Woden User Guide

Table of contents

1. Introduction

The purpose of the Woden User Guide is to explain how to use the external interfaces of Woden. These external interfaces currently consist of the Woden API, but as development progresses they may also include other configuration techniques, command line tools and script-based utilities such as ANT tasks. Woden users are typically developers of other tools and technologies that use Woden for parsing or manipulating WSDL documents.

The Woden User Guide will not discuss the 'internals' of the Woden implementation and it will only touch on the Woden design where it is relevant to this discussion of how to use Woden. A Woden Developer Guide (not yet written) will discuss the design and implementation.

The User Guide reflects the current state of Woden's development and will be updated as new function is added to Woden. See the "Woden Overview" section below for up-to-date details of what function currently exists and what doesn't.

This Guide should be read in conjunction with the Woden API Javadocs included with the milestone distribution (see the "Download and Setup" section below). Please post any questions or comments to the Woden development mailing list, woden-dev@ws.apache.org.

2. Woden Overview

The initial goal of the Woden project is to develop a WSDL 2.0 processor that implements the W3C WSDL 2.0 specification in response to the Working Group's call for implementations. This includes defining an API for Woden separate to its implementation, so that other projects can modify or replace the Woden implementation while maintaining consistent external interfaces. Further goals include support for high performance XML parsing and support for WSDL 1.1.

The objectives to achieve these goals are:

- Develop a WSDL 2.0 processor (DOM-based initially) for serializing and deserializing WSDL 2.0 documents.
- Define a WSDL object model that conforms to the W3C WSDL 2.0 spec, including support for WSDL and type system extensibility.
- Enable WSDL to be created or modified programmatically via this object model.
- Develop Woden as a configurable and extendable framework to support implementation-specific customization, such as customization of object factories, validation and error handling.
- Develop a framework extension mechanism that allows alternative XML parsers to be used to support different usage scenarios and performance requirements, then develop a

WSDL parser implementation based on an XML streaming 'pull' parser such as StAX (JSR173).
- Support deserializing WSDL 1.1 documents and optionally converting them into the WSDL 2.0 object model.
- Support deserializing WSDL 1.1 documents and optionally representing them with the JWSDL object model (i.e. the WSDL 1.1 object model defined by JSR110 "Java APIs for WSDL" and currently implemented by WSDL4J).
- Develop a comprehensive Junit-based test suite that integrates the WSDL test cases from the W3C WSDL 2.0 test suite and includes Woden-specific tests that cover its API and the features of the Woden framework.

The functionality that currently exists in Woden and is described in this User Guide includes:

- A factory mechanism used to obtain a WSDL reader (or parser) object.
- A DOM implementation of the reader, based on Apache Xerces.
- Configuring WSDL reader features and properties - for example, switching the validation feature on or off.
- Using the reader to parse (deserialize) a WSDL document at a specified URL into the Woden WSDL 2.0 object model.
- Two forms of the WSDL 2.0 object model; one representing the WSDL 2.0 abstract Component model and one that maps to the XML elements and attributes in the WSDL namespace.
- Parsing of most of the elements and attributes in the WSDL 2.0 namespace (the 'extends' attribute of Interface is the only outstanding item).
- Partial WSDL validation using the assertions defined in the WSDL 2.0 spec. Validation currently exists for Types, Interface and Binding.
- Manipulating the WSDL via methods of the WSDL 2.0 object model.
- Support for extensibility elements and attributes (i.e. for XML elements and attributes that extend elements in the WSDL 2.0 namespace).
- Use of this extensibility mechanism for the SOAP binding extensions defined in the WSDL 2.0 spec.
- A customizable error handling mechanism for reporting warnings, errors or fatal errors that result from WSDL validation or Woden configuration problems.

Planned functionality that does not yet exist in Woden and is NOT described in this User Guide includes:

- Parsing of HTTP binding extensions, as defined in the WSDL 2.0 spec.
- WSDL validation of import and include, SOAP and HTTP binding extensions, and Service.
- Resolving URLs with an entity or catalog resolver
- A mechanism for extending Woden to support other XML parsers.

