IBM i
Version 2.6

Integrated Web Services Server Administration and Programming Guide

Note

Before using this information and the product it supports, read the information in “Notices” on page 155.

Fourth Edition (December 2018)

This edition applies to version 2.6 of Integrated Web Services Server and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

The integrated web services server is integrated in IBM i and is used to externalize integrated language environment (ILE) business logic as a web service. This integration opens the IBM i system to a variety of web service client implementations, including RPG, COBOL, C, C++, Java, .NET, PHP, WebSphere Process Server, ESB, and Web 2.0. This publication describes how to implement web services using the integrated web services server. It starts by describing the concepts of the major building blocks on which web services rely and leading practices for web services applications. It then illustrates how to use tools to build and deploy a web service application.

Who should read this book?

This book is primarily for application programmers who develop web services that are based on the ILE programming languages (e.g. RPG, COBOL, etc.). Some of the information might also be useful to system administrators who manage systems on which web services are developed and deployed.

What you need to know to understand this book

You will need to have access the IBM i Web Administration GUI (`http://<server>:2001/HTTPAdmin`) in order to create an integrated web services server and deploy web services to the server. In addition, application programming skills in one of the programming languages from the list below:

- RPG
- Cobol
- C

Conventions used in this book

Italics is used for new terms where they are defined.

`Constant width` is used for:

- Program language code listings
- WSDL file listings
- XML listings
- Command lines and options

`Constant width italic` is used for replaceable items in code or commands.

In addition, in order to simplify paths when referring to files or commands in the integrated web services server install directory, `/QIBM/ProdData/OS/WebServices`, we will use `<install_dir>` as the initial path in path names to represent the install directory.

About examples in this book

Examples used in this book are kept simple to illustrate specific concepts. Some examples are fragments that require additional code to work.

What has changed in this document

As new features and enhancements are made, the information in this document will get updated. To use any new features or enhancements you should load the latest HTTP Group PTF for your IBM i release. To see what HTTP Group PTF a feature or enhancement is in, go to the IBM Integrated Web Services for i Technology Updates wiki, at URL:

```
http://www.ibm.com/developerworks/ibmi/techupdates/iws
```

Notes:

1. Sometimes new features or enhancements are not yet part of a group PTF, in which case the wiki will list the PTF number(s) containing the feature or enhancement.
2. To help you see where technical changes have been made since the previous edition, the character | is used to mark new and changed information.

The following lists the changes that have been made to the book since the previous edition:

- **December 2018**
  - The "setWebServiceProperties.sh command" on page 119 has been updated to allow for the ability to change the program object path to the web service implementation code.
  - The "installWebService.sh command" on page 110 command has been updated with new options to indicate whether SOAPAction HTTP header should be set in addition to whether elements should be namespace qualified for SOAP services.
  - The "restoreWebServices.sh command" on page 115 has been updated with a new option to help in the migration of web services from version 1.3 of the server to the most current version of the server.
- **June 2018**
  - Previously, the support for the date and dateTime types was inconsistent and allowed values that did not conform with the standards. To rectify this, the support for the date and dateTime types has been rewritten to ensure consistency and that the values passed to the web service conforms with the standard. In addition, support for the time type has been added. For more information, see "Date and time types" on page 134. The new support only applies to web services running on version 2.6 of the server or newer.
  - Important information in the usage notes for the "restoreWebServices.sh command" on page 115 has been added regarding the migration of web services from old versions (1.3, 1.5) of the server to the most current version of the server.
- **February 2018**
  - New information to allow web services to be run under authenticated user ID via Qshell scripts has been added. For more information, see the documentation updates for the following scripts: "installWebService.sh command" on page 110 and "setWebServiceProperties.sh command" on page 119.
  - New information to allow the pre-initialization of the host server connection pool for a web service via Qshell scripts has been added. For more information, see the documentation updates for the following script: "setWebServiceProperties.sh command" on page 119.
- **February 2017**
  - New information on required licensed program products that need to be installed on your server. For more information, see Chapter 5, "Integrated web services server installation details," on page 55.
  - New information has been added regarding the Swagger document enhancements in the integrated web services server throughout the document. More information on Swagger may be found in "Swagger primer" on page 42.
  - Support for 2-tier web services, where the server is on one IBM i system and the web service implementation code (i.e. ILE program or service program) resides on a remote IBM i system.
  - Information on programming considerations when dealing with various data types has been added, see "Data type considerations" on page 134.

Contents

Part 1. Web service fundamentals

This part of the document introduces web service concepts and architecture, including a discussion on the core technologies that form the basis of web services.

Chapter 1. What is a web service?

A *web service* enables the sharing of logic, data, and processes across networks using a programming interface.

Some of the key features of web services are the following:

- Web services are self-contained.

  On the client side, no additional software is required. A programming language with XML (Extensible Markup Language) and HTTP client support, for example, is enough to get you started. On the server side, merely a web server or application sever is required. It is possible to web service enable an existing application without writing a single line of code.

- Web services are self-describing.

  Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely coupled application integration). The definition of the message format travels with the message. No external metadata repositories is required.

- Web services are modular.

  Web services are a technology for deploying and providing access to business functions over the Web; J2EE (Java 2 Enterprise Edition), CORBA (Common Object Request Broker Architecture), and other standards are technologies for implementing these web services.

- Web services can be published (externalized), located, and invoked across the Web.

  All you need to access the web service from a client perspective is a URI (Uniform Resource Identifier).

- Web services are language independent and interoperable.

  The interaction between a service provider and a service requester is designed to be completely platform and language independent. This interaction requires a document to define the interface and describe the service. Because the service provider and the service requester have no idea what platforms or languages the other is using, interoperability is a given.

- Web services are inherently open and standards based.

  XML, JSON (JavaScript Object Notation) and HTTP are the technical foundation for web services. Using open standards provides broad interoperability among different vendor solutions. These principles mean that companies can implement web services without having any knowledge of the service requesters, and service requesters do not need to know the implementation specifics of service provider applications. This use of open standards facilitates just-in-time integration and allows businesses to establish new partnerships easily and dynamically.

- Web services are composable.

  Simple web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer web services from a web service implementation.

Web services allow applications to be integrated more rapidly, easily and less expensively than ever before. Integration occurs at a higher level in the protocol stack, based on messages centered more on service semantics and less on network protocol semantics, thus enabling loose integration of business functions. These characteristics are ideal for connecting business functions across the Web. They provide a unifying programming model so that application integration inside and outside the enterprise can be done with a common approach, leveraging a common infrastructure. The integration and application of web services can be done in an incremental manner, using existing languages and platforms and by adopting existing legacy applications.

Why web services?

Why should you care about web services? One reason is that web services is well suited to implementing a *Service-Oriented Architecture* (SOA). SOA is a business-centric information technology (IT) architectural approach that supports integrating your business as linked, repeatable business tasks, or services. Within this type of architecture, you can orchestrate the business services in business processes. Adopting the concept of services—a higher-level abstraction that's independent of application or infrastructure IT platform and of context or other services—SOA takes IT to another level, one that's more suited for interoperability and heterogeneous environments.

Because an SOA is built on standards acknowledged and supported by the major IT providers, such as web services, you can quickly build and interconnect its services. You can interconnect between enterprises regardless of their supported infrastructure, which opens doors to delegation, sharing, reuse, and maximizing the benefits of your existing assets.

With an SOA established, you bring your internal IT infrastructure to a higher, more visible, and manageable level. With reusable services and high-level processes, change is easier than ever and is more like disassembling and reassembling parts (services) into new, business-aligned processes. This not only promotes efficiency and reuse, it provides a strong ability to change and align IT with business. Figure 1 on page 4 shows web services in action. The operational systems layer shows the data and applications that contain the information to be delivered as a service. The services layer shows the services that enable the operational layer to be delivered as a service. The business process layer shows how web services can be linked together to create highly flexible and automated business processes. The people and application layer shows how web services are used to create web applications and dashboards. It is all about efficiency in creation, reuse for execution, and flexibility for change and growth.

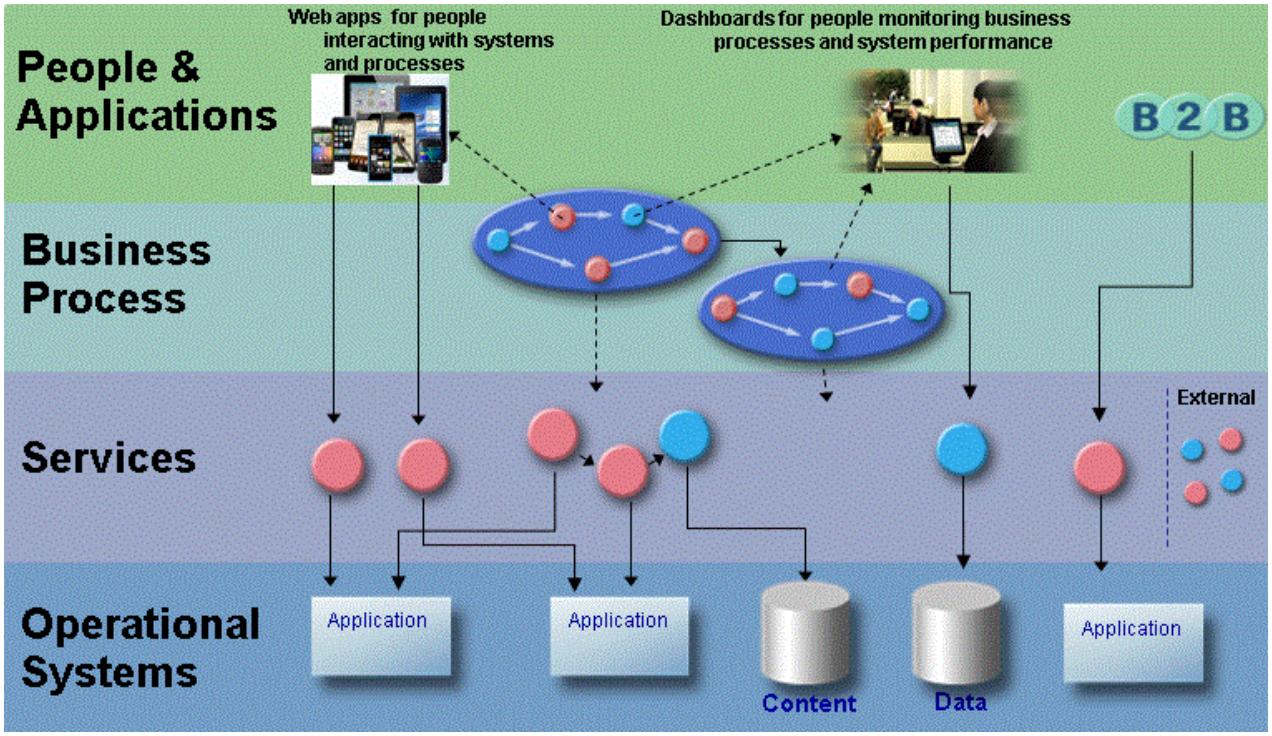


Figure 1: Web services in action

Another reason web services are important is due to web services that is commonly known as web Application Program Interfaces (APIs). An API is a public persona for a company, exposing defined assets, data, or services for public consumption.

In the 1990's when the World Wide Web (WWW) was relatively new many companies focused their business toward creating a web presence. As Internet access became more readily available, speed limitations lifted, and technology improved, many companies migrated from a relatively flat and static web presence to a more dynamic, content rich and interactive approach. Today we live in a data centric

world of connected devices where we expect data to be readily available at our fingertips. These devices include, but are not limited to, smart phones, tablets, games consoles, and even cars and refrigerators. As the number of devices has increased, so too has the complexity to manage and maintain the code for each of these devices and this is where an “API First” approach has really gained the most traction. Exposing the data via a common API allows a single point of maintenance, security, versioning and control. In this way data can be exposed consistently across multiple devices. APIs can help companies expose data that they wish to make available to the outside world or select business partners. These APIs can be used to create applications as well as act as a powerful means to market a company’s product and to help carve out new market opportunities. Once APIs are established they can be used to drive brand awareness and increase profit. Most importantly the APIs, which are a now a core part of the business also need to be treated as a product. Whether or not you or your company are considering exposing APIs it is very likely one of your competitors are. In the highly competitive world we live in today, this in itself is a significant reason to start considering an API strategy.

