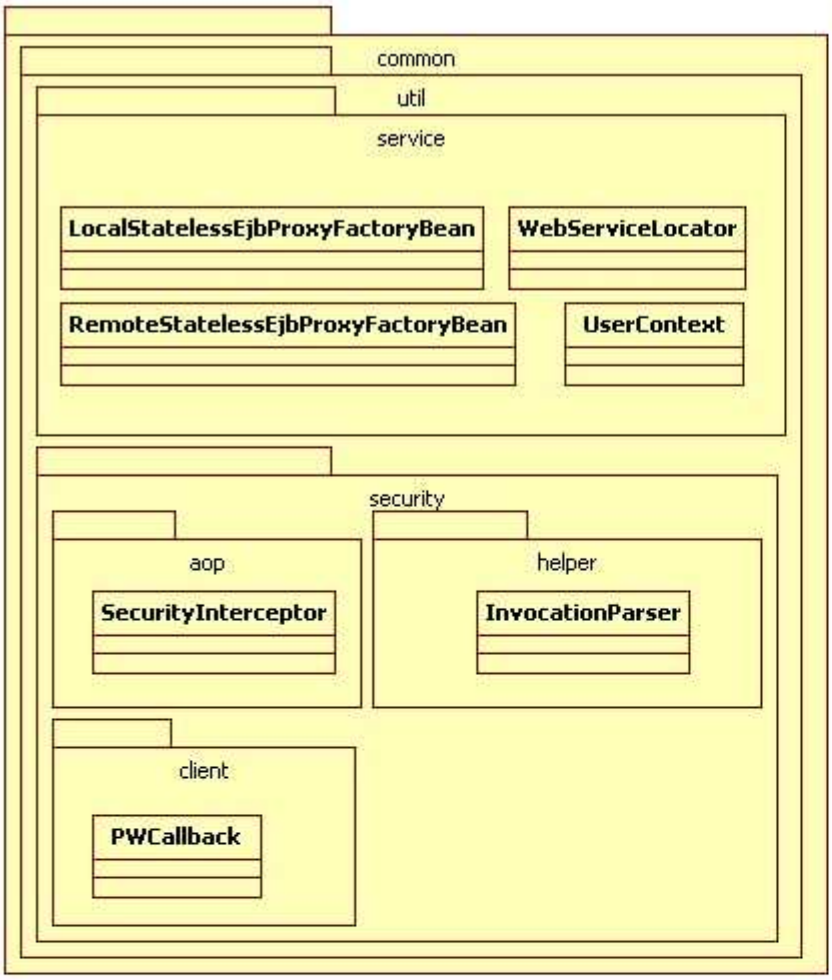
Table of Content

1 Overview

1.1 Class Diagram

Figure 1 shows all classes relevant for the Authentication / Authorisation part of the eSciDoc framework.


common
util
service
LocalStatelessEjbProxyFactoryBean
WebServiceLocator
RemoteStatelessEjbProxyFactoryBean
UserContext
security
aop
SecurityInterceptor
helper
InvocationParser
client
PWCallback

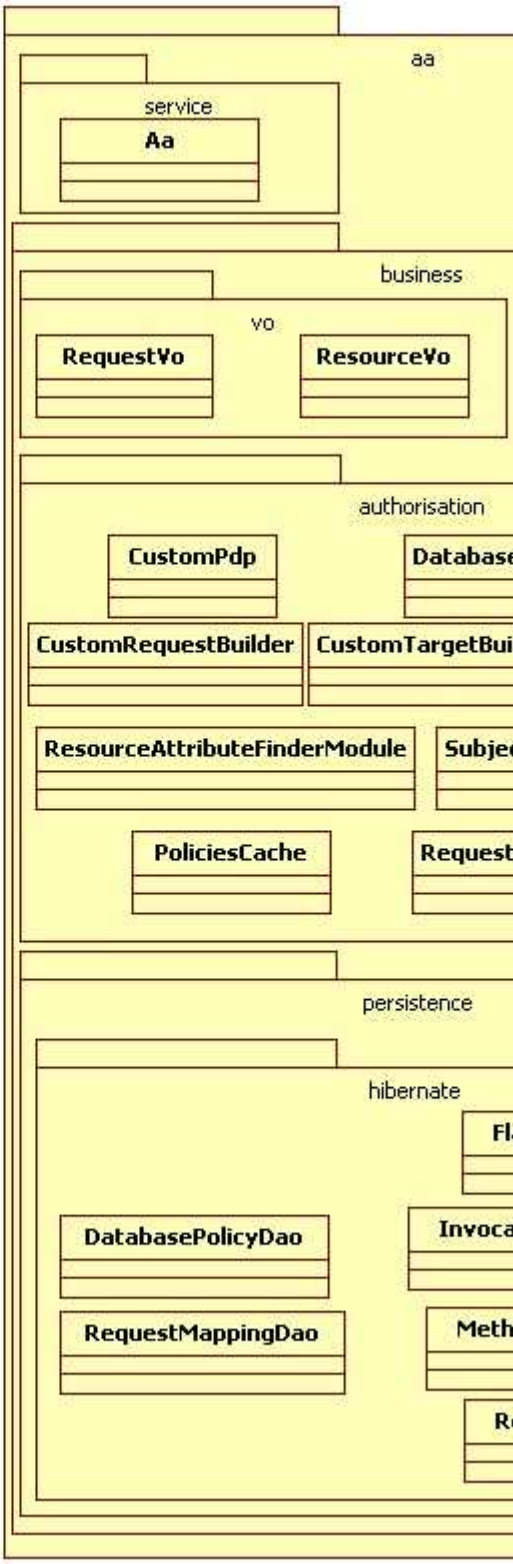

aa
service
Aa
business
vo
RequestVo
ResourceVo
authorisation
CustomPdp
CustomRequestBuilder
ResourceAttributeFinderModule
PoliciesCache
persistence
hibernate
DatabasePolicyDao
RequestMappingDao

Figure 1: Class diagram for Authentication / Authorisation

1.2 Sequence Diagrams

1.2.1 Overall Sequence

Figure 2 shows an overall view on what happens during a security-intercepted webservice call from the client. The fictive example shows the sequence diagram for a call of the webservice method "C", which is implemented in EJB "A". In order to execute, EJB "A" will eventually need to internally call method "D" on EJB "B".

