Jsr181ImplArchitecture

Architectural Overview

The processor represents the core of the JSR-181 implementation. It reads various input files and processes their contents into the configured output format. The processor supports both modes Start with Java and Start with WSDL through the API as well as through the command line interface. Figure 2 depicts a rough breakdown of the processor into its major functional components in a reference block diagram (solid lines are used for data flows, dashed lines for control flows).

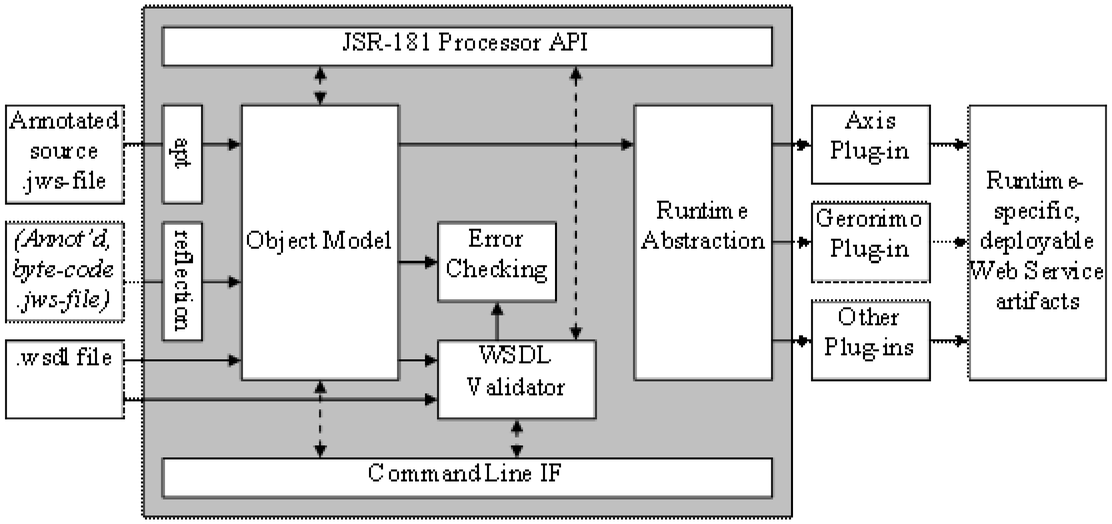

Figure 2: Reference block diagram of JSR-181 processor.

Processing Model

Invocation / Control Flow

Generally, the processor can be invoked either from the command line or through the API. Depending on the runtime's capabilities and configuration, there are several ways to invoke the processor:

1. As integral part of the compilation of an annotated Java Web Service source file to generate deployable Java Web Service (e.g. .war file) as output of the compilation. 2. As an additional step after the compilation to convert the compiled annotated Java Web Service file into a set of artifacts that can be deployed into a container (e.g. bundled as a .war file). 3. Automatically in the container, with WSDL generation and runtime configuration directly from the annotated Java Web Service file.

"Generate Web Service" (~ Start with Java)

While the processor expects the Web Service implementation (input) file to be an annotated Java Web Service file, required message and data type definitions may be provided in either Java or XML Schema. Using the Axis tool Java2WSDL, the annotation processor reads class and annotation information (Web Service meta-data) from various input-files and creates an internal representation of the Web Service based on that meta-data. After checking for semantic errors, the core interprets the meta-data and hands the results over to the runtime abstraction and the processor plug-ins, which in turn generate the Web Service artifacts. If the type definitions are provided in XML Schema, plug-ins will automatically generate Java classes for those definitions, as part of the set of generated Web Service artifacts. The plug-ins may include a staging facility, which may repackage or reformat the generated artifacts, for instance.

"Generate Service Description"

As part of generating a Web Service from annotated Java Web Service source or byte code files, the processor generates a description of the Web Service in form of a WSDL file. TBD.

"Generate Code" (~ Start with WSDL)

The processor reads the description of the Web Service from a single WSDL file and optionally one or more XML schema message or data type description. It then generates an annotated Java Web Service file skeleton, Java message and data type classes and other artifacts. TBD.

"Validate"

The processor reads an annotated Java Web Service file and compares its contents (including annotations) against a service description in a WSDL file. TBD.

"Check Semantic Errors"

The processor performs additional error checking using the Web Service meta-data (see below). TBD.