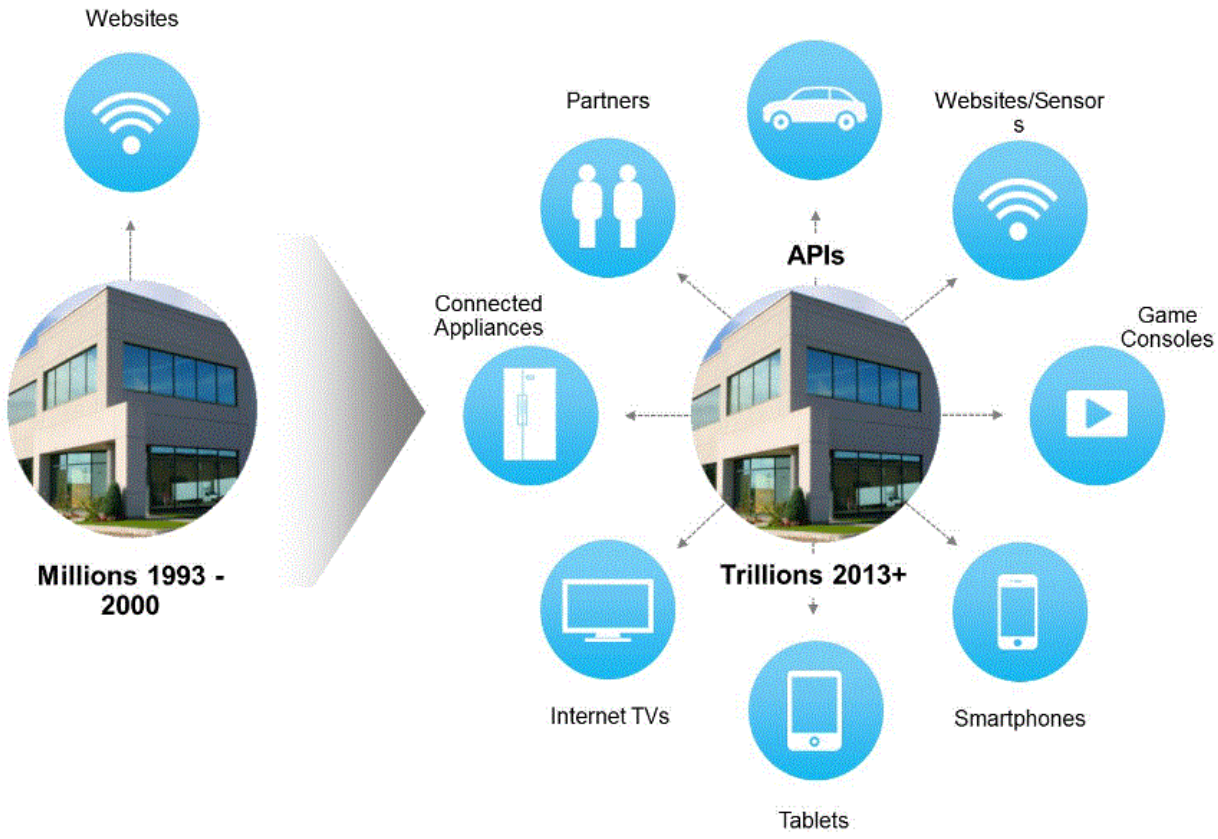


Figure 2: APIs in action

Figure 2 on page 5 shows a fundamental shift from websites as being the information technology access mechanism for the majority of businesses, to the rapidly growing ecosystem of interconnected devices that require APIs to consume business function. Today, we have applications in cars, appliances, smartphones, game consoles, and other devices, that communicate with back-end business functions through APIs. This “interconnected revolution” is here today: refrigerators can tell their manufacturer services systems when maintenance is required; cars can do the same with routine maintenance notification; and smart electric meters can provide usage and consumption information to the utility company.

All of this is possible through web APIs.

Chapter 2. Types of web services

A web service is composed of operations that are offered in one of two styles:

- A web service based on the Service Object Access Protocol (SOAP) protocol.
- A web service that follows the principles of Representational State Transfer (REST).

The following sections discusses each of the types of web services.

SOAP-based web services

A SOAP-based web service is a self-contained software component with a well-defined interface that describes a set of operations that are accessible over the Internet. Extensible Markup Language (XML) technology provides a platform—and programming language-independent means by which a web service's interface can be defined. Web services can be implemented using any programming language, and can be run on any platform, as long as two components are provided to indicate how the web service can be accessed: a standardized XML interface description, called WSDL (Web Services Description Language), and a standardized XML-based protocol, called SOAP (Simple Object Access Protocol). Applications can access a web service by issuing requests formatted according to the XML interface.

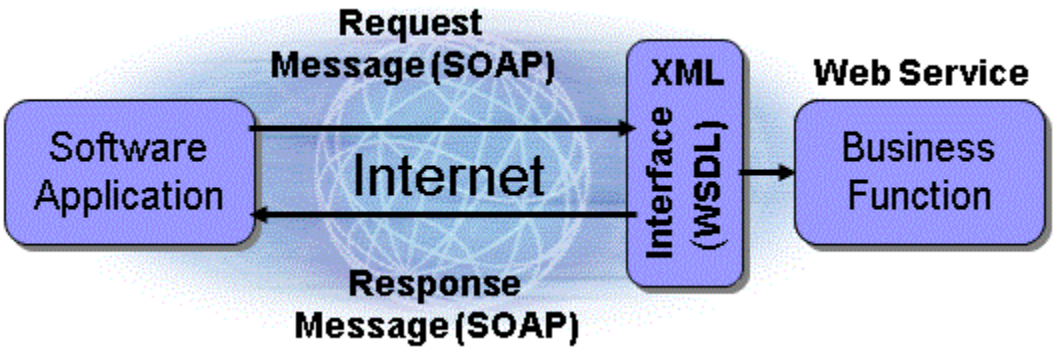


Figure 3: SOAP-based web services

Core technologies and standards

Several key technologies and standards exist within the SOAP-style web services community:

- XML, developed by the World Wide Web Consortium (W3C) for defining markup languages. XML allows the definition, transmission, validation and interpretation of data between applications. It is a meta-language: a language for defining other markup languages, interchange formats and message sets. For information about XML, see “XML primer” on page 8.
- SOAP, a standard protocol for exchanging XML messages. It also details the way applications should treat certain aspects of the message, such as elements in the "header", which enable you to create applications in which a message is passed between multiple intermediaries before reaching its final destination. For information about SOAP, see “SOAP primer” on page 17.
- WSDL, a specification that details a standard way to describe a SOAP-based web service, including the form the messages should take, and where they should be sent. It also details the response to such a message. For information about WSDL, see “WSDL primer” on page 24.

The big interoperability question: can web services continue to interoperate as the various standards they rely on change over time? From a user perspective, the use of arbitrary collections of web services technology should not stand in the way of interoperability between web services.

The WS-I was formed with the intent of promoting standardized interoperability in the web services marketplace. Without a controlled combination of the various technologies that make up web services, interoperability would be almost impossible.

Consider the following: Company X has decided to use WSDL Version 1.2 and SOAP Version 1.1 for their web services. Company Y has decided to use WSDL Version 1.1 and SOAP Version 1.2. Even though both companies are using web services, a client would need to know about the two different combinations of protocols in order to interact with both. The protocols by themselves are not enough to achieve interoperability. A standardized grouping of the protocols would make it possible for Company X, Company Y, and their clients and registries to adopt a common set of protocols and versions. Without a standardized grouping, the companies and clients can only pick what protocols and versions they think are appropriate according to their unique set of constraints or requirements, and hope that they will be able to communicate with each other.

The WS-I Profile initiative addresses the problem that Companies X and Y are facing. A *profile* is a grouping of web services protocols and their versions under a title. By having such a grouping, organizations can negotiate their protocol requirements at more granular levels. Profiles also limit the number of official protocol sets from inestimable to whatever degree of finiteness the WS-I chooses.

As enterprises begin to apply web services technologies to solve their integration and interoperability problems, they increasingly find that they require more advanced features such as security, reliable messaging, management and transactional capabilities. Some of these quality-of-service capabilities demand interoperable infrastructure services for such things as metadata, trust, resource management, event notification, and coordination services. The majority of today's deployed SOAP-style web services are limited to use of only the foundation technologies of SOAP, WSDL, and XML. However, SOAP-style web services provides a broad range of capabilities that compose with the foundation to provide more advanced qualities of service, infrastructure services and service composition. The quality of service extensions to the base SOAP-style web services standards include:

- WS-Addressing, which defines a standardized endpoint reference schema type and a set of message addressing properties that can be used in conjunction with the SOAP process model to effect a broad range of message exchange patterns beyond the simple request/response.
- WS-PolicyFramework, which provides a framework for articulating policy constraints of a service endpoint and a framework for attaching such policy constraints to WSDL and other web services artifacts.
- WS-Security, which provides a framework for an entire family of security specifications including WS-Secure Conversation providing session-based security capabilities and WS-Trust providing a standardized interface to a trust service.
- WS-ReliableMessaging, providing for the reliable exchange of messages between web services endpoints.
- WS-AtomicTransactions, which handles short-lived transactional activities.

For more information on the web services standards, consult an online reference of web services standards, such as is hosted on the IBM®developerWorks® web site, available at:

```
http://www.ibm.com/developerworks/webservices/standards/
```

XML primer

XML stands for Extensible Markup Language and it has become one of the most important standard of modern times. XML is a specification developed by the World Wide Web Consortium (W3C) for defining markup languages. XML allows the definition, transmission, validation and interpretation of data between applications. It is a meta-language: a language for defining other markup languages, interchange formats and message sets. XML is the standard upon which many Web services standards are based and thus we will briefly touch upon some of the more important parts of the specifications as a very quick primer. The entire specification can be studied at the web site of the World Wide Web Consortium at:

```
https://www.w3.org/TR/xml/
```

Basic rules for creating XML documents

Below is an example of an XML document. XML documents are created with three main XML components: *elements*, *attributes* and *"text" contents of the elements*. XML documents should be defined by a corresponding XML definitional document (for example, an XSD) - not shown here - which will be discussed later.

```
<?xml version="1.1"?> 1
<!-- Complete address tag --> 2
<Address>
   <Name> 3
     <Title>Mrs.</Title>
     <First-Name>Ashley</First-Name>
     <Middle-Name/> 4
     <Last-Name>Adams</Last-Name>
     <Phone>777-444-2222</Phone>
   </Name> 5
   <Street>123 Corporation Avenue</Street>
   <City state="NC">Hometown</City> 6
   <Postal-Cde>27709</Postal-Cde>
   <Department>Industrial Design</Department>
</Address>
```

- **XML declaration**: In the above example, line 1 (`<?xml version="1.1"?>`) is the XML declaration that provides basic information about the document to the parser.
- **Tag**: A tag is the text between the left angle bracket (<) and the right angle bracket (>). There are starting tags (such as <Name> on line 3) and ending tags (such as </Name> on line 5).
- **Element**: An element is the starting tag, the ending tag and everything in between. The <Name> element on line 3, contains four child elements: <Title>, <First-Name>, <Middle-Name/> and <Last-Name>.

  Element rules include:

  – There's only one root element in an XML document.
  – The first element is considered the root element. It is also the outermost element, so its end tag is last.
  – Elements must be properly nested and follow well-formed XML code structure.
  – Opening and closing tags cannot cross each other. At any given depth of open tags, it is only valid to close the innermost element (the last one to have been opened at that point).
  – An element does not directly contain characters: consecutive characters are grouped into a “Text” node and the "Text" node is the child of Element. Although "Text" is the official term, schemas can require that a text node actually contain a number, date or other type of data. Schemas can impose similar requirements on attribute values.
- **Attribute**: An attribute is a name-value pair inside the starting tag of an element. On line 6 (`<City state="NC">Hometown</City>`), `state` is an attribute of the <City> element. The `"NC"` is the value of the attribute.

  Attribute rules include:

  – Attributes must have values. However, an attribute can have a value that is an empty string (for example, `<House color="">`.
  – Those values must be enclosed with single or double quotation marks.
- **Comment tag**: Line 2 contains a comment tag. Comments can appear anywhere in the document; they can even appear before or after the root element. A comment begins with `<!--` and ends with `-->`. A comment can not contain a double hyphen (`--`) except at the end; with that exception, a comment can contain anything.
- **Empty element**: An empty element contains no content. Line 4 contains the markup <Middle-Name/>. There is no middle name so it is empty. The markup could also be written as <Middle-Name></Middle-Name>. The shorter version still has an ending tag of `"/>"`. An XML parser would treat them in the same way. If your XML document was referencing an XML schema and the XML

schema was checking for that element, you would make sure that you included that element in your XML document, but leave it empty if you don't have data.

Naming rules for elements and attribute tags

The following are examples of the naming rules for XML (for a complete list of naming rules, see the W3C XML recommendations):

- A name must consist of at least one letter and can be either upper or lower case.
- XML code is case sensitive. <c> and <C> are considered two different tags.
- You can use an underscore (_) as the first character of a name, if the name consists of more than one character.
- Digits can be used in a name after the first character.
- Colons are used to set off the namespace prefix and should not otherwise be used in a name.