- A StAX implementation of this parser extension mechanism.
- A mechanism for extending Woden to support types systems other than W3C XML Schema (e.g. RelaxNG, DTD).
- Writing (serializing) the WSDL object model out to a WSDL document.
- Parsing WSDL 1.1 documents and converting them into the WSDL 2.0 object model.
- Parsing WSDL 1.1 documents and representing them using the JWSDL object model (i.e. a WSDL 1.1 object model)

3. Download and Setup

Obtain the Apache Woden WSDL processor in one of 2 ways:

- extract the source code from the Woden Subversion (SVN) repository and compile it, or
- obtain the Woden binary distribution from the latest milestone build.

Woden's DOM-based XML parsing depends on Apache Xerces 2.7.1. Its XML Schema support it depends on the schema parser and object model implemented by the Apache Web Services Commons (ws-commons) XmlSchema project.

The milestone build includes all of the required libraries and these must be on the classpath:

- woden.jar contains the Woden binary code
- xercesImpl.jar and xml-apis.jar contain Apache Xerces 2.7.1
- XmlSchema-SNAPSHOT.jar contains the Apache ws-commons XmlSchema

If using the Woden source code, rather than the milestone distribution, then the Apache Xerces 2.7.1 distribution can be downloaded from the Apache Xerces project here. The source code for Apache Web Services ws-commons XmlSchema can be extracted from its Subversion (SVN) repository using the logon id "anoncvs".

Woden requires Java 1.4 or higher.

4. Getting Started

This section contains a few code examples to demonstrate the Woden programming model. See "The Woden API" section below and the Javadocs included with the milestone distribution for more details.

The following code example shows how to obtain a `WSDLFactory` object which is then used to obtain a `WSDLReader` object (the WSDL parser). WSDL validation is then enabled on the reader, before the *readWSDL* method reads a WSDL document from the specified URL and returns the WSDL as a `DescriptionElement` object. The `DescriptionElement` represents the WSDL <description> element and along with its

contained objects, it declares an API that maps to the XML elements and attributes in the WSDL 2.0 namespace. This will be referred to as the **Element** API. The *toComponent* method on `DescriptionElement` returns the WSDL as a `Description` object, which represents the Description component from the WSDL 2.0 Component model. The API declared by `Description` and its contained objects will be referred to as the **Component** API.

```
      WSDLFactory factory = WSDLFactory.newInstance();
      WSDLReader reader = factory.newWSDLReader();
      reader.setFeature(WSDLReader.FEATURE_VALIDATION, true);
      DescriptionElement descElem = reader.readWSDL(wsdlurl);        <--
the <description> element
      Description descComp = descElem.toComponent();                 <--
the Description component
```

The parameter `wsdlurl` is the String representation of a URL, e.g.:

```
      wsdlurl="http://ws.org.apache/woden/services/Booking.wsdl"
      wsdlurl="C:/woden/services/Booking.wsdl"
```

To obtain the top-level WSDL elements from the `DescriptionElement`:

```
      InterfaceElement[] interfaces = descElem.getInterfaceElements();
      BindingElement[] bindings = descElem.getBindingElements();
      ServiceElement[] services = descElem.getServiceElements();
```

This example shows how to get the global schema element declaration (represented by the `XmlSchemaElement` class from Apache ws-commons XmlSchema) which is referred to by QName in the 'element' attribute of the interface <fault> element:

```
      InterfaceElement interfaceElem = interfaces[0];
      InterfaceFaultElement[] faults =
interfaceElem.getInterfaceFaultElements();
      XmlSchemaElement xsElem = faults[0].getElement();
```

Where the WSDL is composed of multiple WSDL documents via WSDL <import> and <include>, you can navigate the WSDL modules using the methods *getImportElements* and *getIncludeElements* of `DescriptionElement`:

```
      ImportElement[] imports = descElem.getImportElements();
      DescriptionElement importedDescElem =
imports[0].getDescriptionElement();
```

The `Description` component also has methods to retrieve the top-level WSDL components, but unlike those in `DescriptionElement`, the behaviour here is to 'flatten' the WSDL. That is, to return the top-level components of the initial description and of all imported or included descriptions as well:

```
```

```
        Interface[] allInterfaces = descComp.getInterfaces();
        Binding[] allBindings = descComp.getBindings();
        Service[] allServices = descComp.getServices();
```

The next example shows how to get all of the `ElementDeclaration` and `TypeDefinition` components from the `Description` component. These represent the global schema element declarations and type definitions from the XML Schemas defined in-line or imported within the WSDL <types> element. Once again, this is a 'flattened' view that includes schema components from imported or included WSDL documents (assuming the WSDL 2.0 rules about schema visibility have been followed):

```
        ElementDeclaration[] elemDecls = descComp.getElementDeclarations();
        TypeDefinition[] typeDefs = descComp.getTypeDefinitions();
```

5. The Woden API

This section provides an overview of the Woden API.