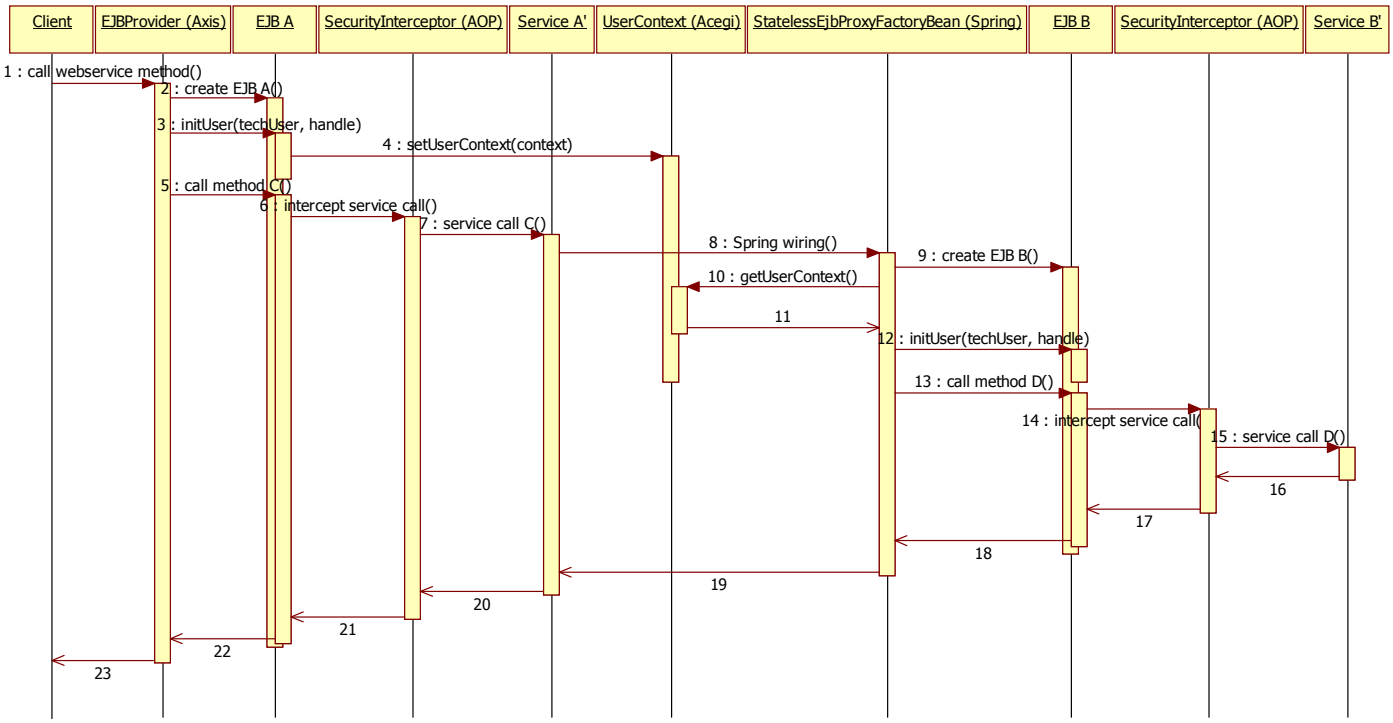


Figure 2: Authentication / Authorisation overall sequence diagram

1. **Client → EJBProvider: call webservice method:** The client (application) calls webservice method C via the WebServiceLocator. The WebServiceLocator sets the WS-Security parameters username and password. Since we are using Shibboleth for authentication, no real username has to be provided here, but only a technical username, i.e. **"ShibbolethUser"**. The password has to be set to the (previously retrieved) Shibboleth handle. **Note:** For the first release there will be no Shibboleth implementation. Instead there are a number of hardcoded handle-username-mappings set up in the system.

2. **EJBProvider → EJB A: create EJB A:** Webservice method calls are handled on the eSciDoc server using Axis. By default the Axis-class EJBProvider "creates" (the EJB will in fact not get created every time, but the application server pools a number of EJBs and dedicates one of these EJBs to the webservice call; only if no EJBs are available in the pool, a new EJB will get created) a new EJB session bean on each webservice method call.

3. **EJBProvider → EJB A: initUser(techUser, handle):** The Axis-class EJBProvider has been adapted for the eSciDoc project in the following way: Directly after the creation of the EJB the EJBProvider retrieves the WS-Security username and password from the SOAP header and calls the method `initUser` on the EJB with username and password as parameters.

4. **EJB A → UserContext: setUserContext(context):** UserContext is a custom eSciDoc class, that encapsulates the Acegi classes. In the `setUserContext` method it creates a new `org.acegisecurity.context.SecurityContext` object, filled with the username and handle (i.e. password) from the provided `context` parameter. This object is then passed to the Acegi-class `org.acegisecurity.context.SecurityContextHolder`, which internally stores the `SecurityContext` in the current thread.

5. **EJBProvider → EJB A: call method C:** Now the EJB Provider calls the actual method C on EJB A.

6. **EJB A → SecurityInterceptor: intercept service call C:** Since all EJBs in the eSciDoc framework are generated as simple wrapper-classes for the actual service classes, which implement the same interface and therefore contain the same methods, all that the EJB A does is forwarding the method invocation to service class A' via calling its method C. However, any call from an EJB to a service class is intercepted by JBoss AOP, which forwards the call to the class `SecurityInterceptor`.
   `SecurityInterceptor` implements the JBoss AOP interface `org.jboss.aop.advice.Interceptor`, therefore implementing the method `invoke`, which gets called every time an interception happens.

7. **SecurityInterceptor → Service A': service call C:** `SecurityInterceptor` checks for each intercepted method call whether the executing user is allowed to execute the business action that maps to this method. Decision criteria for this check are the input parameters of the method as well as system objects that are fetched based on the input parameters (cf. section 1.2.4 Overview for details).
   In case the decision is positive, the actual service call to method C of service class A' is executed.
   Note: In order to avoid unnecessary processing for the authentication and especially the authorization, the flag `INTERNAL_INTERCEPTION` in the SecurityInterceptor can be set to `false`. In this case any internal call to another EJB spawning from this service call will still be intercepted, but no authentication and authorization checks will be executed (cf. section 1.3.3 for details).