Nesting tags

By nesting tags, XML provides you with the ability to describe hierarchical structures as well as sequence. Nesting requirements mean that a well-formed XML document can be treated as a tree structure of elements. Many XML specs will casually refer to the term XML tree when referring to the structure of elements.

Understanding XML namespace

Namespace is a method of qualifying the element and attribute names used in XML documents by associating them with a Universal Resource Identifier (URI). A URI is a string of characters that identifies an Internet Resource (IR). The most common URI is the Uniform Resource Locator (URL), which identifies an Internet domain address along with other system identifiers. Another, not so common, type of URI is the Universal Resource Name (URN).

An *XML namespace* is a collection of names identified by a URI reference, which are used in XML documents and defines the scope of the element and attribute names. Element and attribute names defined in the same namespace must be unique.

An XML document can have a default namespace (using ‘xmlns=') and any element can belong to the default, or another specified namespace. The collection of defined elements and attributes within the same namespace are said to be in the same “XML vocabulary.” The example below shows some examples of the use of namespace:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
<Header>
  <n:AlertControl xmlns:n="http://ibm.com/alertcontrol">
   <n:Priority>1</n:Priority>
  </n:AlertControl>
 </Header>
 <Body>
  <m:Alert xmlns:m="http://ibm.com/alert">
   <m:Msg>Pick up Mary at school at 2pm</m:Msg>
  </m:Alert>
 </Body>
</Envelope>
```

Default Namespaces and Scope

For a namespace definition, a prefix is optional. All elements that are defined without a prefix and appear within the element containing the namespace declaration belong to that default namespace.

A namespace declaration applies to the element that contains the definition as well as its child elements, unless it is overridden by another namespace declaration within the element definition. If we look at the example below, we see that

```
<Books xmlns:BookInfo="http://www.ibm.com/BookInformation"
       xmlns:BookContent="http://www.ibm.com/BookContent"
       xmlns ="http://www.ibm.com/BookDefault" >

   <Book>
      <BookInfo:Name>Understanding Namespaces</BookInfo:Name>
      <Author>Whizlabs</Author>
      <BookInfo:ISBN>s677-898-765-098</BookInfo:ISBN>
      <BookContent:Price>53.50</BookContent:Price>
      <Publisher
          xmlns="http://www.ibm.com/Publishers">Whizlabs</Publisher>
   </Book>
</Books>
```

the following are the element names and the namespaces they belong to:

Table 1: Mapping of element names to namespaces

Element	**Namespace**
<Book>	http://www.ibm.com/BookDefault
<Name>	http://www.ibm.com/BookInformation
<Author>	http://www.ibm.com/BookDefault
<ISBN>	http://www.ibm.com/BookInformation
<Price>	http://www.ibm.com/BookContent
<Publisher>	http://www.ibm.com/Publishers

Attributes

As with elements, you can also qualify attributes by assigning them a prefix that's mapped to a namespace declaration. But attributes behave differently from elements when it comes to the application of namespaces. If an attribute is not qualified with a prefix, it does not belong to any namespace, so default namespace declarations do not apply to attributes.

Definition of XML documents

An *XML schema* is a document that defines constraints for the structure and content of an XML document. This is in addition to the rules imposed by XML itself and should be looked at as a higher level of organizational restriction.

One of the first XML schema definition languages has been the Document Type Definition (DTD) language. Because of its complexity it has been largely replaced by the XML Schema Definition (XSD) specification. XSD allows us to define what elements and attributes may appear in a document, which ones are optional or required and their relationship to each other. It also defines the type of data that can occur in elements and helps define complex data types.

Having an XSD document also allows us to verify an XML document for validity. In addition, you will notice that WSDL documents usually reference the XSD namespace in their `<types>` section and utilize the XSD specification therein to define the input and output messages of the Web Service.

Schema definition

A schema is defined in a separate file and generally stored with the .xsd extension. Every schema definition has a schema root element that belongs to the http://www.w3.org/2001/XMLSchema namespace. The schema element can also contain optional attributes. For example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
```

This indicates that the elements used in the schema come from the http://www.w3.org/2001/XMLSchema namespace.

Schema linking

An XML file links to its corresponding schema using the schemaLocation attribute of the schema namespace. You have to define the schema namespace in order to use the schemaLocation attribute. All of these definitions appear in the root element of the XML document. The syntax is:

```
<ROOT_ELEMENT
    SCHEMA_NAMESPACE_DEFINITION
    SCHEMA_LOCATION_DEFINITION >
```

And here's an example of it in use:

```
<Books
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="http://www.booksforsale.com Books.xsd">
```

Schema elements

A schema file contains definitions for element and attributes, as well as data types for elements and attributes. It is also used to define the structure or the content model of an XML document. Elements in a schema file can be classified as either simple or complex -- defined in “Schema elements - simple types” on page 12 and “Schema Elements - Complex Types” on page 13

Schema elements - simple types

A *simple type* element is an element that cannot contain any attributes or child elements; it can only contain the data type specified in its declaration. The syntax for defining a simple element is:

```
<xs:element name="ELEMENT_NAME" type="DATA_TYPE" default/fixed="VALUE" />
```

Where DATA_TYPE is one of the built-in schema data types (see below).

You can also specify default or fixed values for an element. You do this with either the default or fixed attribute and specify a value for the attribute. The default and fixed attributes are optional.

An example of a simple type element is:

```
<xs:element name="Author" type="xs:string" default="Whizlabs"/>
```

All attributes are simple types, so they are defined in the same way that simple elements are defined. For example:

```
<xs:attribute name="title" type="xs:string" />
```

Schema data types. All data types in schema inherit from anyType. This includes both simple and complex data types. You can further classify simple types into built-in-primitive types and built-in-derived

types. A complete hierarchical diagram from the XML Schema Datatypes Recommendation[1] is shown below:

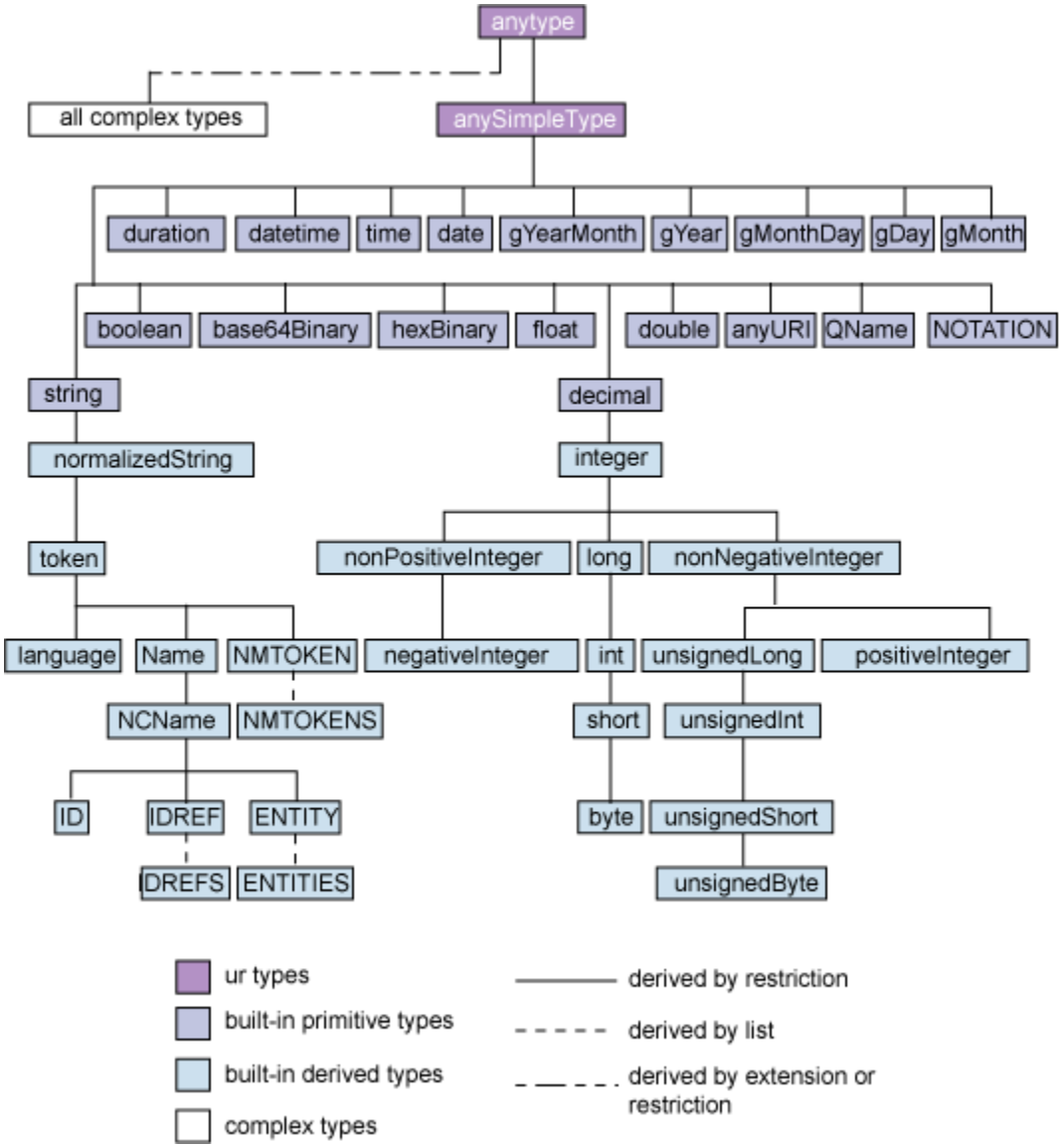


Figure 4: XML schema datatypes

Schema Elements - Complex Types

Complex types are elements that either:

- Contain other elements
- Contain attributes
- Are empty (empty elements)
- Contain text

[1] Copyright 2003 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

To define a complex type in a schema, use a `complexType` element. You can specify the order of occurrence and the number of times an element can occur (cardinality) by using the *order* and *occurrence* indicators, respectively. (See "Occurrence and Order Indicators" on page 14 for more on these indicators.) For example:

```
<xs:element name="Book">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="Name" type="xs:string" />
         <xs:element name="Author" type="xs:string" maxOccurs="4"/>
         <xs:element name="ID" type="xs:string"/>
         <xs:element name="Price" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

In this example, the order indicator is `xs:sequence`, and the occurrence indicator is `maxOccurs` in the `Author` element name.

Occurrence and Order Indicators

Occurrence indicators specify the number of times an element can occur in an XML document. You specify them with the `minOccurs` and `maxOccurs` attributes of the element in the element definition.

As the names suggest, `minOccurs` specifies the minimum number of times an element can occur in an XML document while `maxOccurs` specifies the maximum number of times the element can occur. It is possible to specify that an element might occur any number of times in an XML document. This is determined by setting the `maxOccurs` value to `unbounded`. The default values for both `minOccurs` and `maxOccurs` is `1`, which means that by default an element or attribute can appear exactly one time.

Order indicators define the order or sequence in which elements can occur in an XML document. Three types of order Indicators are:

- **`All`**: If `All` is the order indicator, then the defined elements can appear in any order and must occur only once. Remember that both the `maxOccurs` and `minOccurs` values for `All` are always `1`.
- **Sequence**: If `Sequence` is the order indicator, then the elements must appear in the order specified.
- **`Choice`**: If `Choice` is the order indicator, then any one of the elements specified must appear in the XML document.

Take a look at the following example:

```
<xs:element name="Book">
   <xs:complexType>
      <xs:all>
         <xs:element name="Name" type="xs:string" />
         <xs:element name="ID" type="xs:string"/>
         <xs:element name="Authors" type="authorType"/>
         <xs:element name="Price" type="priceType"/>
      </xs:all>
   </xs:complexType>
</xs:element>

<xs:complexType name="authorType">
      <xs:sequence>
         <xs:element name="Author" type="xs:string" maxOccurs="4"/>
      </xs:sequence>
</xs:complexType >

<xs:complexType name="priceType">
   <xs:choice>
      <xs:element name="dollars" type="xs:double" />
      <xs:element name="pounds" type="xs:double" />
   </xs:choice>
</xs:complexType >
```

In the above example, the `xs:all` indicator specifies that the `Book` element, if present, must contain only one instance of each of the following four elements: `Name`, `ID`, `Authors`, `Price`. The `xs:sequence`

indicator in the `authorType` declaration specifies that elements of this particular type (`Authors` element) contain at least one `Author` element and can contain up to four `Author` elements. The `xs:choice` indicator in the `priceType` declaration specifies that elements of this particular type (`Price` element) can contain either a `dollars` element or a `pounds` element, but not both.

Restriction

A main advantage of schema is that you have the ability to control the value of XML attributes and elements. A *restriction*, which applies to all of the simple data elements in a schema, allows you to define your own data type according to the requirements by modifying the *facets* (restrictions on XML elements) for a particular simple type. To achieve this, use the `restriction` element defined in the schema namespace.

W3C XML Schema defines 12 facets for simple data types. The following list includes each facet, along with its effect on the data type value and an example.

- **`enumeration`**: Value of the data type is constrained to a specific set of values. For example:

```
<xs:simpleType name="Subjects">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Biology"/>
    <xs:enumeration value="History"/>
    <xs:enumeration value="Geology"/>
  </xs:restriction>
</xs:simpleType>
```

- **`maxExclusive`**: Numeric value of the data type is less than the value specified.

  `minExclusive` Numeric value of the data type is greater than the value specified. For example:

```
<xs:simpleType name="id">
  <xs:restriction base="xs:integer">
    <xs:maxExclusive value="101"/>
    <xs:minExclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
```

- **`maxInclusive`** - Numeric value of the data type is less than or equal to the value specified.

  `minInclusive` - Numeric value of the data type is greater than or equal to the value specified. For example:

```
<xs:simpleType name="id">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>
```

- **`maxLength`** - Specifies the maximum number of characters or list items allowed in the value.

  `minLength` - Specifies the minimum number of characters or list items allowed in the value.

  `pattern` - Value of the data type is constrained to a specific sequence of characters that are expressed using regular expressions. For example:

```
<xs:simpleType name="nameFormat">
  <xs:restriction base="xs:string">
    <xs:minLength value="3"/>
    <xs:maxLength value="10"/>
    <xs:pattern value="[a-z][A-Z]*"/>
  </xs:restriction>
</xs:simpleType>
```

- **`length`** - Specifies the exact number of characters or list items allowed in the value. For example:

```
<xs:simpleType name="secretCode">
  <xs:restriction base="xs:string">
    <xs:length value="5"/>
  </xs:restriction>
</xs:simpleType>
```

- **whiteSpace** - Specifies the method for handling white space. Allowed values for the value attribute are preserve, replace, and collapse. For example:

```
<xs:simpleType name="FirstName">
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="preserve"/>
  </xs:restriction>
</xs:simpleType>
```

- **fractionDigits** - Constrains the maximum number of decimal places allowed in the value.

  totalDigits - The number of digits allowed in the value. For example:

```
<xs:simpleType name="reducedPrice">
  <xs:restriction base="xs:float">
    <xs:totalDigits value="4"/>
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Extension

The `extension` element defines complex types that might derive from other complex or simple types. If the base type is a simple type, then the complex type can only add attributes. If the base type is a complex type, then it is possible to add attributes and elements. To derive from a complex type, you have to use the `complexContent` element in conjunction with the `base` attribute of the `extension` element.

Extensions are particularly useful when you need to reuse complex element definitions in other complex element definitions. For example, it is possible to define a `Name` element that contains two child elements (`First` and `Last`) and then reuse it in other complex element definitions. Here is an example:

```
<!--Base element definition -->
<xs:complexType name="Name">
   <xs:sequence>
      <xs:element name="First"/>
      <xs:element name="Last"/>
   </xs:sequence>
</xs:complexType>

<!-- Customer element that reuses it -->
  <xs:complexType name="Customer">
    <xs:complexContent>
      <xs:extension base="Name">
        <xs:sequence>
          <xs:element name="phone" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Student element that reuses it -->
  <xs:complexType name="Student">
    <xs:complexContent>
      <xs:extension base="Name">
        <xs:sequence>
          <xs:element name="school" type="xs:string"/>
          <xs:element name="year" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

Import and Include

The `import` and `include` elements help to construct a schema from multiple documents and namespaces. The `import` element brings in a schema from a different namespace, while the `include` element brings in a schema from the same namespace.

When you use `include`, the target namespace of the included schema must be the same as the target namespace of the including schema. In the case of `import`, the target namespace of the included schema must be different from the target namespace of the including schema.

The syntax for `import` is:

```
<xs:import id="ID_DATATYPE" namespace="anyURI_DATATYPE"
    schemaLocation="anyURI_DATATYPE "/>
```

The syntax for `include` is:

```
<xs:include id="ID_DATATYPE" schemaLocation="anyURI_DATATYPE"/>
```

SOAP primer

SOAP is defined independently of any operating system or protocol and provides a way to communicate between applications running on different computers, using different operating systems, and with different technologies and programming languages as long as the SOAP request and response messages match the message formats that are defined in the WSDL document.

SOAP consists of three parts: An *envelope* that defines a framework for describing message content and process instructions, a set of *encoding rules* for expressing instances of application-defined data types, and a *convention* for representing remote procedure calls and responses.

SOAP is, in principle, transport protocol-independent and can, therefore, potentially be used in combination with a variety of protocols such as HTTP, JMS, SMTP, or FTP. Right now, the most common way of exchanging SOAP messages is through HTTP.

There are two versions of SOAP: SOAP 1.1 and SOAP 1.2. Both SOAP 1.1 and SOAP 1.2 are W3C standards. web services can be deployed that support not only SOAP 1.1 but also support SOAP 1.2. SOAP 1.2 provides a more specific definition of the SOAP processing model, which removes many of the ambiguities that sometimes led to interoperability problems in the absence of the Web Services-Interoperability (WS-I) profiles.

The following sections will cover the SOAP 1.1 specification and the SOAP architecture in detail. For more information on SOAP (including SOAP 1.2), go to the following URL:

```
https://www.w3.org/TR/soap/
```

SOAP message structure

A SOAP message, which is an XML document based on the SOAP protocol, consists of four parts:

1. The SOAP <Envelope> element, the root element of a SOAP message, contains an optional SOAP header and mandatory SOAP body elements. The SOAP protocol namespace prefix (http://schemas.xmlsoap.org/soap/envelope/) is usually declared in the envelope open tag.
2. The optional and extensible <Header> element describes metadata, such as security, transaction, and conversational-state information.
3. The mandatory <Body> element contains the XML document of the sender. The sender's XML document must not contain an XML declaration or DOCTYPE declaration. There are two main paradigms which the sender's document can adhere to: document-style or RPC-style (more about these later). The serialization rules for the contents of the body can be specified by setting the `encodingStyle` attribute. The standard SOAP encoding namespace is http://schemas.xmlsoap.org/soap/encoding/.
4. Elements called <faults> can be used by a processing node (SOAP intermediary or ultimate SOAP destination) to describe any exceptional situations it could encounter that might occur while reading the SOAP message.

The following sections discusses the major elements of a SOAP message.

Namespaces

The use of namespaces plays an important role in SOAP message, because a message can include several different XML elements that must be identified by a unique namespace to avoid name collision. Especially, the WS-I Basic Profile 1.0 requires that all application-specific elements in the body must be namespace qualified to avoid name collision. Table 2 on page 18 shows the namespaces of SOAP and WS-I Basic Profile 1.0.

Table 2: SOAP namespaces

Prefix	Namespace URI	Explanation
SOAP-ENV	http://schemas.xmlsoap.org/soap/envelope/	SOAP 1.1 envelope namespace
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/	SOAP 1.1 encoding namespace
	http://www.w3.org/2001/XMLSchema-instance	Schema instance namespace
	http://www.w3.org/2001/XMLSchema	XML Schema namespace
	http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL framework
	http://schemas.xmlsoap.org/wsdl/soap	WSDL namespace for WSDL SOAP binding

URN

A *unified resource name* (URN) uniquely identifies the service to clients. It must be unique among all services deployed in a single SOAP server, which is identified by a certain network address. A URN is encoded as a *universal resource identifier* (URI).

All other addressing information is transport dependent. For example, when using HTTP as the transport, the URL of the HTTP request points to the SOAP server instance on the destination host.

The SOAP envelope

The basic unit of a web service message is the actual SOAP envelope (see Figure 5 on page 18). This is an XML document that includes all of the information necessary to process the message.

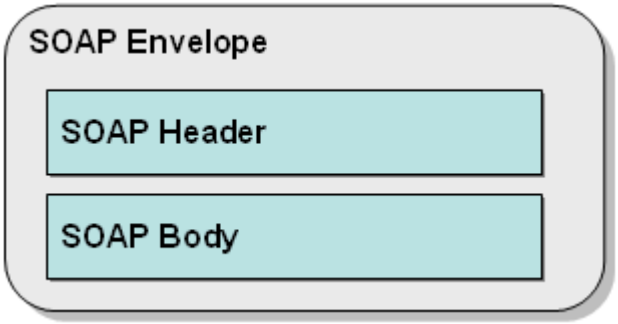


Figure 5: SOAP envelope

A SOAP message is a (possibly empty) set of headers plus one body. The `Envelope` element is the root element of any SOAP message. Generally, it contains the definition for the required envelope namespace. For example:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header>
 </env:Header>
 <env:Body>
 </env:Body>
</env:Envelope>
```

In the example above, you have a simple `Envelope`, with the namespace specified as SOAP version 1.1. It includes two sub elements, a `Header` and a `Body`.

Let's look at what each of those elements do.

The SOAP header

The `Header` in a SOAP message is meant to provide information about the message itself, as opposed to information meant for the application. For example, the `Header` might include routing information, as it does in this example shown below:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header>
   <wsa:ReplyTo xmlns:wsa=
      "http://schemas.xmlSOAP.org/ws/2004/08/addressing">
     <wsa:Address>
       http://schemas.xmlSOAP.org/ws/2004/08/addressing/role/anonymous
     </wsa:Address>
   </wsa:ReplyTo>
   <wsa:From>
     <wsa:Address>
       http://localhost:8080/axis/services/MyService</wsa:Address>
     </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
 </env:Header>
 <env:Body>
 </env:Body>
</env:Envelope>
```

In this case you see a WS-Addressing element, which includes information on where the message is going and to where replies should go.

Headers are optional elements in the envelope. If present, the element must be the first immediate child element of a SOAP envelope element. All immediate child elements of the header element are called *header entries*.

As has been previously stated, headers can include all kinds of information about the message itself. In fact, the SOAP specification spends a great deal of time on elements that can go in the `Header`, and how they should be treated by *SOAP intermediaries* (applications that are capable of both receiving and forwarding SOAP messages on their way to the final destination). In other words, the SOAP specification makes no assumption that the message is going straight from one point to another, from client to server. It allows for the idea that a SOAP message might actually be processed by several intermediaries, on its way to its final destination, and the specification is very clear on how those intermediaries should treat information they find in the `Header`. That discussion is beyond the scope of this document. However, there are two predefined header attributes that you should be aware of: `SOAP-ENV:mustUnderstand` and `SOAP-ENV:actor`.

The header attribute `SOAP-ENV:mustUnderstand` is used to indicate to the service provider that the semantics defined by the element must be implemented. The value of the `mustUnderstand` attribute is either 1 or 0 (the absence of the attribute is semantically equivalent to the value 0):

```
<thens:qos xmlns:thens="someURI" SOAP-ENV:mustUnderstand="1">3</thens:qos>
```

In the example above, the header element specifies that a service invocation must fail if the service provider does not support the quality of service (`qos`) 3 (whatever `qos=3` stands for in the actual invocation and servicing context).

The header attribute `SOAP-ENV:actor` is used to identify the recipient of the header information. The value of the SOAP `actor` attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator does not forward the header). If the `actor` is omitted or set to the predefined default value, the header is for the actual recipient and the actual recipient is also the final destination of the message (body). The predefine value is: `http://schemas.xmlsoap.org/soap/actor/next`. If a node on the message path does not recognize a `mustUnderstand` header and the node plays the role specified by the `actor` attribute, the node must generate a SOAP `mustUnderstand`

fault (more on faults later). Whether the fault is sent back to the sender depends on the message exchange pattern (e.g. request/response) in use.

Now let's look at the actual payload.

The SOAP body

When you're sending a SOAP message, you're doing it with a reason in mind. You are trying to tell the receiver to do something, or you're trying to impart information to the server. This information is called the "payload". The payload goes in the `Body` of the `Envelope`. It also has its own namespace, in this case corresponding to the content management system. The choice of namespace, in this case, is completely arbitrary. It just needs to be different from the SOAP namespace. For example:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header>
...
</env:Header>
<env:Body>
  <cms:addArticle xmlns:cms="http://www.ibm.com/cms">
    <cms:category>classifieds</category>
    <cms:subcategory>forsale</cms:subcategory>
    <cms:articleHeadline></cms:articleHeadline>
    <cms:articleText>Vintage 1963 T-Bird.</cms:articleText>
  </cms:addArticle>
</env:Body>
</env:Envelope>
```

In this case, you have a simple payload that includes instructions for adding an article to the content management system.