The Woden WSDL processor is implemented as a framework with extension points for adding user-defined behaviour. The details of this implementation are 'hidden' by the Woden API. Even the extension points are exposed on the Woden API, either as Java interfaces that can be re-implemented or as Java classes that can be extended. With the Woden extension and programming model based on the API, there should be no need to refer to Woden implementation classes in user code. If you think you have such a need, please post your requirements to the Woden development mailing list.

The Woden API contains two 'sub-APIs', introduced previously in the "Getting Started" section, which represent alternative WSDL 2.0 object models:

- The **Element** API which represents a model of the XML elements and attributes in the WSDL 2.0 namespace, as described by the XML mappings in the WSDL 2.0 specification.
- The **Component** API which represents the abstract WSDL Component model described by the WSDL 2.0 specification.

Whereas the Element and Component APIs are concerned solely with WSDL representation and manipulation, the remainder of the Woden API is concerned with how to use, configure and extend the Woden WSDL processor. The term **Woden** API encompasses these more general features of the Woden processor and the WSDL-specific features. However if we need to discuss these WSDL-specific features of the API, we may use the terms Element or Component API to be more specific.

API Packages

The Woden API is declared by Java interfaces and a small number of Java classes within

package names beginning with `org.apache.woden`. Woden implementation package names begin with `org.apache.woden.internal` to distinguish them from the API packages. All other `org.apache.woden` packages are part of the Woden API.

The most important API packages are:

`org.apache.woden`
This contains the core components of the Woden WSDL processor - WSDLFactory, WSDLReader, WSDLException, ErrorReporter, ErrorHandler to name a few.

`org.apache.woden.schema`
This contains interfaces representing both in-lined and imported XML schemas. These represent schemas in terms of the <xs:schema> and <xs:import> elements that can appear directly under the WSDL <types> element.

`org.apache.woden.wsdl20.extensions`
This represents the extension architecture to support extension elements and attributes (i.e. those that are not in the WSDL 2.0 namespace). This includes a mechanism for registering user-defined serializers, deserializers and Java mappings for these extensions.

`org.apache.woden.wsdl20.extensions.soap`
This contains Java classes that map to the SOAP binding extensions defined in the WSDL 2.0 spec.

`org.apache.woden.wsdl20`
Contains the Java interfaces that make up the **Component** API (i.e. the abstract WSDL Component model).

`org.apache.woden.wsdl20.xml`
Contains the Java interfaces that make up the **Element** API (i.e. the XML mappings for WSDL elements and attributes).

Core API Features

The core features of the Woden API include:

- The factory mechanism for creating Woden objects such as
- Configuring Woden behaviour by setting features or properties of the `WSDLReader`.
- Customizing the error handling behaviour.
- Registering user-defined extensions to support elements and attributes outside of the WSDL 2.0 namespace.
- Manipulating the XML-based model of WSDL elements and attributes (i.e. via the Element API).
- Manipulating the abstract model of WSDL components (i.e. via the Component API).

The `WSDLFactory` class has static methods *newInstance()* and *newInstance(String className)* that return a factory object. The noarg version adopts a strategy to search for a user-configured factory classname, defaulting to a Woden-provided factory class if none is found. The factory class name search strategy is to check first for a Java system property, then check for a property file in the JAVA-HOME/lib directory (we intend also to search for a property in META-INF/services but this is not implemented yet). The Javadoc for this class provides details of the system property and property file names. The *newInstance(String className)* version allows you to specify the factory class to be instantiated. This factory object is used to create some of the key objects of the Woden programming model such as `WSDLReader`, `DescriptionElement` and `ExtensionRegistry`.