8. **Service A' → StatelessEjbProxyFactoryBean: spring wiring:** Now let's assume method C of service class A' needs to execute method D of EJB B in order to fulfill its functionality (**Note:** Even though this is an internal call, method C should not call method D in service

class B' directly, since this call goes to a different component which might be located on a remote server and then can only be accessed via a remote (EJB-)call).
In the eSciDoc framework an EJB call is done via the Spring wiring, which makes this call transparent for method C. Internally the call is done via one of the classes `<Local/Remote>StatelessEjbProxyFactoryBean`, which are eSciDoc-specific extensions of the Spring-classes `<Local/Remote>SlsbInvokerInterceptor`.

9. **StatelessEjbProxyFactoryBean → EJB B: create EJB B:** The `StatelessEjbProxyFactoryBean` works similar to the EJBProvider class above: As a first step, it "creates" a new EJB B (this is default behaviour via the super-class, not adapted for eSciDoc).

10. + 11. **StatelessEjbProxyFactoryBean → UserContext: getUserContext:** The `StatelessEjbProxyFactoryBean` fetches the current user's credentials (techUser, handle) from the `UserContext` class by retrieving the Acegi object `org.acegisecurity.context.SecurityContext` from the Acegi class `org.acegisecurity.context.SecurityContextHolder`, where it has been stored in step 4.
    This call has been explicitly added to `StatelessEjbProxyFactoryBean` for the eSciDoc framework.

12. **StatelessEjbProxyFactoryBean → EJB B: initUser(techUser, handle):** Now the `StatelessEjbProxyFactoryBean` calls the `initUser` method on the created EJB with the user credentials fetched in the last step.
    This call has been explicitly added to `StatelessEjbProxyFactoryBean` for the eSciDoc framework.

13. **StatelessEjbProxyFactoryBean → EJB B: call method D:** The `StatelessEjbProxyFactoryBean` calls the method D of the beforehand created EJB B.

14. **EJB B → SecurityInterceptor: intercept business call:** In the same way as EJB A in step 6 was trying to call method C on service class A', but got intercepted, is now also EJB B trying to call method D on its service class B', but also gets intercepted by the AOP interceptor `SecurityInterceptor`.

15. **SecurityInterceptor → Service B': service call D:** As described in step 7 the SecurityInterceptor class will authenticate the current user and check whether the current user is authorized to execute method D (except if `INTERNAL_INTERCEPTION` is set to `false`; in this case no further authentication and authorization checks are done; see section 1.3.3 for details).
    In case authentication and authorization (cf. section 1.2.2 for details) was successful, the call to service method D is executed.

16. **Service B' → SecurityInterceptor: return value:** In case call D has a return value, this value is returned to the `SecurityInterceptor`. If the return value is a list of objects, these objects will be filtered by the `SecurityInterceptor` (only if `INTERNAL_INTERCEPTION` is set to `true`) in order to remove objects that the user is not authorized to have access to (cf. section 1.2.2 for details).

17. – 19. **SecurityInterceptor → … → Service A': return value:** The (eventually filtered) return (list) is returned to service class A' via EJB B and via `StatelessEjbProxyFactoryBean`.

20. **Service A' → SecurityInterceptor: return value:** After executing some business logic, service class A' returns its return value to the `SecurityInterceptor`.
    Similar as in step 16 (but now also if `INTERNAL_INTERCEPTION` is set to `false`, since this was the original interception), if the return value is a list of objects, these objects will be filtered by the `SecurityInterceptor` in order to remove objects that the user is not authorized to have access to (cf. section 1.2.2 for details).

21. – 23. **SecurityInterceptor → … → Client: return value:** The (eventually filtered) return (list) is finally returned back to the client via EJB A and via `EJBProvider`.

1.2.2 Detailed Sequence: SecurityInterceptor

Figure 3 shows a detailed sequence diagram about what happens during an interception by the `SecurityInterceptor` (step 6 in section Overall Sequence).

