Abstract: In this document we give an overview of the web service development process in the JRA1 Data Management cluster.

EGEE ("Enabling Grids for E-science in Europe") is a project funded by the European Union. For more information on the project, its partners and contributors please see http://www.eu-egee.org.

You are permitted to copy and distribute verbatim copies of this document containing this copyright notice, but modifying this document is not allowed. You are permitted to copy this document in whole or in part into other documents if you attach the following reference to the copied elements: “Copyright © 2004. Members of the EGEE Collaboration. http://www.eu-egee.org”

The information contained in this document represents the views of EGEE as of the date they are published. EGEE does not guarantee that any information contained herein is error-free, or up to date.

EGEE MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, BY PUBLISHING THIS DOCUMENT.

Document Change Log

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Comment</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>12/02/2005</td>
<td>First Version</td>
<td>Akos Frohner</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

1. SERVICE ORIENTED ARCHITECTURE 4
   1.1. SERVICES .................................................. 4
   1.2. MESSAGES .................................................. 5
   1.3. POLICIES .................................................. 5
   1.4. STATE ..................................................... 5

2. OVERVIEW THE DEVELOPMENT .................................. 6

3. DESIGN MODEL .................................................. 6
   3.1. EXAMPLE ................................................... 7

4. DOCUMENTATION ............................................... 9
   4.1. API DOCUMENTATION ....................................... 9
   4.2. EXAMPLE .................................................. 10

5. INTERFACE ..................................................... 12
   5.1. WS-I COMPLIANCE ........................................... 13
   5.2. EXAMPLE .................................................. 13

6. JAVA CLIENTS AND SERVICES ................................ 14
   6.1. CLASSES OF A REMOTE METHOD CALL ....................... 15
   6.2. AXIS CLIENTS AND SERVICES ............................. 15
   6.3. EXAMPLE .................................................. 16
      6.3.1. SERVER SIDE ....................................... 16
      6.3.2. CLIENT SIDE ....................................... 17

7. C, C++ CLIENTS AND SERVICES ................................ 18

8. PERL .......................................................... 18
   8.1. EXAMPLE .................................................. 18

9. JAVASCRIPT ................................................... 20
   9.1. EXAMPLE .................................................. 21

10. CUSTOMIZATION TECHNIQUES ................................. 22
1. **SERVICE ORIENTED ARCHITECTURE**

Traditionally, applications have been built for a single computational entity, by integrating local system services like file systems and device drivers. Since everything is under local control, this model is very flexible in providing access to a rich set of development resources and provides precise control over how the application behaves. At the same time, this is bound to a single operating system and architecture, which can be error prone and costly, especially for upgrades. For distributed communities, it is not always possible to work in this mode with a single supercomputer as the computing entity.

A more modern approach is to construct complex distributed applications by integrating existing applications and services across the network, adding data entities, facades and business logic. The aim is to reduce development time and increase productivity and software quality. This architectural model is powerful in the sense that it is very flexible and extensible, providing added functionality to the users. However, subsequent modification and architectural reuse of the components may be problematic as it may have complex repercussions on services built on top of them.

The term Service Oriented Architecture (SOA) is increasingly used to refer to a discipline for building reliable distributed systems that deliver application functionality as services with the additional emphasis on loose coupling between interacting services [2, 3]. Tight coupling makes it hard for applications to adapt to changing requirements, as each modification to one application may force developers to make changes in other connected applications.

A SOA significantly increases the abstraction level for code re-use, allowing applications to bind to services that evolve and improve over time without requiring modification to the applications that consume them. Services provide a clean model to integrate software systems both inside the organisation and across organisational boundaries. This model is clearly very suitable for Grid middleware, which has to deal with Virtual Organisations and Site policies.

The services communicate with each other through well-defined interfaces and protocols, which can involve simple data passing or some coordination of activities.

1.1. **SERVICES**

If a service-oriented architecture is to be effective, we need a clear understanding of the term service. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. A Web service is an application that exposes its features programmatically using standard Internet protocols.

