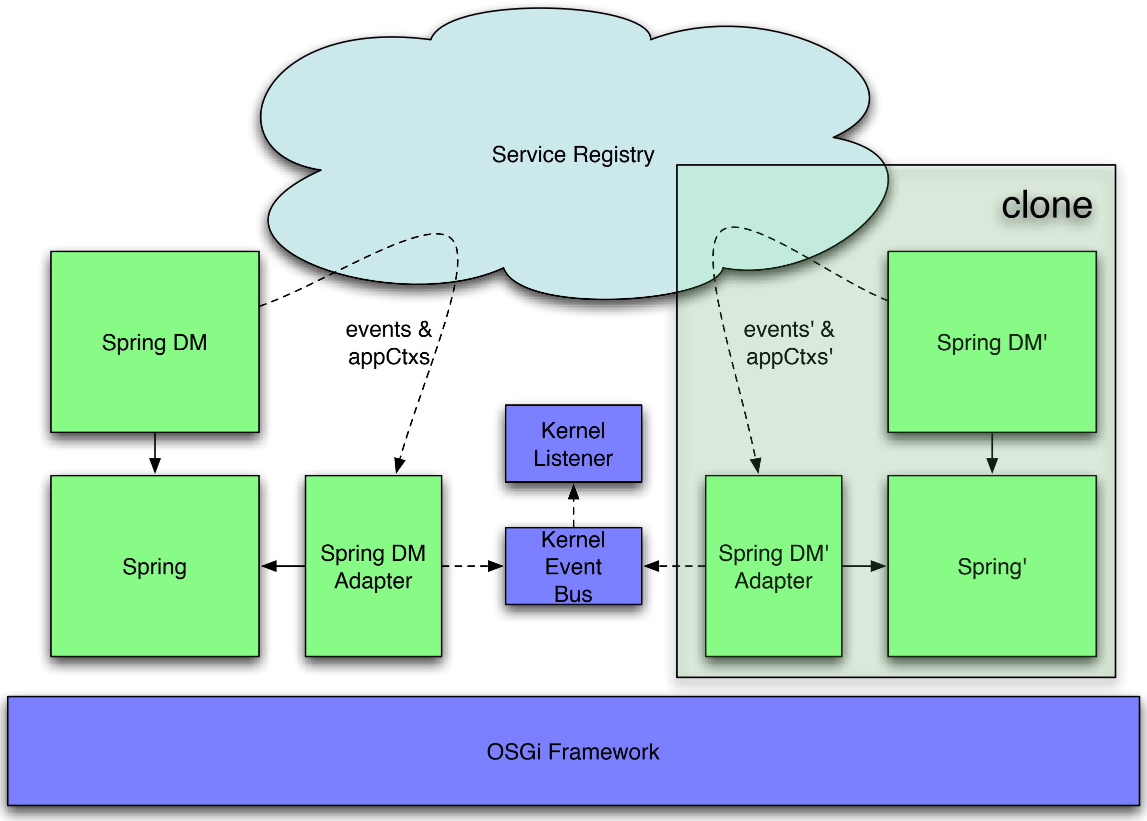

Figure 3: Reusing All of Spring DM and Adapting to the Kernel