## Problem

What's the concept of offshoots, and when should I use it?

## Solution

Conceptually, offshoots are remotely visible objects whose life is limited to that of the component or client that created them. Offshoots logically belong to their "mother" component (or client) out of which they "grow" (that's where the funny term comes from). Yet offshoots can have their own functional interfaces.

Technically, under the hood of ZLegacy/ACS.ContainerServices, offshoots are activated as CORBA objects using a POA that is dependent on the "mother" POA.

Offshoots are mainly used internally by ACS, for example to implement the callback mechanism that allows one component to offer a callback object to another component, or for the DAO objects obtained from the CDB. But also the archive uses offshoots for DB cursors and other things.

Here we list some features and compare offshoots to plain programming language level objects and to ACS components. There it may become clearer whether offshoots are attractive for your application:

- Like PL objects, offshoots have no instance name and therefore no identity aside from a direct reference. Components always have an instance name that can be used as a system-wide unique identifier.
- Offshoots are tightly coupled to their mother component: the ZLegacy/ACS.ContainerServices instance (and thus the logger) are shared.
- Like PL objects and dynamic components, instances of offshoots can be created dynamically. Their number is only limited by system resources. Many offshoots of the same type can be created either by the same component or by different components.
- Like PL objects, offshoot instances can only be created locally to the "mother component". In contrast, components can be created on other containers or physical machines.
- Like dynamic components, offshoots can be visible remotely, which PL objects are not. (There is no client side lookup of offshoots available. The mother component must communicate the offshoot reference in order for a client to use the offshoot.)
- With the usual CORBA ORB settings, IDL-declared offshoot methods are automatically called in separate threads and therefore execute asynchronously from the component that created them. In case of PL objects, separate threads would need to be created by the application component.
- It's legal to pass an offshoot reference to another component or offshoot. This is not the case for component references, c.f. FAQPassComponentReference.
- Unlike dynamic components, offshoots don't need to be (partially) configured in advance in the CDB.
- For dynamic components, the ACS manager determines when the component should be unloaded, based on the client count and the unloading settings in the CDB. For offshoots, only the component that created the offshoot determines when the offshoot should be destroyed. This can happen independently of the component's own life (using `ZLegacy/ACS.ContainerServices#deactivateOffshoot`), or forcibly when the component itself is unloaded.
- in Java, remote calls to offshoot methods get intercepted (and optionally logged) by the container, at least when the offshoot class uses the "tie POA approach". This is quite similar to the way it works for components (where the tie POA approach is always used).

A likely scenario in which offshoots would be useful is the following:

- A component needs to create a number of objects which must be exposed individually to other clients
- These offshoot objects should run asynchronously with respect to each other and to their mother component
- Yet the mother component stays in charge of their lifetime. ACS guarantees that all offshoots are unloaded together with the component.

Warning: in some cases (e.g. operational archive), offshoots have been used as a replacement for dynamic components when the latter concept was not yet available in ACS. These should not serve as role models, but should be changed at some point.

## How to implement your own offshoot

The offshoot class must be defined as an IDL interface that inherits from ACS OffShoot:

```
interface Cursor : ACS::OffShoot
{ ... };
```

Notice that the offshoot base interface does not declare any operations. It serves only as a "marker interface" that helps avoid some wild activation of CORBA objects that could quickly mess with the way that ACS harnesses CORBA.

The IDL compiler will create the same kind of proxy and skeleton classes as it does for components, for example in Java

- `public interface CursorOperations extends alma.ACS.OffShootOperations`
- `public abstract class CursorPOA extends org.omg.PortableServer.Servant implements CursorOperations`
- `public class CursorPOATie extends CursorPOA`
- and so on for `Cursor`, `CursorHelper`, `CursorHolder`, `_CursorStub`.

Most of these classes you can ignore; what you need is described below in the example.

In your component implementation, you can instantiate the offshoot skeleton class and pass it to the method `ZLegacy/ACS.ContainerServices#activateOffShoot`. From that time on, the offshoot is a remotely visible object, and can be passed for example as the return value of the component's method that created the offshoot.

Here are some example code fragments from a component that creates the above Cursor offshoot. First we declare our offshoot class as a subclass of `CursorPOA`:

```
public class CursorImpl extends CursorPOA {
   public CursorImpl(DBCursor cursor, ContainerServices cs) {
      if (cursor == null || cs == null) {
         throw new IllegalArgumentException("DBCursor and ContainerServices must not be null.");
      }
      this.cursor = cursor;
      this.containerServices = cs;
      this.logger = containerServices.getLogger();
   }
   public void close() {
      // ... closing of db cursor and exception handling omitted here in the example ...
      // This is an example where the offshoot object deactivates itself
      // when its client no longer needs it.
      // In a different design the client would tell the mother component that the offshoot is no longer needed
      // (or the component figures this out itself), and then the component calls deactivateOffShoot.
      containerServices.deactivateOffShoot(this);
   }
```

Note that we could also have extended our offshoot class from `CursorPOATie` instead, which would cause any functional method calls on the offshoot to be intercepted by the container. In case of the Cursor offshoot we did not want this though.

Then we instantiate our offshoot class, activate the object, cast it to its correct type, and return it.

```
   CursorImpl externalCursor = new CursorImpl(internalCursor, containerServices);
   return CursorHelper.narrow(containerServices.activateOffShoot(externalCursor));
```

## Related articles

- How can more people do development with ACS on the same machine without disturbing each other?
- Which ports are used by ACS?
- Problems connecting to ACS servers on a remote machine: bad /etc/hosts
- Why does the getComponent method of ZLegacy/ACS.ContainerServices return an object of type None?
- Why are some of my print statements not showing up in the container output section of acscommandcenter?