In more detail, services are discrete units or modules of application logic that expose message-based interfaces suitable for access across a network. Typically, services provide both the semantics and the state management relevant to the problem they address. When designing services, the goal is to effectively abstract and encapsulate the logic and data associated with real-world processes, making intelligent choices about what to include and what to implement as separate services. Well-written services expose a semantically simple logic to the application that binds to them, with clean failure modes.

In a distributed environment, services tend to be oriented toward use over a network by other services, though this is not an absolute requirement. Communicating in a distributed system is intrinsically slower and less reliable than when operating in the same processing environment. This has important architectural implications because distributed systems require that developers (of infrastructure and applications) consider the unpredictable latency of remote access, concurrency issues, possibility of partial failure, and inaccessibility of certain services.

---

1. Provide a unified interface to a set of interfaces in a subsystem. A facade defines a higher-level interface that makes the subsystem easier to use. [1]
1.2. Messages

Services interact by exchanging messages. Ultimately, every service is defined purely by the messages it will accept and produce, and what happens on failures. The routing of messages between services is a complex process that is best handled by a common messaging infrastructure.

Messages are sent in a platform-neutral, standardised format defined by the service interfaces. Service-to-service communication follows the interface contract; by making this contract explicit it is possible to change one service implementation without compromising the interaction. The internal structure of a service, including features such as its implementation language are, by design, abstracted away in the SOA.

For Web services, the Simple Object Access Protocol (SOAP) standard [4] is used to specify how the messages are being passed between services, and the Web Service Definition Language (WSDL) is used to specify the interface a service exposes. From a WSDL document a client or application will know how to bind to a service. Many semantic details are only defined as part of the internal logic and cannot be programmatically exposed; these details must be thoroughly documented.

1.3. Policies

All services are also governed by policies. Policies are less static than business rules and may be regional, organisational or user-specific. Policies are usually applied at run-time.

Policies may represent security, quality-of-service, management, and application concerns. Services negotiate using policies. Services must comply with one another’s policy requirements in order to interoperate.

In a Grid the responsibilities for the management of the service policies is shared between many organisations and sites. Policies are typically managed by the Virtual Organisations (VO) using the service and the site administrators of the site where the service is actually deployed.

The breadth of the policy management rules needs to be considered at the design phase of the project as later changes to the system are difficult. If the policy-enforcement infrastructure is well-integrated, it allows setting, changing, and evolving the run-time rules that govern communication and service behaviour easily.

1.4. State

Services manage state, ensuring through their logic that the state is kept consistent and accurate. State manipulation is governed by the internal semantics of the service, also called business rules. These are relatively stable algorithms and are typically implemented as part of the service application logic.

Almost all services manage durable state; that is, state that is stored on some durable medium such as a file system or in a database. The services receive a request from another service, retrieve some state from that durable medium, and build a response or update the state. This durable state is important; services may be brought down and when they are brought up again, the durable state is still there and they can continue as if nothing has happened. Services do their best to keep that durable state consistent; they would like to keep their application state in memory consistent as well, but if something happens, they can just abort the processing, forget their memory state, and set up again using the durable state.

So in summary, a complete definition of services might be: Services are network-capable units of software that implement logic, manage state, communicate via messages, and are governed by policy.
2. **Overview of the Development**

The EGEE JRA1 Data Management cluster follows the following process of designing, documenting and developing web services and client side libraries, applications.

The starting point is the design UML model, where the static model (classes and their documentation) is directly stored as Java source files. This model is the basis of every further step in the process.

From the design model the documentation and the interface description, in Web Services Description Language [WSDL] format, is directly generated. Later the WSDL description is used to generate client and server side code for the following languages: Java, C, C++, Perl and JavaScript.

3. **Design Model**

The design model is maintained as a UML model, where the static model information is stored as Java source files. The was made possible by the choice of the *Together Control Center* as the UML modeler tool.

![Figure 1: Overview of the Process with the Design Model](image)

The advantage of having the static model stored as Java sources is that one can use a wide variety of tools (see Figure 1) to post-process this information.