The body element is encoded as an immediate child element of the SOAP envelope element. If a header element is present, then the body element must immediately follow the header element. Otherwise it must be the first immediate child element of the envelope element. All immediate child elements of the body element are called body entries, and each body entry is encoded as an independent element within the SOAP body element. In the most simple case, the body of a basic SOAP message consists of:

- A message name.
- A reference to a service instance.
- One or more parameters carrying values and optional type references.

Typical uses of the body element include invoking RPC calls with appropriate parameters, returning results, and error reporting. Fault elements are used in communicating error situations.

The choice of how to structure the payload involves the style and encoding.

Error handling (SOAP faults)

SOAP itself predefines one body element, which is the *fault element* used for reporting errors. If present, the fault element must appear as a body entry and must not appear more than once within a body element.

The XML elements inside the SOAP fault element are different in SOAP 1.1 and SOAP 1.2. In SOAP 1.1, the `<Fault>` element contains the following elements:

- `<faultcode>` is a mandatory element in the `<Fault>` element. It provides information about the fault in a form that can be processed by software. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
  - `soapenv:Client`, indicating incorrectly formatted messages
  - `soapenv:Server`, for delivery problems
  - `soapenv:VersionMismatch`, which can report any invalid namespaces for envelope element

  - soapenv:MustUnderstand, for errors regarding the processing of header content
- <faultstring> is a mandatory element in the <Fault> element. It provides information about the fault in a form intended for a human reader.
- <faultactor> contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the <faultactor> element when it creates a fault. An ultimate SOAP receiver is not obliged to include this element, but may do so.
- <detail> carries application-specific error information related to the <Body> element. It must be present if the contents of the <Body> element were not successfully processed. It must not be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries must be carried in header entries.

In SOAP 1.2, the <Fault> element contains the following elements:

- <Code> is a mandatory element in the <Fault> element. It provides information about the fault in a form that can be processed by software. It contains a <Value> element and an optional <Subcode> element.
- <Reason> is a mandatory element in the <Fault> element. It contains one or more <Text> elements, each of which contains information about the fault in a different native language.
- <Node> contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the <Node> element when it creates a fault. An ultimate SOAP receiver is not obliged to include this element, but may do so.
- <Role> contains a URI that identifies the role in which the node was operating at the point the fault occurred.
- <Detail> is an optional element, which contains application-specific error information related to the SOAP fault codes describing the fault. The presence of the <Detail> element has no significance regarding which parts of the faulty SOAP message were processed.

Here is an example of a SOAP 1.1 fault response message:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
<m:Order xmlns:m="some URI" SOAP-ENV:mustUnderstand="1">
</m:Order>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Not necessary information</faultstring>
  <detail>
      <d:faultdetail xmlns:d = "uri-referrence">
          <msg>application is not responding properly.    </msg>
          <errorcode>12</errorcode>
      </d:faultdetail>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Data model

One of the promises of SOAP is interoperability between different programming languages. That is the purpose of the SOAP data model, which provides a language-independent abstraction for common programming language types. It consists of:

- **Simple XSD types**: Basic data types found in most programming languages such as int, float, and null-terminated character data (i.e. strings).
- **Compound types**: There are two kinds of compound types, *structs* and *arrays*:
  - Structs are named aggregated types. Each element has a unique name, its *accessor*. An accessor is an XML tag. Structs are conceptually similar to records in languages, such as RPG, or method-less classes with public data members in object-based programming languages.

– Elements in an array are identified by position, not by name. Array values can be structs or other compound values. Also, nested arrays (which means arrays of arrays) are allowed.

Let us take a look at an example. Below is a XML schema of a compound datatype named `Mobile`.

```
<? xml version="1.0" ?>
<xsd:schema xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema"
 targetNameSpace= "www.mobilephoneservice.com/phonequote">
 <xsd:element name ="Mobile"> 1
   <xsd:complexType> 2
       <xsd:element name="modelNumber" type="xsd:int"> 3
       <xsd:element name="modelName"  type="xsd:string"> 4
        <xsd:element name="modelColor"> 5
           <simpleType base="xsd:string">
             <enumeration value="blue" />
             <enumeration value="black" />
           </simpleType>
       </xsd:element>
   </complexType>
 </xsd:element>
</xsd:schema>
```

In the listing above, line 1 shows the name (`Mobile`) of our type while line 2 acknowledges that it is a complex datatype that contains sub-elements named `modelNumber`, `modelName` and `modelColor`. The sub-element defined in line 3, `modelNumber`, has a type of `int` (that is, `modelNumber` can take only integer values). The sub-element defined in line 4 is named `modelName` and is of type `string`. The sub-element defined in line 5 requires a bit more understanding since it has a sub element named `simpleType`. Here you are defining a simple type inside the complex type, `Mobile`. The name of your `simpleType` is `modelColor` and it is an `enumeration`. It has an attribute, `base`, carrying the value `xsd:string`, which indicates that the simple type `modelColor` has the functionality of the `string` type defined in the SOAP schema. Each <enumeration> tag carries an attribute, `value` (`blue` and `black`). The enumerated types enable us to select one value from multiple options. Now let us look at how this translates into a SOAP message.

The listing below is demonstrates the use of compound types in a SOAP message. It shows an envelope carrying a request in the `Body` element, in which you are calling the `addModel` method of an `m` namespace. The listing uses the data type `Mobile` that was defined above. The `AddModel` method takes an argument of type `Mobile`. We're referring `Mobile` structure with `msd` namespace reference (see the `xmlns:msd` declaration in <SOAP-ENV:Envelope> element). This is an example of employing user defined data types in SOAP requests.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:msd="www.ibm.com/phonequote">
 <SOAP-ENV:Body>
     <m:addModel xmlns:m="www.ibm.com">
       <msd:Mobile>
      <modelNumber>1</modelNumber>
      <modelName>mlr97</modelName>
      <modelColor>blue</modelColor>
       </msd:Mobile>
  </m:addModel>
 </SOAP-ENV:Body>
<SOAP-ENV:Envelope>
```

SOAP binding and encoding styles

You'll get deeper into this subject in “WSDL primer” on page 24, but as you create your application, you will need to decide[2] on the structure of the actual payload you're sending back and forth. To that end, let's take this opportunity to discuss SOAP binding (also referred as programming or communication binding) and encoding styles.

[2] Well, in the case of integrated web services support, the decision has been made for you! But for completeness we discuss what is available. Integrated web services for i only supports *Document/Literal*. To understand what that means, read on.

To simplify the discussion, the following XML message payload is used as an example:

```
<article>
  <category>classifieds</category>
  <subcategory>forsale</subcategory>
  <articleText>Vintage 1963 T-Bird.</articleText>
</article>
```

This piece of XML payload can be presented in a SOAP message in two different styles: Remote Procedure Calls (RPC) and document. RPC style SOAP describes the semantics of a procedure call and its return value. In this style, the idea is that you're sending a command to the server, such as "add an article", and you're including the parameters command, such as the article to add and the category to which it should as child elements of the overall method. This programming style thus adds extra elements to the SOAP XML to simulate a method call (i.e. the XML payload is wrapped inside an operation element in a SOAP body). A document style message, on the other hand, has the XML payload directly placed in a SOAP body. Document style SOAP is described as being one-way or asynchronous, as there is not a concept of a call and return as in the RPC model. Basically, a document-style message lets you describe an arbitrary XML document using SOAP.

Both the RPC and document message can be either a literal or encoded message. A *literal* message implies that a schema is utilized to provide a description and constraint for an XML payload in SOAP. An Encoded message implies that the message includes type information. Let us look at some examples.

The example below is a typical RPC/literal example.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header></env:Header>
  <env:Body>
    <addArticle>
      <article>
         <category>classifieds</category>
         <subcategory>forsale</subcategory>
         <articleText>Vintage 1963 T-Bird.</articleText>
      </article>
    </addArticle>
  </env:Body>
</env:Envelope>
```

The `addArticle` element is the operation to be invoked. The element `article` (which contains sub-elements`category`, `subcategory`, and `articleText`) is the input parameters to the operation.

If we include type information in the message as in the example below, we have and example of an RPC/encoded message.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header></env:Header>
  <env:Body>
    <addArticle>
      <article>
        <category xsi:type="xsd:string">classifieds</category>
        <subcategory xsi:type="xsd:string">forsale</subcategory>
        <articleText xsi:type="xsd:string">Vintage 1963 T-Bird.</articleText>
      </article>
    </addArticle>
  </env:Body>
</env:Envelope>
```

A document/literal style of message simply involves adding the message:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header></env:Header>
  <env:Body>
      <article>
         <category>classifieds</category>
         <subcategory>forsale</subcategory>
         <articleText>Vintage 1963 T-Bird.</articleText>
      </article>
  </env:Body>
</env:Envelope>
```

In this case, the message itself doesn't include information on the process to which the data is to be submitted; that is handled by the routing software. For example, all calls to a particular URL or endpoint might point to a particular operation.

Finally, you could technically use the document/encoded style, but nobody does, so for now, ignore it.

Different trade-offs are involved with each of these styles. However, the Encoded style has been a source of interoperability problems and is not WS-I compliant, so should be avoided. Although RPC/literal has its usefulness, the most popular form of binding and encoding styles has become document/literal. The document/literal style goes a long way in eliminating interoperability problems, and also has proven to be a good performer while generating the least complex SOAP message.

SOAP response messages

In the previous section the discussion has been about request messages. But what about response messages? What do they look like? By now it should be clear to you what the response message looks like for a document/literal message. The contents of the `soap:body` are fully defined by a schema, so all you have to do is look at the schema to know what the response message looks like.

But what is the child of the `soap:body` for the RPC style responses? The WSDL 1.1 specification is not clear. But WS-I comes to the rescue. WS-I's Basic Profile dictates that in the RPC/literal response message, the name of the child of `soap:body` is "... the corresponding `wsdl:operation` name suffixed with the string 'Response'." For more information on `wsdl:operation`, see “WSDL primer” on page 24.

WSDL primer

WSDL (Web Services Description Language) is an XML document for describing web services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented (RPC) messages. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints or services. WSDL is extensible to allow description of endpoints and their messages, regardless of what message formats or network protocols are used to communicate. Some of the currently described bindings are for SOAP 1.1, HTTP POST, and Multipurpose Internet Mail Extensions (MIME).

There are two versions of the WSDL: WSDL 1.1 and WSDL 2.0. The changes in WSDL 2.0 are generally made for the purposes of interoperability - constructs that are not legal under WS-I's Basic Profile are generally forbidden - or to make it easier to use WSDL with extended SOAP specifications.

The rest of the discussion in this chapter will be from the perspective of the WSDL 1.1 specification. Information on WSDL 1.1 and WSDL 2.0 can be found at the following URLs:

```
https://www.w3.org/TR/wsdl

https://www.w3.org/TR/wsdl20-primer/
```

WSDL 1.1 document structure

WSDL conventionally divides the basic service description into two parts (see Figure 6 on page 25): the service interface and the service implementation. This enables each part to be defined separately and independently, and reused by other parts.

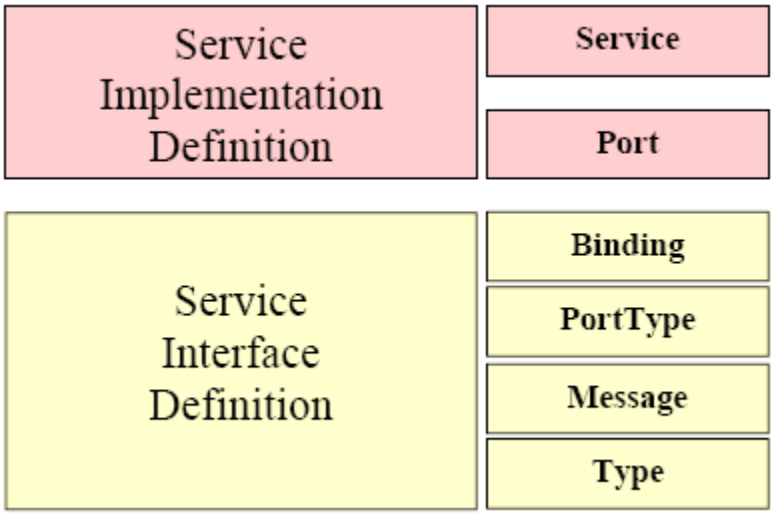


Figure 6: Basic service description

A *service interface definition* is an abstract or reusable service definition that can be instantiated and referenced by multiple service implementation definitions. Think of a service interface definition as an Interface Definition Language (IDL), Java™ interface or web service type. This allows common industry-standard service types to be defined and implemented by multiple service implementers. This is analogous to defining an abstract interface in a programming language and having multiple concrete implementations. The service interface contains WSDL elements that comprise the reusable portion of the service description:

- **`binding`**: Describes the protocol, data format, security and other attributes for a particular service interface (i.e. `portType`).
- **`portType`**: Defines Web service operations. The operations define what XML messages can appear in the input and output data flows. Think of an operation as a method signature in a programming language.
- **`message`**: Specifies which XML data types constitute various parts of a message and is used to define the input and output parameters of an operation.
- **`type`**: Describes the use complex data types within the message.