Deployment Options

The deployment process for generated Web Services depends on the target runtime and is implemented in the respective runtime plug-ins. Here are some sample approaches to the deployment process:

1. Deployment of the annotated Java Web Service source file to a server directory (monitored directory; e.g. Axis): the runtime automatically (e.g. upon the first request sent to the Web Service) invokes the processor to generate the Web Service from the source. 2. Automatic deployment with external overrides: same as 1., but an external configuration file can override the configuration settings in the source file. 3. J2EE Web Services: EJB implementation, see J2EE 1.4, JSR-109.

TBD.

Functional Components

Interfaces

As shown above, the processor provides two interfaces, an API and a command line interface. Each user interface supports both modes, Start with Java and Start with WSDL.

Processor API

The API is intended for applications such as IDEs (e.g. Pollinate, WebLogic Workshop, etc.), or Web Service containers (e.g. Axis) that require a direct, programmatic interface. The API consists mainly of two interfaces:

- The WsmAnnotationProcessor, which subclasses the TwoPhaseAnnotationProcessor. The TwoPhaseAnnotationProcessor subclasses Sun's AnnotationProcessor and splits the process() method into two check() and generate() (see control subproject).
- The AnnotationModel, which provides access to the metadata for a given Web service through the method "getModel(<name>)".

Command Line Interface

When invoked through the command line interface, the processor can be run as a stand-alone tool to manually generate artifacts for deployable Web services.

Annotation Processor

The annotation processor reads annotated Java Web Service files, parses it for annotations and creates a model of the Web Service based on the annotations found in the input files. It can work with byte-code and source of annotated Java Web Service files; adequate abstractions within the annotation processor encapsulate input file properties (e.g. source vs. byte-code). After reading the input files, the same code is used for all input formats. The annotation processor is based on Sun’s annotation processing tool (apt).

Core and Annotation Handler

The core together with the annotation handler uses the annotation processor’s output to generate the Web Service artifacts.

Error Checking

JSR-181 suggests that while generating artifacts for Web Services from service description or other input files the processor check for (semantic) errors that cannot be detected by the Java compiler:

- Type checks (e.g. check if a String parameter represents a valid URL when required)
- Check annotation vs. code (e.g. check input/output parameters for @Oneway)
- Check annotation compatibility and consistency (e.g. @Oneway requires @WebMethod)
- TBD.

WSDL Validator

The validator compares a WSDL file against a Web Service implementation file (in source or byte-code format); it is based on the Axis tools WSDL2Java and Java2WSDL.

Runtime Abstraction

The runtime abstraction provides a unified interface to all runtime plug-ins for the processor. Thus, the JSR-181 processor internals are kept entirely independent from runtime specific Web Service implementation details.

Runtime Plug-in

Each runtime plug-in produces the (runtime-specific) physical artifacts for the Web Service for a specific target runtime. Each runtime plug-in encapsulates the implementation details for one specific runtime. The runtime plug-in may include a packaging tool if required, which packages the generated artifacts in the format expected by the runtime. Typically, there is one runtime plug-in per supported runtime. Note that the physical image (number, format and size of artifacts) of one and the same Web Service looks typically different for each runtime. Finally, runtime plug-ins may extend existing tools, such as Java2WSDL.

Input Files

Annotated Java Web Services File

When run in Start with Java mode, the processor consumes one JSR-181-compliant, annotated Java Web Service (source or byte-code) file as input and optionally one or more .xsd files that define message and/or types in XML Schema. For most annotation parameters, JSR-181 defines default values or how such values can be derived. If the processor starts with a byte-code file instead of a source file, certain information (such as a method's parameters name in the source file) is not available anymore; such information is lost during the compilation process. The processor then falls back to using implementation specific defaults, which simplifies the development of the Web Service, but is not always compliant with JSR-181.

Service Description File

When run in Start with WSDL mode, the processor consumes one service description (.wsdl) file.

Output Files

Depending on its operational mode, the processor generates different file sets:

Web Service Artifacts

When run in Start with Java mode, the processor produces a Java Web Service in form of a set of deployable artifacts. These artifacts vary depending on the target runtime and typically include at least a service description (.wsdl) file and one or more Java class files or archive files that contain such Java class files.

Java Skeletons

When run in Start with WSDL mode, the processor produces Java code skeletons that can be used to implement the Web Service described in the original WSDL input file.

External Dependencies

The suggested design depends on the following external tools and projects:

- JDK 1.5 (with support for annotations)
- apt (annotation processing tool)
- Apache Axis tools: SOAP stack, Java2WSDL, WSDL2Java

Security

WsmSecurityModel