It also has the advantage that smaller changes, customizations can be edited in the Java source files directly, with any available editor.
3.1. Example

We will follow the stages of the process via a simple example interface, the File Authorization Service, which has three methods (but inherits some more) and uses some complex types for communication (see Figure 2).

![File Authorization Service Class Diagram](image)

**Figure 2:** File Authorization Service Class Diagram

The source code or the static model information for the FASBase class is the following:

```java
package org.glite.data.catalog.service.fas;

import org.glite.data.catalog.service.AuthorizationException;
import org.glite.data.catalog.service.ExistsException;
import org.glite.data.catalog.service.InternalException;
import org.glite.data.catalog.service.InvalidArgumentException;
import org.glite.data.catalog.service.PermissionsEntry;
import org.glite.data.catalog.service.ServiceBase;

/**
 * The middleware level API for File Authorization Service.
 * @stereotype simple - interface
 */

public interface FASBase extends ServiceBase {
    /**
     * Sets full set of permissions {BasicPermission,ACL} for a given item
     * (GUID, LFN or SchemaName).
     * @param permissions - list of {item, permission} pairs.
     * The PermissionsEntry PermissionEntry object takes the item and a list of ACLs. This method sets the full
     * permission, replacing all existing ACLs.
     * @throws AuthorizationException No access right to update the permissions.
     * @throws NotExistsException The item does not exist.
     * @throws InvalidArgumentException Some part of the argument is invalid.
     * @throws InternalException
    */
```

INFSO-RI-508833
<dt>
<br><b>Semantic description of the method:</b></dt>
<br>The following steps are performed for each entry:
<ul>
<li>Check format of item (depends on the context);
  <br>if it is invalid, throws
  <br>[@link org.glite.data.catalog.serviceInvalidArgumentException].</li>
<li>Check if the item exists;
  <br>if it does not exist, throws
  <br>[@link org.glite.data.catalog.serviceNotExistsException].
</li>
<li>Check PERMISSION permission on the item;
  <br>if permission is not granted, throws
  <br>[@link org.glite.data.catalog.serviceAuthorizationException].</li>
<li>Updates permission details in the catalog.</li>
</ul>
/*
public void setPermission (PermissionEntry[] permissions)
  throws InternalException, AuthorizationException,
           NotExistsException, InvalidArgumentException;

/**
 * Retrieves all set permissions for any kind of item
 * (GUID, LFN or SchemaName).
 *<p>
 * @param items list of items we want permission for
 * @return returns item with corresponding permission
 *<p>
 * @throws AuthorizationException No access right to get the permission information.
 * @throws NotExistsException The item does not exist.
 * @throws InvalidArgumentException Some part of the argument is invalid.
 * @throws InternalException
 *</p>
<br><b>Semantic description of the method:</b></dt>
<br>The following steps are performed for each entry:
<ul>
<li>Check format of item (depends on the context);
  <br>if it is invalid, throws
  <br>[@link org.glite.data.catalog.serviceInvalidArgumentException].</li>
<li>Check if item exists;
  <br>if it does not exist, throws
  <br>[@link org.glite.data.catalog.serviceNotExistsException].
</li>
<li>Check LIST permission on the item;
  <br>if permission is not granted, throws
  <br>[@link org.glite.data.catalog.serviceAuthorizationException].
</li>
<li>Query permission details from the catalog and add it to the list to be returned.</li>
</ul>
/*
public PermissionEntry[] getPermission (String[] items)
  throws InternalException, AuthorizationException,
           NotExistsException, InvalidArgumentException;

/**
 * Checks if the current user has the required permission bits on the specified items.
 *<p>
 * @param items list of items we want to check permission of
 * @param perm the permission bits to be checked against
 */
∗ @throws AuthorizationException No access right to check permission information
∗ <!−− @throws NotExistsException The item does not exist
∗ @throws InvalidArgumentException Some part of the argument is invalid.−−>
∗ @throws InternalException
∗
∗ <dt>
∗ <br> &lt;b&gt;Semantic description of the method:&lt;/b&gt;
∗ <br>The following steps are performed for each item entry:
∗ <ul>
∗ <li>Check PERMISSION permission on the item;
∗ <br>if permission is not granted, throws
∗ { @link org.glite.data.catalog.service.AuthorizationException }.
∗ <li>Check required permission bits on the item;
∗ <br>if any of the permission bits is not verified, throws
∗ { @link org.glite.data.catalog.service.AuthorizationException }.
∗</ul>
∗
∗</dt>
public void checkPermission( String [] items, Perm perm) throws InternalException , AuthorizationException ;
}

The full source of Figure 2 is longer, including the base interfaces, exceptions and argument types.