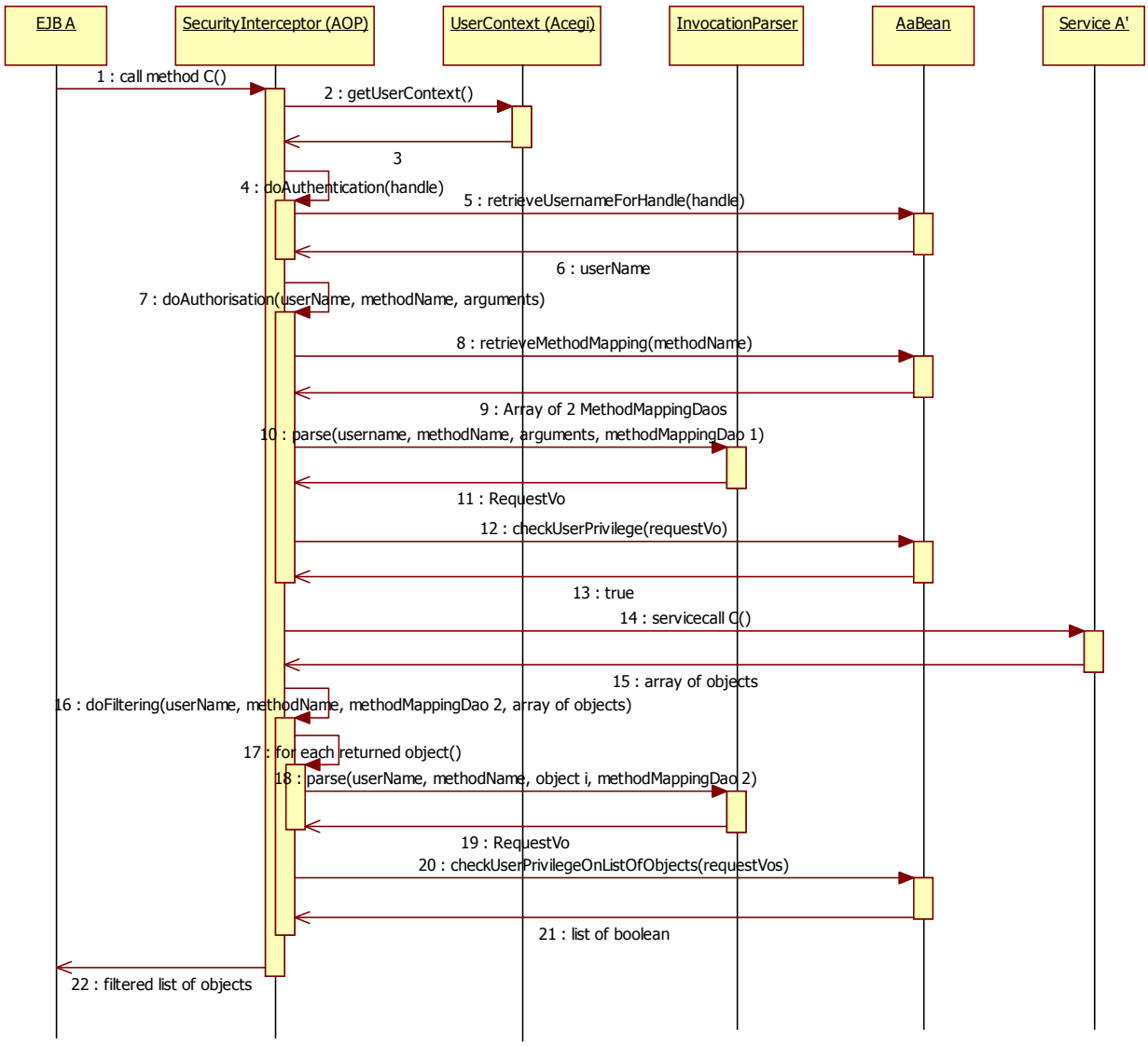


Figure 3: Detailed sequence diagram: SecurityInterceptor

1. **EJB A → SecurityInterceptor: call method C:** As described in step 6 in section Overall Sequence, each call from an EJB method to a service method is intercepted by AOP. In this case, AOP forwards to class `de.fiz.escidoc.common.util.security.aop.SecurityInterceptor`, which implements `org.jboss.aop.advice.Interceptor`, therefore its method `invoke` is executed.

2. + 3. **SecurityInterceptor → UserContext: getUserContext:** First the `SecurityInterceptor` fetches the user credentials (techUser, handle) from the `UserContext` class (details see step 10 in section 1.2.1), where it has been stored during step 4 in section 1.2.1.

4. **SecurityInterceptor → SecurityInterceptor: doAuthentication(handle):** The SecurityInterceptor calls its own private method `doAuthentication`, providing the handle fetched in step 2 as parameter.

5. + 6. **SecurityInterceptor → AaBean: retrieveUsernameForHandle(handle):** The real username for the provided handle is retrieved. See section 1.2.3 for details.

7. **SecurityInterceptor → SecurityInterceptor: doAuthorisation(userName, methodName, arguments):** The SecurityInterceptor calls its own private method `doAuthorisation`, providing
   - the (real) username, which was the return value of step 6
   - the name of the invoked method, which is C in this case
   - the list of arguments that have been passed in for method C

8. + 9. **SecurityInterceptor → AaBean: retrieveMethodMapping(methodName):** Depending on the currently called method, some of the input parameters will be checked to decide whether authorization is permitted or denied. This varies from method to method. Therefore there exists a database table, which contains information for each method, which input parameters are relevant for the authorization.
   The method `retrieveMethodMapping` from EJB AaBean retrieves this information from the database and returns it in form of the custom object `MethodMapping` (see section 1.2.3 for details).
   Note: For the filtering of return lists, similar information from the database is required. For performance reasons, this information is also fetched at this point in form of the same object type, therefore the method `retrieveMethodMapping` returns (up to) two `MethodMapping` objects, one for authorization and one for return list filtering (in case only one object is returned, this object is automatically assumed to be dedicated for the authorization; in this case the return value has to be a simple value or a single object).

10. + 11. **SecurityInterceptor → InvocationParser: parse(username, methodName, arguments, methodMapping 1):** This step calls method `parse` of the helper class `InvocationParser` (also included in the common package), which assembles all information that is required as input parameter for step 12 to decide on the authorization for the current method call.
    This information includes
    - the (real) username of the current user
    - the business action that maps to the currently called method
    - any information about the to-be-accessed resource (i.e. the system object that this method call deals with) that can be directly taken from the input parameters (all further required information about resource or user, which is not contained in the input parameters, will be fetched directly from the system by the XACML engine; see section 1.2.4 for details).
    The information is returned as an object of custom type `RequestVo`.

12. + 13. **SecurityInterceptor → AaBean: checkUserPrivilege(requestVo):** The `RequestVo` object assembled in the last step is now used as input parameter for method `checkUserPrivilege` of the `AaBean`, which
    - builds an XACML request from the information in the `RequestVo`,
    - sends the request to its internal XACML engine to decide based on the currently existing XACML policies

- returns true or false depending on the outcome of the XACML response
(see section 1.2.4 for details)

14. + 15. **SecurityInterceptor → Service A': service call C:** In case the authorization check in step 12 was successful (i.e. return value in step 13 is `true`), the method C of service class A' is invoked.
If the invoked method has a return value (as is the case in our example for method C, which in fact returns an array of objects as result value), this return value is returned to the `SecurityInterceptor`.

16. **SecurityInterceptor → SecurityInterceptor: doFiltering(userName, methodName, methodMapping 2, array of objects):** If the return value returned in step 15 is an array of objects (as in our example), this array has to be filtered in order to remove those objects, which the user is not authorized to access.

17. **SecurityInterceptor → SecurityInterceptor: for each returned object:** The object filtering is done by invoking the XACML engine for each of the returned objects.

18. + 19. **SecurityInterceptor → InvocationParser: parse(userName, methodName, object i, methodMapping 2):** Like in step 10 the `parse` method of helper class `InvocationParser` will be invoked for each returned object i, which assembles all information that is required to decide on the access authorization for object i for the current user and the current method. Now the second `MethodMapping` fetched in step 9 is used. Like in step 10 the return value for each invocation of `parse` is a `RequestVo` object.