The *service implementation definition* describes how a particular service interface is implemented by a given service provider. A web service is modeled as a `service` element. A `service` element contains a collection (usually one) of `port` elements. A port associates an endpoint (for example, a network address location or URL) with a `binding` element from a service interface definition.

The service interface definition together with the service implementation definition makes up a complete WSDL definition of the service. This pair contains sufficient information to describe to the service requestor how to invoke and interact with the web service. Now lets dive into the details.

Figure 7 on page 26 shows the elements comprising a WSDL document and the various relationships between them.

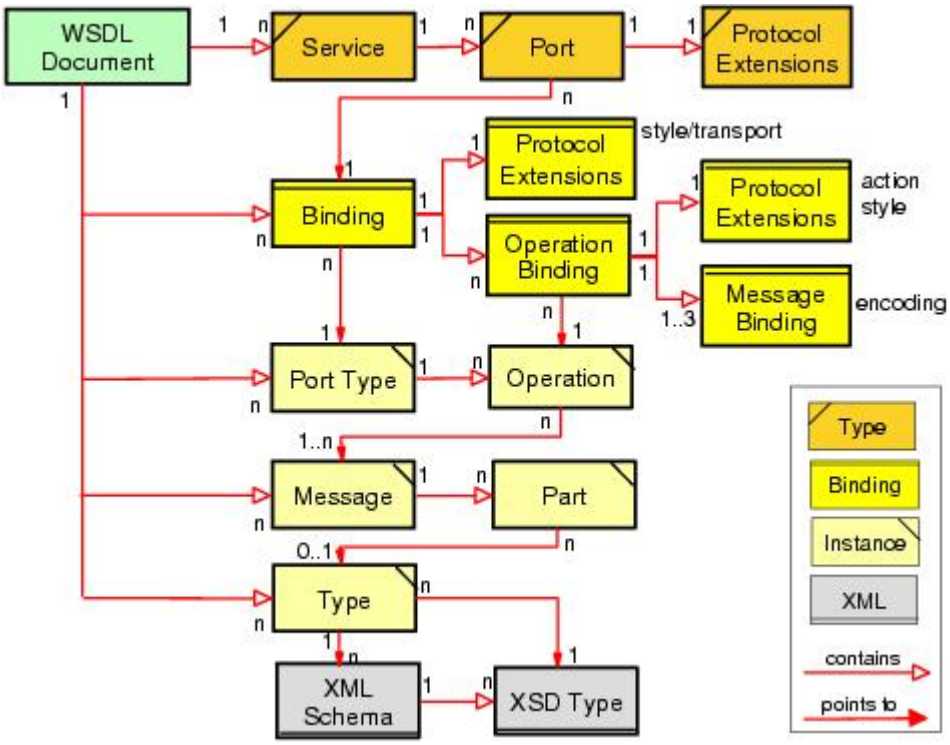


Figure 7: WSDL 1.1 elements and relationships

The diagram should be read in the following way:

- One WSDL document contains zero or more services. A service contains zero or more port definitions (service endpoints), and a port definition contains a specific protocol extension.
- The same WSDL document contains zero or more bindings. A binding is referenced by zero or more ports. The binding contains one protocol extension, where the style and transport are defined, and zero or more operations bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three messages bindings, where the encoding is defined.
- The same WSDL document contains zero or more port types. A port type is referenced by zero or more bindings. This port type contains zero or more operations, which are referenced by zero or more operations bindings.
- The same WSDL document contains zero or more messages. An operation usually points to an input and an output message, and optionally to some faults. A message is composed of zero or more parts.
- The same WSDL document contains zero or more types. A type can be referenced by zero or more parts.
- The same WSDL document points to zero or more XML Schemas. An XML Schema contains zero or more XSD types that define the different data types.

The containment relationships shown in the diagram directly map to the XML Schema for WSDL.

Below is an example of a simple, complete, and valid WSDL file. As we will see, even a simple WSDL document contains quite a few elements with various relationships to each other.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://address.samples"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://address.samples"
    xmlns:intf="http://address.samples"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
   <schema elementFormDefault="qualified"
       targetNamespace="http://address.samples"
```

```
            xmlns="http://www.w3.org/2001/XMLSchema">

    <complexType name="AddressBean">
     <sequence>
      <element name="street" type="xsd:string"/>
      <element name="zipcode" type="xsd:int"/>
     </sequence>
    </complexType>

    <element name="AddressBean" type="impl:AddressBean"/>
  </schema>
 </wsdl:types>

 <wsdl:message name="updateAddressRequest">
   <wsdl:part name="in0" type="intf:AddressBean"/>
   <wsdl:part name="in1" type="xsd:int"/>
 </wsdl:message>
 <wsdl:message name="updateAddressResponse">
   <wsdl:part name="return" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="updateAddressFaultInfo">
   <wsdl:part name="fault" type="xsd:string"/>"
 </wsdl:message>

 <wsdl:portType name="AddressService">
   <wsdl:operation name="updateAddress">
     <wsdl:input message="intf:updateAddressRequest"
              name="updateAddressRequest"/>
     <wsdl:output message="intf:updateAddressResponse"
              name="updateAddressResponse"/>
     <wsdl:fault message="intf:updateAddressFaultInfo"
              name="updateAddressFaultInfo"/>
   </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
   <wsdlsoap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http"/>

   <wsdl:operation name="updateAddress">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="updateAddressRequest">
         <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="updateAddressResponse">
         <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="updateAddressFaultInfo">
           <wsdlsoap:fault name="updateAddressFaultInfo" use="literal"/>
      </wsdl:fault>
   </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="AddressServiceService">
      <wsdl:port binding="intf:AddressSoapBinding" name="Address">
         <wsdlsoap:address
            location="http://localhost:8080/axis/services/Address"/>
      </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

So let us begin discussing the various components that make up a WSDL document.

Namespaces

WSDL documents begin with a declarative section that lays out two key components. The first declarative component consists of the various namespace declarations, declared as attributes of the root element (the second is the `types` element discussed in "Types" on page 28):

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://address.samples"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://address.samples"
    xmlns:intf="http://address.samples"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
```

WSDL uses the XML namespaces listed in Table 3 on page 28.

Table 3: WSDL namespaces

Prefix	Namespace URI	Explanation
wsdl	http://schemas.xmlsoap.org/wsdl/	Namespace for WSDL framework.
soap	http://schemas.xmlsoap.org/wsdl/soap/	SOAP binding.
http	http://schemas.xmlsoap.org/wsdl/http/	HTTP binding.
mime	http://schemas.xmlsoap.org/wsdl/mime/	MIME binding.
soapenc	http://schemas.xmlsoap.org/soap/encoding/	Encoding namespace as defined by SOAP 1.1.
soapenv	http://schemas.xmlsoap.org/soap/envelope/	Envelope namespace as defined by SOAP 1.1.
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD.
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD.
tns	(URL to WSDL file)	The this namespace (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD target namespace, which is a different concept.

The first four namespaces are defined by the WSDL specification itself; the next four definitions reference namespaces that are defined in the SOAP and XSD standards. The last one is local to each specification. Note that in our example, we do not use real namespaces; the URIs contain localhost.

Types

The types element encloses data type definitions used by the exchanged messages. WSDL uses XML Schema Definitions (XSDs) as its canonical and built-in type system:

```
<definitions .... >
    <types>
        <xsd:schema .... />(0 or more)
    </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is XML. There is an extensibility element (placeholder for additional XML elements,

that is) that can be used to provide an XML container element to define additional type information in case the XSD type system does not provide sufficient modeling capabilities. In our example, the type definition, shown below, is where we specify that there is a complex type called `AddressBean`, which is composed of two elements, `street` and `zipcode`. We also specify that the type of the `street` element is a `string` and the type of the `zipcode` element is a number (`int`).

```
...
 <wsdl:types>
   <schema targetNamespace="http://address.samples"
           xmlns="http://www.w3.org/2001/XMLSchema">

     <complexType name="AddressBean">
      <sequence>
       <element name="street" type="xsd:string"/>
       <element name="zipcode" type="xsd:int"/>
      </sequence>
     </complexType>

     <element name="AddressBean" type="impl:AddressBean"/>
   </schema>
 </wsdl:types>
...
```

Messages

Messages consist of one or more logical parts. A message represents one interaction between a service requestor and service provider. If an operation is bidirectional (a call returning a result, for example), at least two message definitions are used in order to specify the transmission on the way to and from the service provider:

```
<definitions .... >
    <message name="nmtoken"> (0 or more)
        <part name="nmtoken" element="qname"(0 or 1) type="qname" (0 or 1)/>
            (0 or more)
    </message>
</definitions>
```

The abstract message definitions are used by the operation element. Multiple operations can refer to the same message definition. Operations and messages are modeled separately in order to support flexibility and simplify reuse of existing specifications. For example, two operations with the same parameters can share one abstract message definition. In our example, the messages definition, shown below, is where we specify the different parts that compose each message. The request message `updateAddressRequest` is composed of an `AddressBean` part and an `int` part. The response message `updateAddressResponse` is composed of a `string` part. The fault message `updateAddressFaultInfo` is composed of a `string` part.

```
...
 <wsdl:message name="updateAddressRequest">
   <wsdl:part name="in0" type="intf:AddressBean"/>
   <wsdl:part name="in1" type="xsd:int"/>
 </wsdl:message>
 <wsdl:message name="updateAddressResponse">
   <wsdl:part name="return" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="updateAddressFaultInfo">
   <wsdl:part name="fault" type="xsd:string"/>"
 </wsdl:message>
...
```

Port types

A port type is a named set of abstract operations and the abstract messages involved:

```
<wsdl:definitions .... >
    <wsdl:portType name="nmtoken">
            <wsdl:input name="nmtoken"(0 or 1) message="qname"/> (0 or 1)
            <wsdl:output name="nmtoken"(0 or 1) message="qname"/> (0 or 1)
```

```
            <wsdl:fault name="nmtoken" message="qname"/> (0 or more)
    </wsdl:portType>
</wsdl:definitions>
```

Presence and order of the input, output, and fault messages determine the type of message. For example, for one-way messages the `wsdl:fault` and `wsdl:output` operations would be removed. For a request/response messages, one would include both `wsdl:input` and `wsdl:output` operations. It should be noted that a request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent. For example, the HTTP protocol is a request/response protocol; however, it does not preclude you from sending one-way messages. It simply means that the web service must send an HTTP response back to the client. The response will be consumed by the transport and nothing is propagated back to the client since the response is purely an HTTP response - that is, no SOAP data is associated with the response.

In our example, the port type and operation definition, shown below, are where we specify the port type, called `AddressService`, and a set of operations. In this case, there is only one operation, called `updateAddress`. We also specify the interface that the web service provides to its possible clients, with the input message `updateAddressRequest`, the output message `updateAddressResponse`, and the `updateAddressFaultInfo` that are used in the transaction.

```
...
 <wsdl:portType name="AddressService">
   <wsdl:operation name="updateAddress">
     <wsdl:input message="intf:updateAddressRequest"
              name="updateAddressRequest"/>
     <wsdl:output message="intf:updateAddressResponse"
              name="updateAddressResponse"/>
     <wsdl:fault message="intf:updateAddressFaultInfo"
              name="updateAddressFaultInfo"/>
   </wsdl:operation>
 </wsdl:portType>
...
```

Bindings

A binding contains:

- Protocol-specific general binding data, such as the underlying transport protocol and the communication style for SOAP.
- Protocol extensions for operations and their messages, such as the URN and encoding information for SOAP.

Each binding references one port type; one port type can be used in multiple bindings. All operations defined within the port type must be bound in the binding. The pseudo XSD for the binding looks like this:

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> (0 or more)
        <-- extensibility element (1) --> (0 or more)
        <wsdl:operation name="nmtoken"> (0 or more)
            <-- extensibility element (2) --> (0 or more)
            <wsdl:input name="nmtoken"(0 or 1) > (0 or 1)
                <-- extensibility element (3) -->
            </wsdl:input>
            <wsdl:output name="nmtoken"(0 or 1) > (0 or 1)
                <-- extensibility element (4) --> (0 or more)
            </wsdl:output>
            <wsdl:fault name="nmtoken"> (0 or more)
                <-- extensibility element (5) --> (0 or more)
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

As we have already seen, a port references a binding. The port and binding are modeled as separate entities in order to support flexibility and location transparency. Two ports that merely differ in their network address can share the same protocol binding.