4. DOCUMENTATION

The design and API documentations are generated from the design model, because we have every information at one place with all the necessary cross-references, which could ease the understanding of the details.

Beside the above mentioned static class diagram the design model contains interaction, state, component and deployment diagrams to cover various aspects of the full system.

One example of these diagrams is Figure 3, which shows an interaction including the checkPermission() method of the example, FASBase interface.

Apart from the static model, we currently only use these diagrams for documentation purposes. We considered generating implementation code (state-machine) for some classes, with complex internal state, but their low number did not validate the effort of implementing a generator.

4.1. API DOCUMENTATION

The static model is stored as Java source files, so we use the Java documentation tool to generate various formats of the API documentation. Fortunately the Java documentation extensions allow rich formatting and extensive set of hyperlinking among the various parts of the documentation snippets (see the details on page 7), therefore the generated documentation is very useful for interactive purposes and also looks good on paper.

The Java documentation tool is using small programs, so called Doclets, to specify the content and format of the output of the Javadoc tool. The standard Doclet generates HTML format output, which we use for on-line documentations and also add it to our interface packages.

Another Doclet provides LATEX formatted output (see TexDoclet in Figure 1), which is then rendered into PostScript and Portable Document Format (PDF) formats for printing and on-line publishing.
One could also generate documentation from the language neutral web service interface descriptions (see Section 5.), but there are no standardized extensions for formatting and hyperlinking in those documentation snippets, therefore it would lack some qualities both for interactive usage and for printing.

4.2. Example

The following interface documentation of the FASBase interface is generated from the Java source by the TeXDoclet and included in this documentation as-is\textsuperscript{2}:

---

BEGINNING OF GENERATED API DOCUMENTATION

INTERFACES – ORG.GLITE.DATA.CATALOGSERVICE.FAS

Interface FASBase

---

The middleware level API for File Authorization Service.

Declaration

public interface FASBase extends org.glite.data.catalog.service.ServiceBase

All known subclasses

FAS

\textsuperscript{2}minor formatting modifications were taken to match the style of this document
**ALL KNOWN SUBINTERFACES**

**FAS**

**METHODS**

- void `checkPermission(String[] items, Perm perm)`
  throws InternalException, AuthorizationException

  **Description**
  Checks if the current user has the required permission bits on the specified items.

  **Parameters**
  * items – list of items we want to check permission of
  * perm – the permission bits to be checked against

  **Throws**
  * AuthorizationException – No access right to check permission information
  * InvalidArgumentException – Some part of the argument is invalid.

  **Semantic description of the method:**
  The following steps are performed for each item entry:
  * Check PERMISSION permission on the item;
    if permission is not granted, throws `AuthorizationException`.
  * Check required permission bits on the item;
    if any of the permission bits is not verified, throws `AuthorizationException`.

- PermissionEntry[] `getPermission(String[] items)`
  throws InternalException, AuthorizationException, NotExistsException, InvalidArgumentException

  **Description**
  Retrieves all set permissions for any kind of item (GUID, LFN or SchemaName).

  **Parameters**
  * items – list of items we want permission for

  **Returns**
  returns item with corresponding permission

  **Throws**
  * AuthorizationException – No access right to get the permission information.
  * NotExistsException – The item does not exist.
  * InvalidArgumentException – Some part of the argument is invalid.
  * InternalException –

  **Semantic description of the method:**
  The following steps are performed for each entry:
  * Check format of item (depends on the context);
    if it is invalid, throws `InvalidArgumentException`.
  * Check if item exists;
    if it does not exist, throws `NotExistsException`.
  * Check LIST permission on the item;
    if permission is not granted, throws `AuthorizationException`.
  * Query permission details from the catalog and add it to the list to be returned.