20. + 21. **SecurityInterceptor → AaBean: checkUserPrivilegeOnListOfObjects(requestVos):** All `RequestVo` objects created during steps 17 – 19 are now sent as input parameter to method `checkUserPrivilegeOnListOfObjects` of EJB `AaBean`.

    Similar as in step 12 the `AaBean`
    - builds an XACML request from the information in each `RequestVo`,
    - sends the requests to its internal XACML engine to decide based on the currently existing XACML policies
    - returns true or false for each request depending on the outcome of the XACML response

    The decision of the XACML engine for each `RequestVo` is returned by the `AaBean` as an array of boolean, which has the same size and order as the input parameter array of `RequestVo` objects, so each boolean maps to exactly one `RequestVo`.

22. **SecurityInterceptor →EJB A: filtered list of objects:** Based on the array of boolean received in step 21 the `SecurityInterceptor` filters the returned array of objects of service method call C (received in step 15), i.e. in case boolean at position X is true, the object, which was originally at position X in the returned array, remains in the filtered list, otherwise it is removed from the list.
    The filtered list is finally returned back to the EJB A.

1.2.3 Detailed Sequence: AaBean Methods

Figure 4 shows the details of the methods of AaBean, which are invoked during execution of the SecurityInterceptor.

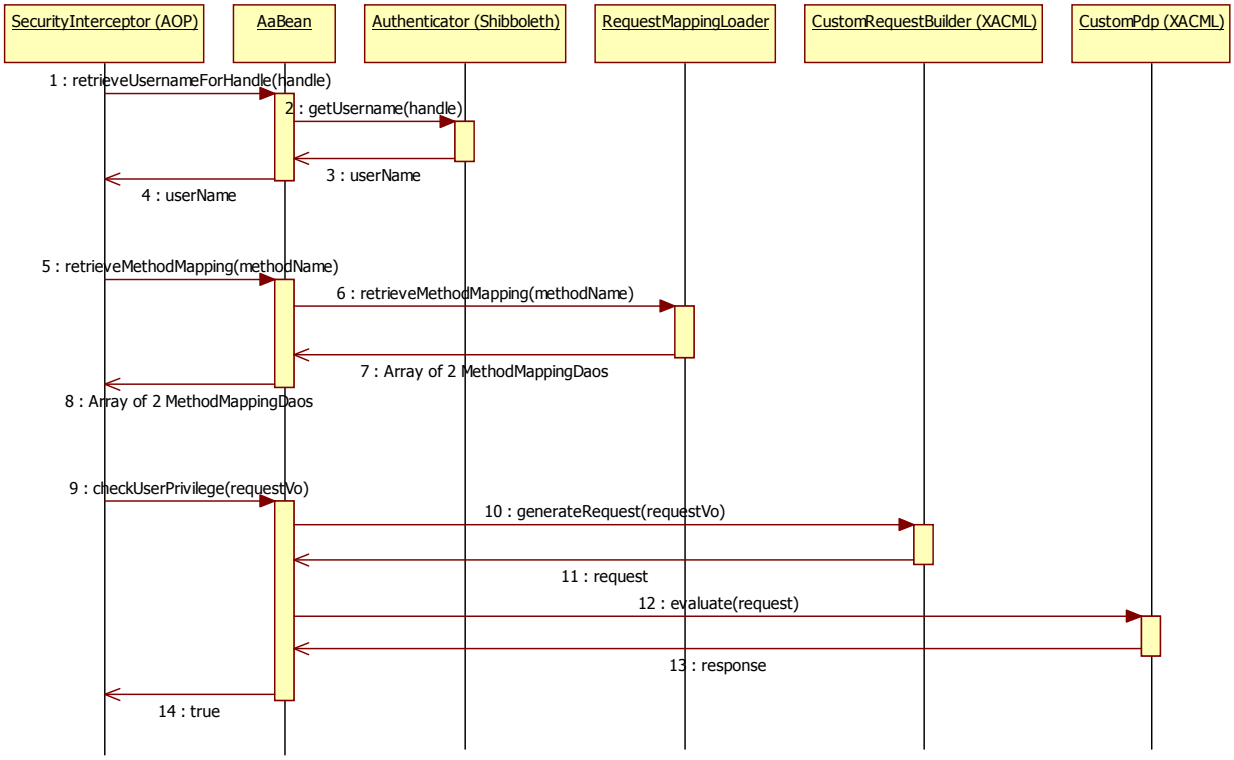


Figure 4: Detailed sequence diagram: Method calls in AaBean

1. **SecurityInterceptor → AaBean: retrieveUsernameForHandle(handle):** During the authentication process, the `SecurityInterceptor` calls `retrieveUsernameForHandle` of `AaBean`, providing the handle fetched previously from `UserContext`. This is the same call as described in step 5 in 1.2.2.

2. - 4. **AaBean → Authenticator: getUsername(handle):** The AaBean passes on the handle to the `getUsername` method of class `Authenticator`. For the first release, a number of handle-username-mappings are hardcoded in this class.
   In a later release this class should be replaced by a Shibboleth Service Provider implementation, which retrieves the username (and probably some more useful user attributes) from a Shibboleth Identity Provider.
   The returned username is handed back via the `AaBean` to the `SecurityInterceptor`.

5. **SecurityInterceptor → AaBean: retrieveMethodMapping(methodName):** During the authorization process the `SecurityInterceptor` calls `retrieveMethodMapping` of `AaBean`, provided the currently called methodName. This is the same call as described in step 8 in 1.2.2.

6. - 8. **AaBean → RequestMappingDao: retrieveMethodMapping(methodName):** The call is forwarded to the database persistence class `RequestMappingDao`. This class fetches all information needed to transform a method call into an XACML request from the database (for table definitions see Appendix A).

   The return value of `retrieveMethodMapping` is an array of up to two `MethodMapping` objects,
   - one for the authorization check before the method invocation and
   - one for the return list filtering after the method invocation.

   This object is handed back via the `AaBean` to the `SecurityInterceptor`.

9. **SecurityInterceptor → AaBean: checkUserPrivilege(requestVo):** During the authorization process, the `SecurityInterceptor` calls `checkUserPrivilege` of AaBean, providing the `RequestVo` assembled before by the `InvocationParser`. This is the same call as described in step 12 in 1.2.2.