The extensibility elements `<-- extensibility element (x) -->` use XML namespaces in order to incorporate protocol-specific information into the language- and protocol-independent WSDL specification.

In our example, the binding definition, shown below, is where we specify our binding name, `AddressSoapBinding`. The connection is SOAP HTTP, and the style is `document`. We provide a reference to our operation, `updateAddress`; define the input message `updateAddressRequest` and the output message `updateAddressResponse`; and the fault message, `updateAddressFaultInfo`. Additionally, the input and output messages of the operation are defined as literal XML in compliance with the WS-I Basic Profile.

```
...
 <wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
   <wsdlsoap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http"/>

   <wsdl:operation name="updateAddress">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="updateAddressRequest">
         <wsdlsoap:body use="literal"/>
         </wsdl:input>

         <wsdl:output name="updateAddressResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>

         <wsdl:fault name="updateAddressFaultInfo">
            <wsdlsoap:fault name="updateAddressFaultInfo" use="literal"/>
         </wsdl:fault>
   </wsdl:operation>
 </wsdl:binding>
 ...
```

In the above example, both input and output messages are specified. Thus, the operation is governed by the request-response message exchange pattern. If the output message (`wsdl:output` element) was removed, you would have one-way message exchange pattern.

Service definition

A service definition merely bundles a set of ports together under a name, as the following pseudo XSD definition of the service element shows. This pseudo XSD notation is introduced by the WSDL specification:

```
<wsdl:definitions .... >
    <wsdl:service name="nmtoken"> (0 or more)
        <wsdl:port .... /> (0 or more)
    </wsdl:service>
</wsdl:definitions>
```

Multiple service definitions can appear in a single WSDL document.

Port definition

A port definition describes an individual endpoint by specifying a single address for a binding:

```
<wsdl:definitions .... >
    <wsdl:service .... > (0 or more)
        <wsdl:port name="nmtoken" binding="qname"> (0 or more)
        <-- extensibility element (1) -->
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

The binding attribute is of type `QName`, which is a qualified name (equivalent to the one used in SOAP). It refers to a binding. A port contains exactly one network address; all other protocol-specific information is contained in the binding.

Any port in the implementation part must reference exactly one binding in the interface part.

The `<-- extensibility element (1) -->` is a placeholder for additional XML elements that can hold protocol-specific information. This mechanism is required, because WSDL is designed to support multiple runtime protocols. For SOAP, the URL of the service is specified as the SOAP address here.

In our example, the service and port definition, shown below, is where we specify our service, called `AddressServiceService`, that contains a collection of our ports. In this case, there is only one that uses the `AddressSoapBinding` and is called `Address`. In this port, we specify our connection point.

```
...
 <wsdl:service name="AddressServiceService">
      <wsdl:port binding="intf:AddressSoapBinding" name="Address">
         <wsdlsoap:address
            location="http://localhost:8080/axis/services/Address"/>
      </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

REST-based web services

REST defines a set of architectural principles by which you can design web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. REST does not define the technical building blocks of the Web, such as URIs and HTTP, but rather provides guidelines for the development and use of such technologies in a manner designed to provide the necessary scalability and flexibility for a distributed system of global proportions, such as the World Wide Web.

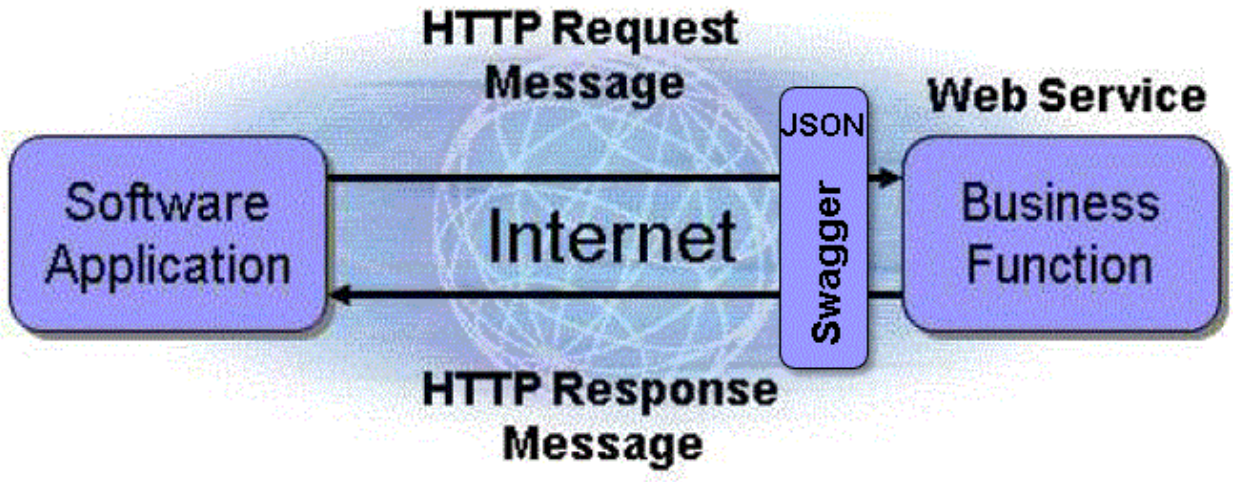


Figure 8: REST-based web services

Core technologies

Several key technologies and standards exist within the web services community:

- HTTP, a communications protocol for the transfer of information on intranets and the World Wide Web. For information on HTTP, see “HTTP protocol” on page 33.
- Uniform Resource Identifier (URI) provide a simple, consistent and persistent means of identifying and locating resources wherever they may exist online. For information about URIs, see “Uniform Resource Identifiers (URIs)” on page 33
- Architectural principles defined by REST. For information on REST, see “REST primer” on page 36.

- XML and/or JSON. For information about XML, see "XML primer" on page 8. For information about JSON, see "JSON primer" on page 34.
- Swagger, a specification for describing RESTful APIs, has become the defacto standard for describing RESTful APIs. For information about Swagger, see "Swagger primer" on page 42.

HTTP protocol

Hypertext Transfer Protocol (HTTP) is a communications protocol for the transfer of information on intranets and the World Wide Web. Its original purpose was to provide a way to publish and retrieve hypertext pages over the Internet.

HTTP development was coordinated by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF), culminating in the publication of a series of Request for Comments (RFCs), most notably RFC 2616 (June 1999), which defines HTTP/1.1, the version of HTTP in common use.

HTTP is a request/response standard between a client and a server. A client is the user and the server is the Web site. The client making an HTTP request using a Web browser, spider, or other user tool is referred to as the user agent. The responding server, which stores or creates resources such as HTML files and images, is called the origin server. In between the user agent and the origin server may be several intermediaries, such as proxies, gateways, and tunnels. HTTP is not constrained to using TCP/IP and its supporting layers, although TCP/IP is the most popular transport mechanism on the Internet. Indeed, HTTP can be implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport. Any protocol that provides such guarantees can be used.

Typically, an HTTP client initiates a request. It establishes a Transmission Control Protocol (TCP) connection to a particular port on a host (port 80 by default). An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server sends back a status line, such as HTTP/1.1 200 OK, and a message of its own, the body of which is perhaps the requested file, an error message, or some other information.

Resources to be accessed by HTTP are identified using Uniform Resource Identifiers (URIs) (or, more specifically, Uniform Resource Locators (URLs)) using the http or https URI schemes.

For more information about the HTTP standard, go to the following URL:

```
http://www.ietf.org/rfc/rfc2616.txt
```

Uniform Resource Identifiers (URIs)

Universal Resource Identifiers (URIs) are, without question, one of the single most important characteristics of web-based applications. URIs provide a simple, consistent and persistent means of identifying and locating resources wherever they may exist online.

An example of an URI is as follows:

```
http://www.ibm.com/systems/power/software/i/iws/index.html
```

According to the URI standard, the example is a URI and has several component parts:

- A scheme name (`http`)
- A domain name (`www.ibm.com`)
- A path (`/systems/power/software/i/iws/index.html`)

For more information about the URI standard, go to the following URL:

```
http://www.ietf.org/rfc/rfc3986.txt
```

JSON primer

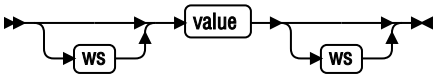
JavaScript Object Notation (JSON) is an open standard format for data interchange. Although originally used in the JavaScript scripting language, JSON is now language-independent, with many parsers available in many languages.

Compared to XML, JSON has many advantages. Most predominantly, JSON is more suited to data interchange. XML is an extremely verbose language: every element in the tree has a name, and the element must be enclosed in a matching pair of tags. Alternatively, JSON expresses trees in a nested array format similar to JavaScript. This enables the same data to be transferred in a far smaller data package with JSON than with XML. This lightweight data package lends itself to better performance when parsing.

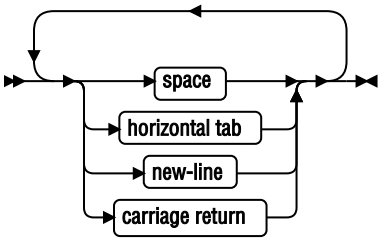
JSON can be seen as both human and machine-readable. JSON is an easy language for humans to read, and for machines to parse.

According to the standard, the JSON syntax is made up of a sequence of tokens. The tokens consist of six structural characters, strings, numbers, and three literal names. The tokens are logically organized into data, objects and arrays. Figure 1 on page 34 shows the syntax diagram for JSON text.

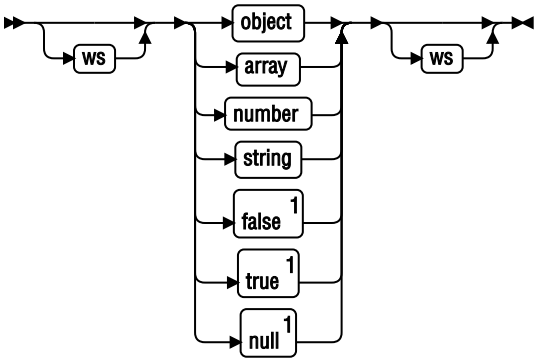
JSON text



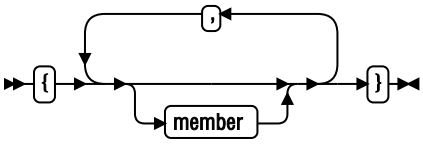
ws



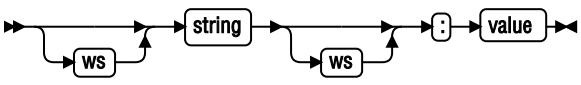
value



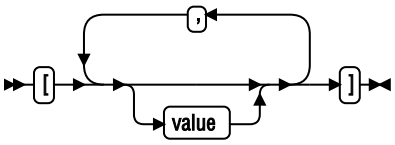
object



member



array



Notes:

[1] The actual literal name: `false`, `true`, or `null`. These values must be lowercase.

Figure 9: JSON text

The following sections provides basic information on JSON. More information about JSON may be found at:

```
http://www.rfc-editor.org/rfc/rfc7159.txt
```

JSON objects

The primary concept in JSON is the object, which is an unordered collection of name/value pairs, where the value can be any JSON value. JSON objects can be nested, but are not commonly deeply nested.

JSON objects begin with a left brace ({) and ends with a right brace (}). Name/value pairs in the object are separated by a comma (,). The name and value for a pair is separated by colon (:). The name is a string (see "JSON strings" on page 36 for more details).

The value may be a JSON object, a JSON array (see "JSON arrays" on page 35 for more details), or one of the four atomic types shown in Table 4 on page 35:

Table 4: JSON data types

Data type	**Example**
string	`"someStringValue"`

number	`3`
	`6.2`
	`-122.026020`
	`9.3e5`

boolean	`true` `false`
the special "null" value	`null`

The following example shows a simple JSON object:

```
{
 "isbn": "123-456-222",
 "title": "The Ultimate Database Study Guide",
 "abstract": "What you always wanted to know about databases",
 "price": 28.00
}
```

JSON arrays

A JSON array is an ordered collections of values. Arrays begin with a left bracket ([) and ends with a right bracket (]). Values in the array are separated by a comma (,).

The following is a simple example of a JSON object that contains arrays:

```
{
 "category": ["Non-Fiction", "Technology"],
 "ratings": [10, 5, 32, 78, 112]
}
```

JSON strings

Strings begins and ends with a quotation mark ("). Within the quotation marks any character may be used except for characters that must be escaped: quotation mark, backslash (\), and the control characters. Any character may be escaped. In addition, characters between Unicode hexadecimal values 0000 through FFFF may be represented by a six character sequence: backslash, followed by lowercase letter u, followed by four hexadecimal digits that encode the character's code point.