- void `setPermission(PermissionEntry[] permissions)`
  throws InternalException, AuthorizationException, NotExistsException, InvalidArgumentException
– **Description**
Sets full set of permissions \{BasicPermission,ACL\} for a given item (GUID, LFN or SchemaName).

– **Parameters**
* permissions -- list of \{item, permission\} pairs. The PermissionEntry PermissionEntry object takes the item and a list of ACLs. This method sets the full permission, replacing all existing ACLs.

– **Throws**
* AuthorizationException – No access right to update the permissions.
* NotExistsException – The item does not exist.
* InvalidArgument Exception – Some part of the argument is invalid.
* InternalException –

**Semantic description of the method:**
The following steps are performed for each entry:
* Check format of item (depends on the context);
  if it is invalid, throws InvalidArgument Exception.
* Check if the item exists;
  if it does not exist, throws NotExistsException.
* Check PERMISSION permission on the item;
  if permission is not granted, throws Authorization Exception.
* Updates permission details in the catalog.

[end of generated API documentation]

5. **INTERFACE**

The starting point of every client and server side web service implementations is the interface description in the XML based Web Services Description Language (WSDL) format.

Interface descriptions are generated from the static model, more precisely from the Java source files. We are using the Apache Axis java2wsdl converter, because it can handle interface inheritance and also generates complex type descriptions for every parameters.

Unfortunately it also has some shortcomings, because it uses the pre-compiled, Java byte-code format of the interfaces as the starting point. This format does not include the documentation, and does not have parameter names in the methods. We could not overcome the first problem, but we managed to add parameter names, by generating fake implementation classes for the interfaces and using them as for the java2wsdl conversion.

If one could generate WSDL interface descriptions directly from the Java source, then these shortcomings would disappear. Or not only disappear, but such a generator could add additional meaning to the WSDL documents from special tags in the documentation or from annotations in the source code. At the time of choosing our tools we could not find such a generator with a sufficient output.

One could also generate WSDL documents from the design model itself, exploiting the scripting or generator capabilities of the design tool. Unfortunately our chosen design tool was not able to generate WSDL, and we did not have resources to write such a plug-in ourselves.

---

3Java bytecode does not include parameter names for interfaces, but it does for classes
4Java 1.5 feature
5.1. WS-I COMPLIANCE

There are many web service language bindings defined and many more actual implementations of those language bindings. Because of the limitations of the various programming languages there is only a certain subset of the valid WSDL documents, which are valid in most of the implementations.

The Web Services-Interoperability Organization was formed by the users and implementors of the various bindings to determine the valid subset of the WSDL documents, which can be used interchangeably. Their first recommendation is called Basic Profile v1.0. Every implementation claiming WS-I compliance has to be able to accept WSDL documents conforming this profile.

The organization has also written a test tool to check if the WSDL documents conform this profile.

We integrated this test as a unit-test for the generating process, to see if our generated interfaces could be used by others.

Unfortunately Axis v1.1 implementation was not able to generate fully compliant WSDL interface descriptions, but Axis v1.2 is able to.

5.2. EXAMPLE

The following WSDL interface description is for the earlier FASBase interface. The complex type definitions and the inherited methods are omitted for brevity:

```xml
    <xs:element name="checkPermissionRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="items" type="impl:ArrayOfString"/>
                <xs:element name="perm" type="tns1:Perm"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="checkPermissionResponse">
    </xs:element>
    <xs:element name="getPermissionRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="items" type="impl:ArrayOfString"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="getPermissionResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="getPermissionReturn" type="impl:ArrayOf_tns1_PermissionEntry"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="setPermissionRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="permissions" type="impl:ArrayOf_tns1_PermissionEntry"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="setPermissionResponse">
    
</xs:schema>
```

6. **JAVA CLIENTS AND SERVICES**

Client and server side classes and helper classes are generated from the WSDL interface description. Before the details of these classes a language independent naming convention has to be introduced.
6.1. Classes of a Remote Method Call

To make a remote method call transparent from the client application to the server implementation, at least two intermediate classes are generated (see 4).