10. + 11. **AaBean → CustomRequestBuilder: generateRequest(requestVo):** The `RequestVo` object is forwarded to method `generateRequest` of class `CustomRequestBuilder`, which uses the API of Sun's XACML implementation to generate an XACML-specific `com.sun.xacml.ctx.RequestCtx` object from the information provided in the `RequestVo` object.

12. + 13. **AaBean → CustomPdp: evaluate(request):** `CustomPdp` is a custom implementation of an XACML Policy Decision Point (PDP). Its method `evaluate` is invoked with the generated request object from the previous step to decide on the permission or denial of the request, based on the current XACML policies (see section 1.2.4. for details).
    The return value is a Sun XACML specific object of type `com.sun.xacml.ctx.ResponseCtx`.

14. **AaBean → SecurityInterceptor: return true or false:** The response object is interpreted and then, according to the result, `true` or `false` is returned to the `SecurityInterceptor`.
    In our example the access of method C is allowed, therefore the return value is `true`.

1.2.4 Detailed Sequence: XACML process

Figure 5 shows the detailed sequence that happens when the method `evaluate` of class `CustomPDP` is executed.

Please note that most of the classes described here are custom built classes, which are hooked into the Sun XACML engine. Therefore in some cases no direct invocation can be found in the source code. Nevertheless these calls exist and happen via (not-changed Sun XACML-) super-classes of the classes described here.

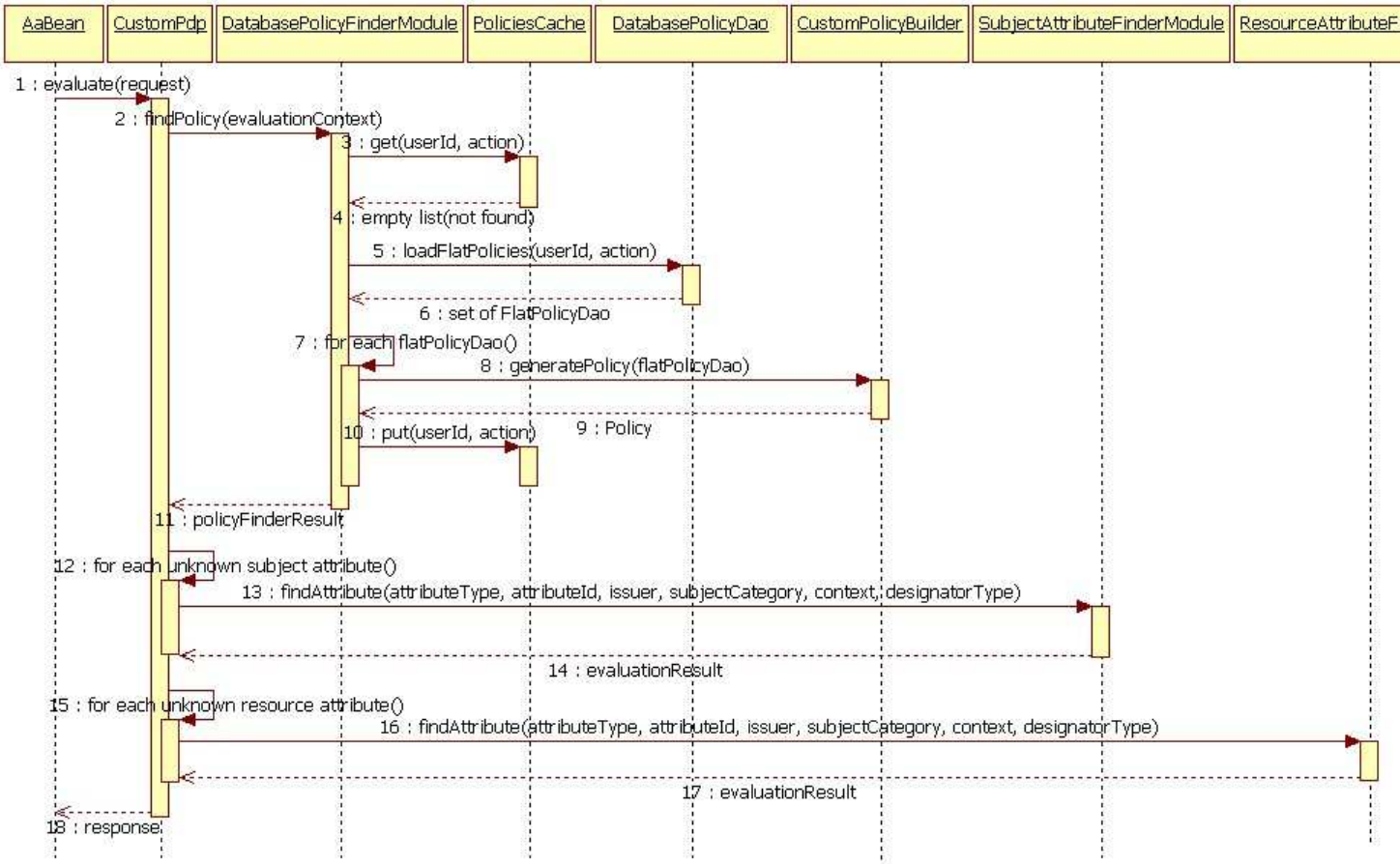


Figure 5: Detailed Sequence diagram: The XACML process

1. **AaBean → CustomPdp: evaluate(request):** A call to `CustomPdp`'s `evaluate` method with a valid XACML request object triggers the XACML request evaluation process. This step corresponds to step 12 in section 1.2.3.

2. **CustomPdp → DatabasePolicyFinderModule: findPolicy(evaluationContext):** The first step of the evaluation process is to find suitable XACML policies for the current request, which is now contained in an object of type `com.sun.xacml.EvaluationCtx`. This method has been explicitly designed for the eSciDoc project in order to evaluate as few policies as possible. The policies are stored in the database (thus the name *Database*PolicyFinderModule). However, only those policies are fetched from the database, which apply to the current user and business action. Therefore the user ID and action are retrieved from the `evaluationContext` first of all.

