Creating and deleting contexts

Joop Ringelberg 10-09-20, 17-01-22 Version: 2

Introduction

What user roles have the right to create a context instance? Perspectives grants its users their powers exclusively through perspectives on roles. In order to dole out rights for creating contexts, we introduce a verb specifically for creating contexts, to be bound to a context role in an embedding context: `CreateAndFill`. A user role A with a perspective on context role C that includes this verb, can create a new instance of C and bind it to the external role in a new instance of the possible binding of C – all in one go.

So we see that we do not have the notion of a right to create an instance of a context *as such*; rather, this right is scoped to a particular context *in which* an embedded context may be created. And there may be multiple occasions to create the same context type, as embedded in various other contexts.

Local and remote

In the simplest case, a user role has a `CreateAndFill` perspective on a local context role. It allows them to add a role instance (bound to a context instance) in the context that they belong to.

However, with the existence of Calculated roles, a user role has access to 'remote' roles, too. As long as there is a path leading from the user role's context to a role, the modeller can grant a perspective on that role. And this perspective may include the verb `CreateAndBindContext`.

However, in order to create[1] a context somewhere else, that 'somewhere else' must be well specified. Let's first examine the way the modeller instructs the system to do so, on behalf of some user.

Create a remote context automatically on behalf of a user

First, we create a Calculated role that consists of a remote context role[2]:

```
context: Remote = A >> binding >> context >> C
```

Instances of `Remote` actually are instances of `C`, in some other context.

```
in state <SomeState>
```

[1] And bind; in the following, I will not tediously repeat 'create and bind' but just write 'create', meaning 'create and bind in a new context role'.
[2] Without losing generality, we can assume for the sake of simplicity that roles A and C are functional.

```
on entry
  do for User
    create context X bound to C in A >> binding >> context
```

This causes a new instance of role C to be created in the context identified by `A >> binding >> context`, on entering state `<SomeState>` (for an otherwise unspecified resource). It is filled by the external role of a new instance of context type X.

Of interest is the clause `in A >> binding >> context`. It says *in what context instance* the new context role is created. Notice that it equals the definition of `Remote`, except for the last step. So the clause identifies a particular context instance (remember we created `Remote` to be a functional Calculated role) and creates a new context role in that instance.

Obviously, creating a context locally is simply:

```
create context X bound to C
```

We just leave out the `in` clause to create in the local context. The general syntax is:

```
create context ContextType bound to RoleType in <contextExpression>
```

Create a context, bind in existing role

A variant of the use case given in the previous paragraph would be the situation where we already have a role instance to bind the new context instance in. We then use a variant of the operator:

```
create context_ ContextType bound to <roleExpression>
```

Notice that

- we do not need to provide a context: the role instances selected have a context;
- we can never create an unbound context (see below for a definition) in this way.

How an end user creates contexts

Can an end user create a context remotely, too? Let's explore what that would mean. But before we do so, I'll show how an end user creates a context *locally*.

Through some user interface - let's assume it to be a GUI - the user has navigated to some context instance. In that context instance they play a particular role and with that role comes a screen, providing perspectives on the context instance. Returning to our example, we assume the context to be the one that has context role `C`. Now in order to create a new context in `C`, the end user instructs the core system, through its API, to execute `create context x bound to c` on the current context, with type `X` for role `C`. That's it.

In contrast, assume the end users' current context is the one that holds the role `Remote`. Now, the context holding role `C` is remote. For the end user to create a new context

instance in that remote context, *he has to identify that remote context in some way*. This means that his GUI must offer him a means to do so.

That may be quite easy: for example, just consider a conventional order-detail screen, where both order and detail are represented as contexts. The end user can simply point to a detail and (for example) push a button in it to create a new context in it (say, a delivery sub context; so `C` - a role in the detail context - is bound to a new delivery context). The core API receives, along with the instruction `create context`, the (remote) context representing an order detail and the role type (delivery) to create a new instance in.

Non-functional Calculated roles

What if role `C` were not functional? The `in` clause in the bot's rule would now potentially identify many contexts. If not modified, the rule would create and bind new contexts *in all of them*. It is up to the modeller to decide if that is desired behaviour. To avoid it, he might, for example, filter away from the retrieved contexts those that already have an instance of `B`.

Unbound contexts

Consider the role Chats:

```
context Chats = callExternal cdb:RoleInstances(
"model:SimpleChat$Chat$External" ) returns: Chat$External
```

Its roles are computed by executing a query on the database, requesting role instances of a particular type, in this case the external role of the `Chat` context. Can we create a new instance for such a role?

Yes we can, but we cannot bind it. There simply is no Enumerated role to bind it in. Nevertheless, we can retrieve the instances of the role. The matter how a role is represented (enumerated within a context, or by a query on a database) is an implementation detail rather than a conceptual issue. So `Chats` is a bona fide context role.

If we make the core system execute `create context` on such a role, it will just create a context instance (including its external role)

```
create context Chat in Chats
```

Of interest is the question: how does the core system recognise this situation? How does it decide that the created context cannot be bound? The crucial property of a Calculated context role that identifies it as one in which contexts cannot be bound, is that its computation *consists of a call to a database query*.

There is no consequence for the modeler. He can use the same expressions for bound and unbound embedded contexts. It is the core system that differentiates. This is as it should be, as it is more an implementation detail than a conceptual issue.

Removing and deleting contexts

It follows that we cannot explicitly delete a context: we have no way of specifying the right to do so. However, we can provide a user role with a perspective on a context role that includes the `DeleteWithContext` or `RemoveWithContext` verb[3].

The PDR does not cascade delete resources (otherwise than removing the roles of a context with the context). This is because the true test whether a context can be removed is if there is still a user with a perspective on some part of the context. This is a resource intensive test.