![Remote Method Call Classes diagram]

**Figure 4:** Remote Method Call Classes

On the client side it is called the *Stub* class. This class implements the service interface and connects to a remote server via the network. When a method is called on the interface from the client application, this class serializes the method parameters (even complex types), sends them to the server over the network, waits for the response, and de-serializes the reply and returns it.

Apart from the additional exceptions, this *Stub* class hides the details of the remote connection, so the client application can simply be coded against the service interface.

On the server side the main helper class is called the *Skeleton* class. This class does the handling of the calls, coming from the network: de-serializes the method parameters, calls the implementation, serializes the returned object(s) or exception and sends the result back to the client.

The actual body of the method implementations is separated into another class, which is called by the *Skeleton*. This *implementation* is provided by the developer, simply by implementing the service interface.

The service interface may contain non-primitive types and exceptions. These are part of the interface description, thus additional helper classes are generated both on client and server side for them.

6.2. Axis Clients and Services

We are using the *Axis wsdl2java* converter to generate the client and serverside code for Java.

Both client and server side generates the helper classes for the objects, passed as arguments in the interfaces. These classes define the fields, described by the WSDL, and additional metadata about the original WSDL type description. According to the Java Bean naming schemes additional access methods are also generated for the data fields. To be able to pass them over the network serializer and de-serializer methods are also part of the generated code.

Naturally both the client and the server side generates the interface description as a Java interface.

On the client side there are few additional classes generated:

- **stub**: an instance of this class communicates with an instance of the service
- **service factory**: a factory class (and its interface) to create *stub* instances from a URL

On the server side there are not only Java sources, but also configuration files are generated:
skeleton: to bridge the gap between the transport layer and the actual implementation for each service instance

example: an example implementation, which could be used as the starting point of the development (empty method bodies)

configuration: service (un)deployment configuration for the Axis runtime environment

The implementation of the service simply replaces the generated example implementation (i.e. using the same name).

6.3. Example

Using the above described FAS WSDL file the following files are generated by the \textit{wsdl2java} program:

6.3.1. Server Side

FAS.java – interface

\texttt{FASSoapBindingSkeleton.java} – the skeleton class

\texttt{FASSoapBindingImpl.java} – implementation example

deploy.wsdd – Axis, web service deployment descriptor

undeploy.wsdd – Axis, web service un-deployment descriptor

The \texttt{deploy.wsdd} file is used by Axis to bind the actual implementation class with the methods of a WSDL interface. It is added to the WAR file of the web service.

Server side implementation by a simple example code:

```java
package org.glite.data.catalog.service.fas;

import org.glite.data.catalog.service. *
;

public class FASSoapBindingImpl implements fas.FAS {
    public void addGUID(PermissionEntry[] permissions) throws RemoteException, InternalException, ExistsException, AuthorizationException {
        /* ... */
    }

    public void removeGUID(String[] guids) throws RemoteException, InternalException, NotExistsException, AuthorizationException {
        /* ... */
    }

    public void checkPermission( String[] items, Perm perm) throws RemoteException, InternalException, AuthorizationException {
        /* ... */
    }

    public PermissionEntry[] getPermission( String[] items)
        throws RemoteException, InvalidArgument exception, InternalException, NotExistsException, AuthorizationException {
        return /* ... */;
    }
```
public void setPermission (PermissionEntry[] permissions) 
throws RemoteException, IllegalArgumentException, InternalException, 
NotExistsException, AuthorizationException {
    /* ... */
}

public String getVersion() throws RemoteException {
    return /* ... */;
}

public String getSchemaVersion() throws RemoteException {
    return /* ... */;
}

public String getInterfaceVersion() throws RemoteException {
    return /* ... */;
}

public String getServiceMetadata(String key) throws RemoteException {
    return /* ... */;
}

6.3.2. CLIENT SIDE

FAS.java – interface

FASService.java – service factory interface

FASServiceLocator.java – service factory implementation

FASSoapBindingStub.java – client side stub

Usage by a simple example code:

// [...] package imports

public class BasicFASTestFunc extends TestCase {
    private FAS m_stub = null;

    public void getSimplePermission(String user, String group) {
        /* ... */
    }

    public void checkSimplePermission(Permission perm) {
        /* ... */
    }

    public void test() throws Exception {
        String endpoint = // [...] getting the endpoint

        FASServiceLocator locator = new FASServiceLocator();
        m_stub = locator.getFAS(new URL(endpoint));

        String version = m_stub.getVersion();
        System.out.println("Server_version_is_” + version);

        // adding a new file
        String guid = // [...] getting the guid
PermissionEntry p1 = new PermissionEntry();
p1.setItem (guid);

Permission perm = getSimplePermission("egee", "egee−group");
p1.setPermission (perm);

m_stub.addGUID(new PermissionEntry[]{ p1 });

// checking the added file
PermissionEntry[] p2array = m_stub.getPermission(new String[] { guid });
PermissionEntry entry_back = p2array[0];

// check item is the same guid we asked for
assertEquals (guid, entry_back.getItem ());

// check the permissions are the same
Permission perm_back = entry_back.getPermission ();
assertNotNull (perm_back);
checkSimplePermission(perm_back);
}

7. C, C++ CLIENTS AND SERVICES

For C and C++ language bindings we use the gSOAP framework.

TODO: add details

8. PERL

We only have client side solution for Perl. This is based on the SOAP::Lite module. By default this module assigns simple SOAP types to complex Perl language structures, which are passed as arguments. This usually does not match the name and structure of the complex types described in the WSDL, therefore one has to tell these mappings explicitly.

The WSDL contains all the static type information, so we wrote an XSLT filter, which directly generates a glue Perl code. This glue code makes the coding of the client side convenient in Perl, but enforces the proper type names and structures in the method calls.

We also use a simple base library (GLite::HTTPS), which configures the transport layer with the user’s proxy certificate, if the service endpoint is a secure one.

8.1. EXAMPLE

The following generated glue code shows the encoding of the Permission class: it associates the proper WSDL type name with all elements of the structure.

package GLite::Data::FAS;
use strict ;
no strict 'refs';
use SOAP::Lite;
use GLite::HTTPS;
use Data::Dumper;

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};
    $self->{SOAP} = undef;
    bless($self, $class);

    my $url = shift;
    defined $url or print STDERR "FAS endpoint was not defined!\n" and return;

    GLite::HTTPS::setEnv() if $url =~ m/https:/i;

    $self->{SOAP} = new SOAP::Lite(
        uri => 'org.glite.data.catalog',
        proxy => $url,
        on_fault => sub {
            my($soap, $res) = @_;
            die $soap->transport->status, "\n" unless ref $res);
            local $Data::Dumper::Pad = "\nDetails: \n";
            die $res->faultcode(), "", $res->faultstring(), "\n",
            Dumper($res->faultdetail()), "\n";
        },
    );

    # namespace definitions for
    # 'http://glite.org/wsd/ services/org.glite.data.catalog'
    $self->{SOAP} = new SOAP::Serializer(->
        serializer() =
        maptype => {'Perm'} = "http://glite.org/wsd/ services/org.glite.data.catalog";
    # [... some more definitions

    return $self;
}

# [...] some other methods

sub checkPermission {
    my ($self) = shift;
    my $result = $self->{SOAP} = new SOAP::Serializer(->
        serializeArrayOfSoapenc_string ('items', shift),
        serialize_Per('perm', shift)
    ) =
    result();
    return $result;
}

sub getPermission {
    my ($self) = shift;
    my $result = $self->{SOAP} = new SOAP::Serializer(->
        serializeArrayOfSoapenc_string ('items', shift)
    ) =
    result();
    return $result;


```perl
sub setPermission {
    my ($self) = @_;  # the name of the function
    my $result = $self{SOAP}->setPermission(  # the Perl structure w/o type info
        'permissions', $self{ArrayOf_tns1_PermissionEntry} );
    return $result;
}

sub _serialize_Perms {
    my $name = @_;  # the name of the element
    my $rawValue = @_;  # the Perl structure w/o type info

    # dealing with inheritance: optional parameters from the subclass
    # calling the superclass’s serializer routine with its type and fields
    my $type = @_;  # the name of the element
    my $value = @_;  # the Perl structure w/o type info

    $value{permission} = SOAP::Data->new (
        name => 'permission',
        type => 'xsd:boolean',
        value => $rawValue{permission}
    );
    # [...] some other fields
    return SOAP::Data->new (name => $name, type => $type, value => $value);
}

# [...] other type serializers
```