The Woden parsing behaviour can be configured by setting features or properties of the `WSDLReader` object. Note, these are Woden-specific configuration details, not to be confused with the WSDL Feature and Property components. Reader features are configured via the *setFeature* method with a feature name and a boolean value, indicating whether the feature is enabled. The *getFeature* method is used to query whether a specified feature is enabled. Reader properties are configured via the *setProperty* method with a property name and an object representing the property. Likewise, a *getProperty* method returns the property object for a specified property name. The names of the Woden-defined features and properties are specified on the API as `public static final` constants on the `WSDLReader` interface. See the API Javadoc for details. These methods may also be used to configure user-defined, implementation-specific features and properties. The "Getting Started" section above showed an example of feature configuration - the Woden validation feature was enabled on the reader object by the code:

```
reader.setFeature(WSDLReader.FEATURE_VALIDATION, true);
```

The API provides error handling through four interfaces and the `WSDLException` class. System configuration errors are typically handled by throwing a `WSDLException` containing appropriate error information. WSDL parsing errors are reported by the `ErrorReporter` which delegates the reporting style to the `ErrorHandler`. `ErrorHandler` recognizes 3 types of error; warnings, errors and fatal errors. A default error handler implementation is provided with Woden which prints all 3 types of message to `System.out` and then for fatal errors only, terminates processing with a `WSDLException`. Users may provide their own implementation of `ErrorHandler` to change this behaviour. The *setErrorHandler* method on `ErrorReporter` is used to set a user-defined custom error handler. User-defined extensions to Woden may use `ErrorReporter` to report their errors via the `ErrorHandler` or to obtain a formatted error message, for example to place inside an exception object. Messages are expected to have an error id and some message text, but users have the option of defining fully formatted messages or using parameterized strings in a Java ResourceBundle. `ErrorInfo` declares a data object containing the error information passed to the `ErrorHandler`. This includes

the `ErrorLocator` which specifies the URI of the WSDL source document and the line and column number where the error occurred (although this feature is not yet implemented).

Extension elements and attributes (those outside of the WSDL 2.0 namespace) are handled by the Woden extension architecture. For each extension element, a user-defined implementation of the `ExtensionDeserializer` and `ExtensionSerializer` interfaces will map the element to/from some user-defined implementation of `ExtensionElement` which represents the element. The deserializer, serializer and Java mapping classes are registered in the `ExtensionRegistry` so that the `WSDLReader` (or WSDLWriter when it gets implemented) will know what to do when it encounters this element. The Woden API includes `ExtensionElement` implementations to represent the SOAP binding extensions defined in the WSDL 2.0 spec (and HTTP extensions will follow soon). To handle extension elements that have not been registered, default behaviour is provided by the `UnknownDeserializer`, `UnknownSerializer` and `UnknownExtensionElement` classes. These Woden-defined extensions (SOAP and Unknown) are pre-registered in the `ExtensionRegistry` by the Woden implementation.

The package `org.apache.woden.xml` contains classes that represent the more common types of extension attribute values (e.g. string, QName, boolean, etc). These are all subclasses of `XMLAttr` which defines the *init* method for parsing an extension attribute value and the *toExternalForm* method for representing the value as a string. Users may extend `XMLAttr` to support other types of values. The `XMLAttr` subclass must be registered with its parent class name (i.e. its containing element) and the QName of the extension attribute in the `ExtensionRegistry`, so that the `WSDLReader` will have the information necessary to parse it correctly. The extension attributes defined in the WSDL 2.0 spec (i.e. those for the SOAP and HTTP binding extensions) will be pre-registered in the `ExtensionRegistry` using the `XMLAttr` subclasses defined in package `org.apache.woden.xml`.

The Element and Component APIs are discussed below.

Element API

The Element API allows you to navigate the nested hierarchy of WSDL elements that would appear in a WSDL document (as defined by the WSDL 2.0 Schema). For example, `DescriptionElement` declares methods *getInterfaceElements*, *getBindingElements* and *getServiceElements* which provide access to the top-level WSDL elements. `InterfaceElement` declares the methods *getInterfaceFaultElements* and *getInterfaceOperationElements* and so on. The Element API is described in detail in the Javadocs included in the milestone distribution.

Within the `org.apache.woden.wsdl20.xml` package, each WSDL element is represented by a Java interface. The WSDL attributes present in each WSDL element are

represented by appropriate methods on those interfaces. So for example, `DescriptionElement` has the method *getTargetNamespace*.

Note that the methods of the Element API do not 'flatten' composite WSDL structures. For example, the *getServiceElements* method returns the <service> elements defined directly within the containing <description> element, but not those defined within any imported or included descriptions. To retrieve all of the `ServiceElement`s from a composite WSDL, you need to navigate the WSDL structure using the *getImportElements* or *getIncludeElements* methods on `DescriptionElement`.