3. + 4. **DatabasePolicyFinderModule → PoliciesCache: get(userId, action):** Since fetching policies from the database and building the XACML policy objects from the information pieces fetched from the database are very performance-intensive tasks, a PolicyCache exists where recently retrieved and built policies are cached. This cache is consulted for

policies for the current user ID and action.
Only if no policies are found in the cache (i.e. an empty list is returned), step 5 is executed.

5. + 6. **DatabasePolicyFinderModule → DatabasePolicyDao: loadFlatPolicies(userId, action):** Since no policies are found in the cache in step 3, any policies defined for the current user ID and action have to be fetched from the database (see Appendix A for table definitions).
   The policies are returned as a set of `FlatPolicy` objects.

   Note: A *flat* policy object is an object that contains all information needed to build an XACML policy object as a flat list of properties.

7. **DatabasePolicyFinderModule → DatabasePolicyFinderModule: for each flatPolicy:** Steps 8 – 10 are executed for each flat policy object retrieved in step 5.

8. + 9. **DatabasePolicyFinderModule → CustomPolicyBuilder: generatePolicy(flatPolicy):** XACML policy objects have to be created from the flat policy objects retrieved in step 5, so the XACML engine understands the policies. This is done via the custom built helper class `CustomPolicyBuilder` and its subordinate class `CustomTargetBuilder` (not shown in diagram). The return value of `generatePolicy` is an object of type `com.sun.xacml.Policy`.

10. **DatabasePolicyFinderModule → PoliciesCache: put(userId, action):** The policy built during step 8 is stored in the policy cache, so it is directly available to following calls without having to fetch it again from the database.

11. **DatabasePolicyFinderModule → CustomPdp: return policyFinderResult:** All policies that are potentially matching with the current request are returned to the `DatabasePolicyFinderModule` via an object of type `com.sun.xacml.finder.PolicyFinderResult`.

12. **CustomPdp → CustomPdp: for each unknown subject attribute:** Now the `CustomPdp` starts to evaluate the policies retrieved in the previous steps. However, some of the attributes required to decide whether a certain policy evaluates to permit or deny are probably not contained within the XACML request. In this case these attributes have to be fetched directly from the corresponding objects in the eSciDoc system.
    First all attributes concerning the subject are handled. For access to these attributes, the custom class `SubjectAttributeFinderModule` has been designed, which extends the abstract Sun XACML-specifc class `com.sun.xacml.finder.AttributeFinderModule` and which is hooked into the `CustomPdp`.

13. + 14. **CustomPdp → SubjectAttributeFinderModule: findAttribute(attributeType, attributeId, issuer, subjectCategory, context, designatorType):** For each subject attribute (i.e. information about the user like her affiliation) that is required to decide on a specific policy, but which cannot be found in the request, the `findAttribute` method of the `SubjectAttributeFinderModule` is invoked. This method first retrieves information about the user that *is* available in the request (normally the user ID) and then accesses the various eSciDoc system components (e.g. Organizational Unit Manager, User

Management, etc.) to fetch the missing subject attribute.
The retrieved attribute is returned in form of a Sun XACML specific `com.sun.xacml.cond.EvaluationResult` object.

15. **CustomPdp → CustomPdp: for each unknown resource attribute:** Now all missing attributes concerning the resource are handled. For access to these attributes, the custom class `ResourceAttributeFinderModule` has been designed, which also extends the abstract Sun XACML-specifc class `com.sun.xacml.finder.AttributeFinderModule` and which is also hooked into the `CustomPdp`.

16. + 17. **CustomPdp → ResourceAttributeFinderModule: findAttribute(attributeType, attributeId, issuer, subjectCategory, context, designatorType):** Similar as in step 13, for each resource attribute (i.e. information about the currently accessed resource) that is required to decide on a specific policy, but which cannot be found in the request, the `findAttribute` method of the `ResourceAttributeFinderModule` is invoked. This method first retrieves information about the resource that *is* available in the request (normally the resource ID) and then accesses the various eSciDoc system components (e.g. Object Manager, Basket Manager, etc.) to fetch the missing resource attribute.
The retrieved attribute is also returned in form of a Sun XACML specific `com.sun.xacml.cond.EvaluationResult` object.

    Note: Since attributes are fetched one-by-one, but in many cases more than one attribute are to be fetched from the same system object, fetched system objects are cached after the first attribute retrieval in a temporary cache for the lifetime of the request evaluation. This is realized by the custom class `RequestAttributesCache` (not shown in diagram).

18. **CustomPdp → AaBean: return response:** After all applicable policies have been evaluated, the result is finally returned back to the `AaBean`.
The return value is a Sun XACML specific object of type `com.sun.xacml.ctx.ResponseCtx`.
This step corresponds to step 13 in section 1.2.3.

1.3 Technical Concepts

1.3.1 Mapping Input Parameters → XACML Request

The XACML request is built based on the current user, the currently invoked webservice method and its input parameters.

The information how to assemble this information into an XACML request is kept in an `MethodMapping` (and its contained `InvocationMappings`), which is fetched from database table `method_mappings` (and its dependent table `invocation_mappings`; cf. Appendix A for table definitions).

The `MethodMapping` only contains information about to which XACML business action the current webservice method maps and whether the contained `InvocationMappings` are to be used

- for authorization before the method invocation (i.e. property `beforeAfter` has value `1`) or
- for filtering of a return list after the method invocation (i.e. property `beforeAfter` has value `2`). This of course only applies if the method is in fact returning a list of objects.

The XACML request will look as follows:

```
<Request>
  <Subject>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
               DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>[current user]</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    [see below]
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
               DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>[action]</AttributeValue>
    </Attribute>
  </Action>
</Request>
```

The `InvocationMapping` objects can be set up in three different mapping types. These three types are described in the following three subsections, with a description, how to get from the `InvocationMapping` to the XACML request part.

1.3.1.1 Simple Attribute Mapping

This mapping is used if the value to be retrieved from the input parameter is simply one of these parameters itself, which has to be a simple type (e.g. int, long, etc.) or a String.