RemoveWithContext

With remove, we can pick and remove a single (context) role instance. Because we identify the role instances we want to remove, we don't have to identify their contexts:

```
remove context <role expression>
```

Notice the `context` keyword that follows remove. Also note that this not only removes the context, but also the context role that is filled with the contexts' external role.

The following would only remove the context role that is filled with (the external role of) the context:

```
remove role <role expression>
```

Furthermore, the former expression requires a perspective with `RemoveWithContext`; the latter just requires `Remove`.

Delete

Deleting removes all instances of a role type.

```
delete role <role type>
```

This will remove all instances of the given role type from the current context.

```
delete role <role type> from <context expression>
```

This will remove all instances of the given role type from the context(s) retrieved by `<context expression>` (a path query). **It is an error** if the type of the role instances is not defined in the type of the contexts!

As with removing a single context, we can delete all contexts that fill a particular context role:

```
delete context <role type> from <context expression>
```

[3] `Remove` does away with a single, specific role instance; `Delete` removes them all (within a context).

Again, all instances of the `<role type>` (which must be of kind context role) are removed as well.

Removing a context is computed per bubble

Consider this: the core system serving a particular end user has removed a context *from the bubble of that user*. Should it communicate all Delta's generated by this process to peers with a perspective on that context role?

No! It must, however, communicate a Delta that says that the context has been removed. Each peer's core works out what to remove from that Delta. This means, for example, that a user without a perspective on a particular role in the removed context, will make a user with that perspective remove that role as well.

Removing unbound contexts

The operational semantics of deleting is different for Database Query Roles. We stipulate that an external role instance and its context are removed permanently from the end users bubble[4] whenever it is being removed from a Database Query Role (and when the removing user has the verb `RemoveWithContext` or `DeleteWithContext` for that calculated role).

The consequence of this is that when this occurs, no DBQ Role based on the same type will show the instance any more.

Synchronisation: delta representation

In this more technical chapter we describe how the various operations should be represented in terms of deltas, to synchronise changes with peers. We describe the change to the users' bubble in terms of assignment statements in bot rules.

Notice that we only create deltas for single role- and context instances, even if, for example, we remove all instances of a role in a context. This is because we record the deltas with the role instance representations themselves. Furthermore, not all users may have the same instances for a particular role type in a context instance (this may arise as a user role has a perspective on a remote role that is defined with a filter).

Create a bound context

`create context <context type> bound to <role type> in <context expression>`

Represent this with:

- For each context instance produced by `<context expression>`, a `UniverseRoleDelta`

[4] The part of the Perspectives Data Universe that is accessible to him.

    - whose `id` identifies that context instance
    - where `roleType` equals <role type>
    - and where the new role instance is in `roleInstances`
    - and `deltaType` is `ConstructEmptyRole`.
- Pair each `UniverseRoleDelta` with a `ContextDelta`.
- For each newly created context instance a `UniverseContextDelta`, where
    - the `id` identifies the new context instance
    - the `contextType` is the context type
    - the `deltaType` is `UniverseContextDeltaType`.
- For the external role of each newly created context instance, create a `UniverseRoleDelta` whose `deltaType` is `ConstructExternalRole`.
- For each pair of `UniverseRoleDelta`'s (that of the binder and the bound external role) a `RoleBindingDelta`
    - whose `id` identifies the new role instance in the context;
    - whose `binding` identifies the new external role.

Delete a bound context

`delete <role type> from <context expression>`

`<role type>` is a context role. Represent this, for each context instance, for each role instance of `<role type>` in that context instance:

- A `UniverseRoleDelta`:
    - whose `id` identifies that context instance
    - where `roleType` equals <role type>
    - and where the role instance is in `roleInstances`
    - and `deltaType` is `RemoveRoleInstance`.

Notice that we do not generate `ContextDelta`'s, nor `UniverseContextDelta`'s. The receiver checks whether any binders of the external role of the context remain. If not, he'll remove it.

Remove a bound context

`remove <role expression>`

We handle this in exactly the same way as for deleting a role type (with `UniverseRoleDelta`'s) but we limit ourselves to the role instances identified by `<role expression>` and their bound contexts.

Create an unbound context

`create context <context type> bound to <role type> in <context expression>`

Represent this with:

- For each newly created context instance a `UniverseContextDelta`, where
  - the `id` identifies the new context instance
  - the `contextType` is the context type
  - the `deltaType` is `UniverseContextDeltaType`.
- For the external role of each newly created context instance, create a `UniverseRoleDelta` whose `deltaType` is `ConstructExternalRole`.

Notice that, in contrast to creating a bound context, we do not create a `UniverseRoleDelta` for the context role, nor a `ContextDelta` to represent its connection to the context instance, nor a `RoleBindingDelta` to represent the binding between context role and external role.

Remove an unbound context

`remove <role expression>`

The sender establishes that we handle an unbound role by finding that it is an external role and there are no binders. If there are binders, deleting an instance from a DBQ Role is a no-op.

It then represents this with a `UniverseRoleDelta`:

  - whose (context) `id` is the context of the external role
  - where `roleType` equals the type of the instances
  - and where the role instance is in `roleInstances`
  - and `deltaType` is `RemoveUnboundExternalRoleInstance`.

The receiver determines that we handle an unbound role from the `deltaType`. If the instance has no binders (on his side), he removes the external role and its context.

Delete an unbound context

`delete <role type> from <context expression>`

The sender recognises the fact that we want to remove an unbound context from the role type. However, we now want to remove *every* unbound context from the given contexts. So we generate a `UniverseRoleDelta` like above, but only for every role instance that can be retrieved from the database that has no binders. As long as it has binders, deleting an instance from a DBQ Role is a no-op.