Component API

The Component API represents the abstract WSDL Component model described in the WSDL 2.0 spec. This differs from the Element API in that certain aspects of WSDL XML are not represented in the Component model. The <documentation> element is not captured in the Component model. The <types> element and particular type systems like XML Schema are not represented, however the Component API does contain `ElementDeclaration` and `TypeDefinition` which provide a general representation for global element declarations and type definitions, such as those used in XML Schema.

The composition of WSDL documents via the <import> and <include> elements is not represented in the Component model. Instead, the Description component represents the entire, composite WSDL structure and its properties which represent top-level WSDL components, like Interface, Binding and Service, contain a 'flattened' representation of the WSDL. For example, the *getInterfaces* method of `Description` will return not just the interfaces defined within the initial description, but those defined within any imported or included descriptions as well.

The Component API provides a read-only view of the WSDL Component model (i.e. it defines accessors but no mutators). The only way to create a `Description` object is by calling the *toComponent* method on a `DescriptionElement` object. Once you have a `Description` object you can access the rest of the WSDL component model, but you cannot modify it. WSDL can only be created or modified programmatically via the Element API.

Mapping of WSDL elements to the API

```
          WSDL element              Element API
Component API

       <description>           DescriptionElement
Description
         <documentation>       DocumentationElement
         <import>              ImportElement
         <include>             IncludeElement
```

```
         <types>                TypesElement
         <interface>            InterfaceElement
Interface
           <fault>              InterfaceFaultElement
InterfaceFault
           <operation>          InterfaceOperationElement
InterfaceOperation
             <input>            InterfaceMessageReferenceElement
InterfaceMessageReference
             <output>           InterfaceMessageReferenceElement
InterfaceMessageReference
             <infault>          FaultReferenceElement
InterfaceFaultReference
             <outfault>         FaultReferenceElement
InterfaceFaultReference
         <binding>              BindingElement
Binding
           <fault>              BindingFaultElement
BindingFault
           <operation>          BindingOperationElement
BindingOperation
             <input>            BindingMessageReferenceElement
BindingMessageReference
             <output>           BindingMessageReferenceElement
BindingMessageReference
             <infault>          FaultReferenceElement
BindingFaultReference
             <outfault>         FaultReferenceElement
BindingFaultReference
         <service>              ServiceElement
Service
           <endpoint>           EndpointElement
Endpoint
         <feature>              FeatureElement
Feature
         <property>             PropertyElement
Property

          XML Schema element

       <xs:import>              ImportedSchema
       <xs:schema>              InlinedSchema
         <xs:element name="..">
ElementDeclaration
         <xs:complexType name="..">
TypeDefinition
```

6. Woden URI Resolver

This allows URIs referred to in WSDL 2.0 and XML Schema documents to be redirected to alternative URIs. Woden is equipped with such a resolver as default, and an API to define alternative implementations.

The Resolver API

Users are free to create their own custom URI Resolvers, by implementing the interface `org.apache.woden.resolver.URIResolver`.

The resolver should be registered with the WSDLReader object before invoking *readWSDL()* methods.

Example:

```
URIResolver myResolver = new CustomURIResolver();
WSDLFactory factory = WSDLFactory.newInstance();
WSDLReader reader = factory.newWSDLReader();
reader.setURIResolver(myResolver);
...
reader.readWSDL(…..);
```

SimpleURIResolver

This is the URI resolver implementation provided with the Woden distribution, and it is also the default. When a WSDLReader object is requested, a `SimpleURIResolver` is automatically instantiated and registered with it. In other words the following happens implicitly:

```
reader.setURIResolver(new SimpleURIResolver());
```

If required, a custom resolver can be registered programmatically in place of the default, as shown above.

1 - Catalog file format

The catalog file follows the Java Properties file syntax: rows of entries of the form `<property name>=<property value>`, interspersed with comment lines starting with the "#" character. However, with catalog notation the meaning of the left and right hand expressions is slightly different:

```
<resolve-from URI>=<resolve-to URI>
```

where *resolve-from URI* is the subject of the resolution, and *resolve-to URI* is the place where the resolver looks for the resource. To be meaningful, the *resolve-to URI* should be a valid URL (that is, a reference a physical document).

By convention, URI catalog file names have the suffix `.catalog`, though this is not mandatory.

Note that the first ":" in the line of each entry must be escaped. See examples below.

The schema catalog is read sequentially when a `SimpleURIResolver` is instantiated. Where multiple entries exist in the catalog for a given resolve-from URI, the last such entry is used.