The properties of the `InvocationMapping` have in this case to be set as in the following example:

Field	Value	Comment
mappingType	1 (= InvocationMapping. SIMPLE_ATTRIBUTE_MAPPING)	This is a simple mapping
id	urn:oasis:names:tc:xacml:1.0: resource:resource-id	The XACML name of the attribute
attributeType	http://www.w3.org/2001/ XMLSchema#string	The XACML type of the attribute
path	<empty>	No path needed
position	3	The value is the third parameter of the invoked webservice method (this value only applies if

		the mapping is used for a “before-request”; for a “after-request” this value has to be set to 0)
value	<empty>	No value needed

This could for example map to the following XACML snippet:

```
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
           DataType="http://www.w3.org/2001/XMLSchema#string">
  <AttributeValue>123</AttributeValue>
</Attribute>
```

1.3.1.2 Complex Attribute Mapping

This mapping is used if the value to be retrieved from the input parameter is to be fetched from inside one of these parameters, which in this case has to be a JavaBean.

The parameters of the `InvocationMapping` have in this case to be set as in the following example:

Field	Value	Comment
mappingType	2 (= InvocationMapping. COMPLEX_ATTRIBUTE_MAPPING)	This is a complex mapping
id	urn:escidoc:names:resource:content-item:admin-group-id	The XACML name of the attribute
attributeType	http://www.w3.org/2001/ XMLSchema#string	The XACML type of the attribute
path	getAdminGroupId	From the fetched input parameter, the method “getAdminGroupId” should be invoked to fetch the value.
position	2	The value is the second parameter of the invoked webservice method
value	<empty>	No value needed

In this case, the object at the defined `position` is fetched and the `path` is used to fetch the value inside this object. In this example, on the object at position 2 the method `getAdminGroupId` is invoked, which will return a simple value to be used in the request.

In case the navigation should be more than one level down, the path has to have the following syntax: “`<getterFirstObject>:<getterIncludedObject>:…`”.

This could for example map to the following XACML snippet:

```
<Attribute AttributeId="urn:escidoc:names:resource:content-item:admin-group-id"
           DataType="http://www.w3.org/2001/XMLSchema#string">
  <AttributeValue>1</AttributeValue>
</Attribute>
```

1.3.1.3 Value Attribute Mapping

This mapping is used if the value to be included in the XACML request is not contained in the input parameters, but will always be the same for the provided method and therefore can be hardcoded in the database.

The parameters of the `InvocationMapping` have in this case to be set as in the following example:

Field	Value	Comment
mappingType	3 (= InvocationMapping. VALUE_ MAPPING)	This is a value mapping
id	urn:escidoc:names:resource:object-type	The XACML name of the attribute
attributeType	http://www.w3.org/2001/ XMLSchema#string	The XACML type of the attribute
path	<empty>	No path needed
position	0	No position needed
value	content_item	The value to be used

This could for example map to the following XACML snippet:

```
<Attribute AttributeId="urn:escidoc:names:resource:object-type"
           DataType="http://www.w3.org/2001/XMLSchema#string">
  <AttributeValue>content_item</AttributeValue>
</Attribute>
```

1.3.2 Mapping Database values → XACML Policy

Tables `flatpolicies` and `resources`, defined in Appendix A, are used to build an XACML policy in the following way (values that come from the tables are in brackets and printed in red):

```
<Policy PolicyId="[policy_id]"
   RuleCombiningAlgId="[rule_alg_id]">
<Target>
        <Subjects>
           <Subject>
              <SubjectMatch MatchId="[policy_subject_match_id]">
                 <AttributeValue
                    DataType="[policy_subject_designator_type]">
                    [policy_subject_value]
                 </AttributeValue>
                 <SubjectAttributeDesignator
                    AttributeId="[policy_subject_designator_id]"
                    DataType="[policy_subject_designator_type]"/>
              </SubjectMatch>
           </Subject>
        </Subjects>
           <Resources>
             <AnyResource/>
           </Resources>
           <Actions>
             <Action>
```

```
                <ActionMatch MatchId="[policy_action_match_id]">
                  <AttributeValue DataType="[policy_action_designator_type]">
                    [policy_action_value]
                  </AttributeValue>
                  <ActionAttributeDesignator
                    DataType="[policy_action_designator_type]"
                    AttributeId="[policy_action_designator_id]"/>
                </ActionMatch>
              </Action>
          </Actions>
</Target>
   <Rule RuleId="[rule_id]" Effect="Permit">
      <Target>
         <Subjects>
            <AnySubject/>
         </Subjects>
            <Resources>
               <Resource>
                 <ResourceMatch MatchId="[resource_match_id]">
                   <AttributeValue DataType="[resource_designator_type]">
                    [resource_value]
                  </AttributeValue>
                   <ResourceAttributeDesignator
                    DataType="[resource_designator_type]"
                       AttributeId="[resource_designator_id]"/>
                 </ResourceMatch>
                  ...
               </Resource>
            </Resources>
            <Actions>
               <AnyAction/>
            </Actions>
      </Target>
          [condition]
   </Rule>
</Policy>
```

1.3.3 Internal Interception Mode

As explained before, any call between an EJB and its service class is intercepted by the `SecurityInterceptor`, which leads to authentication, authorization and filtering. This applies for calls coming as a webservice call via Axis as well as internal calls from one component to another component (Note: Since it should be possible to distribute components on different machines, calls between components should always happen across the EJB of the target component.).

However, it might be desireable to avoid the authentication, authorization and filtering on internal calls, since

- the user has already been authenticated,
- s/he is authorized for the original method invocation and
- filtering will happen anyway on return of the original method.

This can be achieved by setting flag `INTERNAL_INTERCEPTION` in class SecurityInterceptor to `false`. The result of this flag is, that once the authentication and authorization have passed, the technical user will be set from “ShibbolethUser” to “internal”. Each further interception will not execute its authentication, authorization and filtering if it finds the technical user set to “internal”.

In order to prevent that an external call is trying to bypass the authentication, authorization and filtering by providing “internal” as technical user, the Axis class `EJBProvider` has been adapted, so that it now checks whether the user is set to “internal” and in this case sets username and password to “” (empty string), which will finally result in denial of the method invocation.

Note: In order to make sure that an external system cannot bypass the security checks by invoking an EJB directly, providing “internal” as technical username, it has to be made sure that all ports for remote access to the EJBs are closed externally.

Besides, it has to be made sure that the policies checked in the “outer” interception are strong enough to also secure the – now not secured – internal calls.