9. **JavaScript**

For platform independent GUI we decided to use the web interface. The logic of rendering the interface can be placed on the server side, returning full HTML pages to the browser. However this technique might look slow, if only a small part of the interface has to be updated.

For the best user experience we decided to put the rendering logic into the browser and use asynchronous calls to the already existing web service interface. The basis of the interface is HTML, which is updated by a logic, written in JavaScript.

The JavaScript implementation of browsers already provide the basic functionality to implement a SOAP call, but these implementations cannot deal with complex types, only with strings. Therefore we wrote our own extension library, which is similar to the Perl solution.

An XSLT filter directly generates the glue JavaScript library from the WSDL, to make the development of the user interface simple. The generated library exports the web service interface and all of its argument types as JavaScript objects.
9.1. Example

The following example is a short part of the generated JavaScript library, which wraps the endpoint and the same Permission class, as the Perl example.

The glue code relies on our basic SOAP routines to do the actual SOAP call.

The library calls are asynchronous, each of them happen in a separate and new thread. The return value (or thrown exception) of the call has to be handled by a callback routine, specified by the caller.

```javascript
/**
 * Stub definition to access a FAS.
 */
function FAS(endpoint) {

    // The endpoint of the service
    this.endpoint = endpoint;

    // Get the SOAP engine object
    var soap = new SOAP();

    // Set the endpoint of the service
    soap.endpoint = "services/FAS";

    this.addGUID = function(permissions, callback) {
        // Prepare the input parameters
        var inputParams = new Array();
        inputParams.push(
            new SOAPParam("permissions",
                "impl: ArrayOf_tns1_PermissionEntry",
                permissions));

        // Prepare the output parameter
        var outputParam = undefined;

        // Make the call
        return soap.call("addGUID",
            "http://glite.org/wsdl/services/org.glite.data.catalog.service.composite",
            inputParams,
            outputParam,
            callback);
    };

    // further method stubs are removed for brevity
}

/******* Complex Types **********/
/**
 * Perm object.
 */
function Perm() {
```
this.permission = new SOAPParam("permission", "xsd:boolean", "init -permission");
this.remove = new SOAPParam("remove", "xsd:boolean", "init -remove");
this.read = new SOAPParam("read", "xsd:boolean", "init -read");
this.write = new SOAPParam("write", "xsd:boolean", "init -write");
this.list = new SOAPParam("list", "xsd:boolean", "init -list");
this.execute = new SOAPParam("execute", "xsd:boolean", "init -execute");
this.getMetadata = new SOAPParam("getMetadata", "xsd:boolean", "init -getMetadata");
this.setMetadata = new SOAPParam("setMetadata", "xsd:boolean", "init -setMetadata");
this.setPermission = function (permission) {
  this.permission = new SOAPParam("permission", "xsd:boolean", permission);
}
this.getPermission = function () {
  return this.permission.value;
}

// further access methods are removed for brevity

Perm.prototype = new SOAPObject();
// further complex type handlers are removed for brevity

10. CUSTOMIZATION TECHNIQUES

There are several techniques to make maximize use of the generated code and minimize the developers work. These solutions have to be synchronized or woven into the generation process.

- using inheritance to implement services
- using code diffs/patching to augment the generated classes (not mentioned in the article)
- using Axis handlers for runtime aspect weaving (doing things before or after a method is called). For example authentication and coarse grained authorization.
- complex customization: fine grained authorization has to be merged into the functionality, the implementing code, because decisions are also based on the arguments passed to the operation

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


INFSO-RI-508833 PUBLIC 22/23