Absolute URIs

Examples:

```
       Resource held locally on an NTFS file system:
       http\://test.com/interface.wsdl=file:///c:/resources/interface.wsdl

       Similarly on a Un*x-based file system:
       http\://test.com/interface.wsdl=file:///resources/interface.wsdl

       Resource held remotely and accessed over http:
http\://test.com/interface.wsdl=http://aplace.org/resources/interface.wsdl
```

Relative URIs

If relative URIs appear in any resolve-to entries in the catalog, then a search path is used (on initialisation of the resolver) to convert them to absolute URIs. Any *resolve-to* entry that does include a Protocol (e.g. starting with *file:* or *http:*) is regarded as relative. Otherwise it is treated as absolute.

By default, the Java classpath is searched left to right for a base URI to complete the relative URI in the catalog. However, it is more useful to prepend the classpath with a user-defined list of base locations. The System Property `org.apache.woden.resolver.simpleresolver.baseURIs` may be used to specify such a list.

For example, say we wish to resolve to two files stored on the local file system as `/wsdl/resources/interface.wsdl`, `/xsd/resources/schema.xsd` and one file `/wibble/random.wsdl` contained in a JAR called `/mydocs.jar`.

We set the `org.apache.woden.resolver.simpleresolver.baseURIs` property to the value `file:///wsdl/;file:///xsd/;file:///mydocs.jar`. Note the trailing "/" on the first two semi-colon separated entries which indicates a base URI. If this is omitted the entry is assumed to be a URL of a JAR file. Now we can use the following in the catalog to reference the files:

```
       http\://test.com/importinterface.wsdl=resources/interface.wsdl
       http\://test.com/myschema.xsd=resources/schema.xsd
       http\://test.com/random.wsdl=wibble/random.wsdl
```

Note that when the resolver creates its resolution table, for each relative entry the baseURIs list is searched left-to-right and the first match that references a physical resource is used.

Typically, baseURIs will be set to a single path from which all relative URIs in the catalog descend.

URLs from JAR files

These are references to resources contained within a jar file, and may be used as absolute resolve-to URLs in the catalog.

Example:

```
      http\://test.com/doit.wsdl=jar:file:///wibble/pling.jar!/doit.wsdl
```

2 – Configuration Properties

When a SimpleURIResolver is instantiated, it examines two system properties:

- org.apache.woden.resolver.simpleresolver.catalog
- org.apache.woden.resolver.simpleresolver.baseURIs

The first should contain a URL for the location of the user's catalog file. If this is unset, no URI resolving will occur, except for that defined in the woden schema catalog (see below).

The second is introduced in the discussion on relative URIs above.

An application using the Woden WSDLReader to parse a document might configure the URI resolver as in the flowing snippet:

```
System.setProperty("org.apache.woden.resolver.simpleresolver.catalog",
                          "file:///myplace/myresolves.catalog");
System.setProperty("org.apache.woden.resolver.simpleresolver.baseURIs",
                          "file:///wsdl/;file:///xsd/;file:///mydocs.jar");

      WSDLReader reader = factory.newWSDLReader(); // instantiates the
default resolver
      reader.readWSDL("file:///mydoc.wsdl"); // this is also a candidate
for the resolver
```

3 – Automatic schema resolution - schema.catalog

The Woden *schema catalog* is a predefined catalog which is loaded automatically when a `SimpleURIResolver` is instantiated. It is loaded immediately before the user-defined catalog (if any).

The Woden schema catalog contains resolutions of the standard XML Schema schema, and

the WSDL 2.0 schema, necessary to allow the parser to operate when in network isolation. Because the user catalog is loaded second, it is possible to override schema entries by redefining them there.

The schema catalog is located in `meta-inf/schema.catalog` in the Woden distribution jar.

7. More topics to be added...

This User Guide is a work-in-progress. The content will be expanded and restructured as the development of the Woden project progresses. The following list indicates some topics to be added:

- More on WSDLReader and readWSDL options (including URL resolution)
- Types support (XML Schema)
- Extension architecture (extension elements and attributes, extension registry)
- Validation strategy (validation feature, continue-on-error feature)
- Error reporting, error messages, customizing the ErrorHandler
- creating or modifying WSDL programmatically
- serializing WSDL with WSDLWriter
- How to extend the Woden framework (details of extension points)