The following shows examples of JSON strings:

```
 "category"

 "15\u00f8C"
```

JSON numbers

JSON numbers are represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. A fraction part is a decimal point followed by one or more digits. An exponent part begins with the letter E in upper or lower case, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

The following shows examples of JSON numbers:

```
 123

 -122.026020

 9.3e5
```

REST primer

REST was first introduced in 2000 by Roy Fielding at the University of California, Irvine, in his academic dissertation, "*Architectural Styles and the Design of Network-based Software Architectures*[3]," which analyzes a set of software architecture principles that use the Web as a platform for distributed computing.

The dissertation suggests that in its purest form, a concrete implementation of a REST web service follows four basic design principles:

- Expose directory structure-like URIs.
- Use HTTP methods explicitly.
- Be stateless.
- Transfer XML, JavaScript Object Notation (JSON), or both.

The following sections expand on these four principles. For more information about the REST, read Chapter 5 of Roy Fielding's dissertation, located at the following URL:

```
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
```

[3] The dissertation can be found at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Expose directory structure-like URIs

The notion of using URIs to identify resources is central to REST style web services; by virtue of having a URI, resources are part of the Web. From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST web service is going to be and whether the service is going to be used in ways that the designers can anticipate.

REST web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes. For example, in a discussion threading service that gathers topics ranging from RPG to paper, you might define a structured set of URIs like this:

```
http://www.myservice.org/discussion/topics/{topic}
```

The root, `/discussion`, has a `/topics` node beneath it. Underneath that there are a series of topic names, such as gossip, technology, and so on, each of which points to a discussion thread. Within this structure, it's easy to pull up discussion threads just by typing something after `/topics/`.

In some cases, the path to a resource lends itself especially well to a directory-like structure. Take resources organized by date, for instance, which are a very good match for using a hierarchical syntax. This example is intuitive because it is based on rules:

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little silly to explain it that way, but this is the level of simplicity we're after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:

```
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}
```

Some additional guidelines to make note of while thinking about URI structure for a RESTful web service are:

- Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs.
- Keep everything lowercase.
- Substitute spaces with hyphens or underscores (one or the other).
- Avoid query strings as much as you can.
- Learn from popular APIs (Google, Facebook, Twitter, and so on.)

URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It's also important that the relationship between resources that's encoded in the URIs remains independent of the way the relationships are represented where they are stored.

Designing the URIs for a REST style web service requires special care, as they may be referenced by large numbers of applications, documents, or bookmarks for many years and thus have to be designed so that they are stable.

Use HTTP methods explicitly

One of the key characteristics of a RESTful web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a web server, or to execute a query with the expectation that the web server will look for and respond with a set of matching resources.

REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

An unfortunate design flaw inherent in many web APIs is in the use of HTTP methods for unintended purposes. The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. At least this is how the HTTP/1.1 RFC describes GET. But there are many cases of unattractive web APIs that use HTTP GET to trigger something transactional on the server - for instance, to add records to a database. In these cases the GET request URI is not used properly or at least not used RESTfully. If the web API uses GET to invoke remote procedures, it looks like this:

```
GET /adduser?name=Robert HTTP/1.1
```

It's not a very attractive design because the web method above supports a state-changing operation over HTTP GET. Put another way, the HTTP GET request above has side effects. If successfully processed, the result of the request is to add a new user - in this example, Robert -to the underlying data store. The problem here is mainly semantic. web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI and return these or a representation in a response, not to add a record to a database. From the standpoint of the intended use of the protocol method then, and from the standpoint of HTTP/1.1-compliant web servers, using GET in this way is inconsistent.

Beyond the semantics, the other problem with GET is that to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way, it invites web caching tools (crawlers) and search engines to make server-side changes unintentionally simply by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into the HTTP request payload (e.g. XML). The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity as follows:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

The method above is exemplary of a RESTful request: proper use of HTTP POST and inclusion of the payload in the body of the request. On the receiving end, the request may be processed by adding the resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of `/users`. This containment relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent and defines the new entity's URI in the POST request.

A client application may then get a representation of the resource using the new URI, noting that at least logically the resource is located under `/users` as follows:

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Using GET in this way is explicit because GET is for data retrieval only. GET is an operation that should be free of side effects, a property also known as idempotence.

A similar refactoring of a web method also needs to be applied in cases where an update operation is supported over HTTP GET, as shown below.

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

This changes the name attribute (or property) of the resource. While the query string can be used for such an operation, and Listing 4 is a simple one, this query-string-as-method-signature pattern tends to break down when used for more complex operations. Because your goal is to make explicit use of HTTP methods, a more RESTful approach is to send an HTTP PUT request to update the resource, instead of HTTP GET, for the same reasons stated previously:

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

Using PUT to replace the original resource provides a much cleaner interface that's consistent with REST's principles and with the definition of HTTP methods. The PUT request is explicit in the sense that it points at the resource to be updated by identifying it in the request URI and in the sense that it transfers a new representation of the resource from client to server in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names and values on the request URI. The PUT request in the example also has the effect of renaming the resource from Robert to Bob, and in doing so changes its URI to `/users/Bob`. In a REST web service, subsequent requests for the resource using the old URI would generate a standard 404 Not Found error.

As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by using nouns in URIs instead of verbs. In a RESTful web service, the verbs - POST, GET, PUT, and DELETE - are already defined by the protocol. And ideally, to keep the interface generalized and to allow clients to be explicit about the operations they invoke, the web service should not define more verbs or remote procedures, such as `/adduser` or `/updateuser`. This general design principle also applies to the body of an HTTP request, which is intended to be used to transfer resource state, not to carry the name of a remote method or remote procedure to be invoked.

Stateless

REST web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a web service call. Using intermediary servers to improve scale requires REST web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Figure 10 on page 40 illustrates a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a `previousPage` variable somewhere to be able to respond to requests for next.

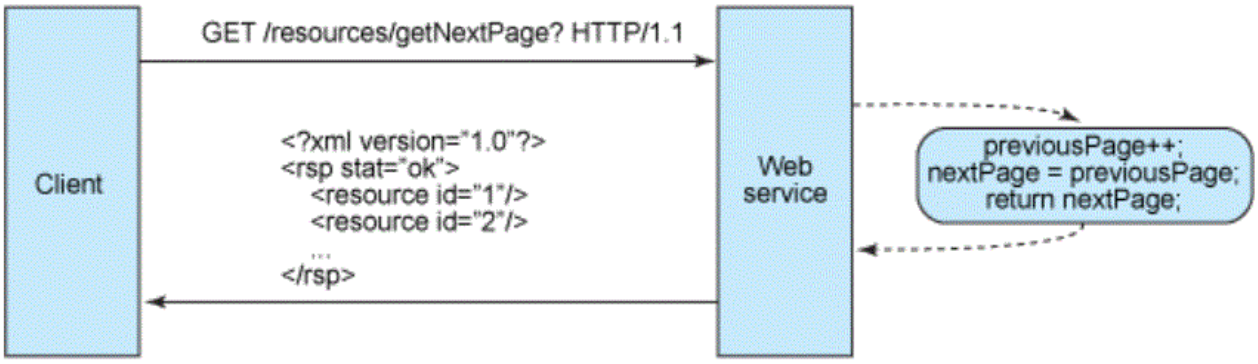


Figure 10: Stateful design

Stateful services like this get complicated. Stateful services may require a lot of up-front consideration to efficiently store and enable the synchronization of session data. Session synchronization adds overhead, which may impact server performance.

Stateless server-side components, on the other hand, are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful web service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for next (see Figure 11 on page 40).

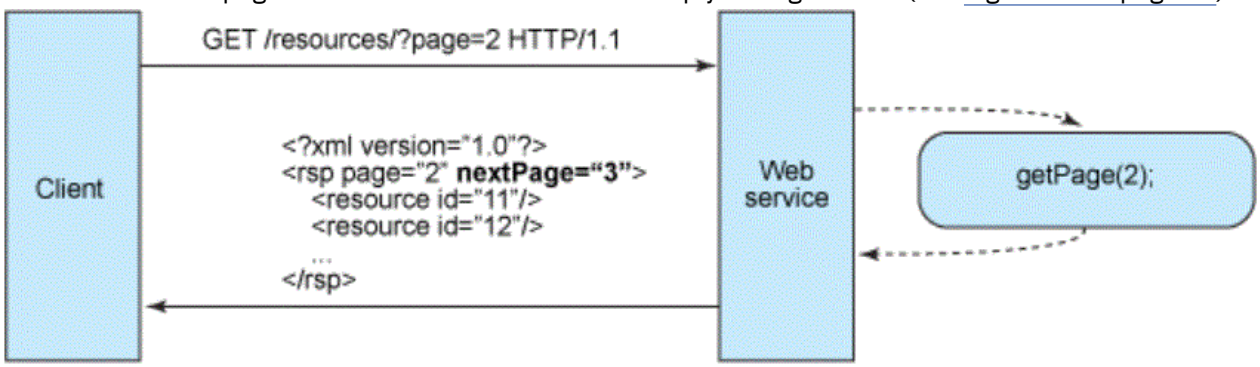


Figure 11: Stateless design

A stateless web service generates a response that links to the next page number in the set and lets the client do what it needs to in order to keep this value around. This aspect of RESTful web service design can be broken down into two sets of responsibilities as a high-level separation that clarifies just how a stateless service can be maintained:

- **Server**
  - Generates responses that include links to other resources to allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources so that these remain connected.
  - Generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a `Cache-Control` and `Last-Modified` (a date value) HTTP response header.
- **Client application**

- Uses the `Cache-Control` response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the `Last-Modified` response header and sends back the date value in an `If-Modified-Since` header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand in hand in that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes.
- Sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the web service interface and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server, the server's ability to add context to a request, or about application state that is kept in between requests.

This collaboration between client application and service is essential to being stateless in a RESTful web service. It improves performance by saving bandwidth and minimizing server-side application state.

REST style web service payloads

A resource representation typically reflects the current state of a resource, and its attributes, at the time a client application requests it. Resource representations in this sense are mere snapshots in time. This could be a thing as simple as a representation of a record in a database that consists of a mapping between column names and XML tags, where the element values in the XML contain the row values. Or, if the system has a data model, then according to this definition a resource representation is a snapshot of the attributes of one of the things in your system's data model. These are the things you want your REST Web service to serve up.

The last set of constraints that goes into a RESTful Web service design has to do with the format of the data that the application and service exchange in the request/response payload or in the HTTP body. This is where it really pays to keep things simple, human-readable, and connected.

The objects in your data model are usually related in some way, and the relationships between data model objects (resources) should be reflected in the way they are represented for transfer to a client application. In the discussion threading service, an example of connected resource representations might include a root discussion topic and its attributes, and embed links to the responses given to that topic.

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

And last, to give client applications the ability to request a specific content type that's best suited for them, construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a MIME type. Some common MIME types used by RESTful services are shown in Table 5 on page 41.

Table 5: Common MIME types used by RESTful services

MIME-type	**Content-type**
JSON	application/json
XML	application/xml

This allows the service to be used by a variety of clients written in different languages running on different platforms and devices. Using MIME types and the HTTP Accept header is a mechanism known as content negotiation, which lets clients choose which data format is right for them and minimizes data coupling between the service and the applications that use it.

Swagger primer

Swagger is an open specification for defining REST APIs. A Swagger document is the REST API equivalent of a WSDL document for a SOAP-based web service. The Swagger document specifies the list of resources that are available in the REST API and the operations that can be called on those resources. The Swagger document also specifies the list of parameters to an operation, including the name and type of the parameters, whether the parameters are required or optional, and information about acceptable values for those parameters. Additionally, the Swagger document can include JSON Schema that describes the structure of the request body that is sent to an operation in a REST API, and the JSON schema describes the structure of any response bodies that are returned from an operation.

The integrated web services server supports version 2.0 of the Swagger specification. Information on Swagger and the version 2.0 of the Swagger specification may be found at the following URLs:

```
http://swagger.io/

https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md
```

Chapter 3. Leading practices for web services

This section discusses leading practices to incorporate into the development cycle for web services.

This chapter introduces approaches that you should consider in order to deliver quality of service for your web services development, design, and architecture. It contains leading practices for leveraging your web services components.

Web services design best practices

This section discusses basic considerations for designing a web services solution. It includes high-level guidelines that apply to any development effort, and then discusses technology selection options.

Basics of web services planning

The first step is to perform the basics of design planning. Understand what you have in place today, what your goals are, and what technology you want to use in order to position your applications for future growth. The following list provides a high-level view of the planning tasks that you should perform:

1. Review the standards used in the development and design phase.

   Web services can be based on a variety of programming models. Start by identifying how your existing web services are designed as well as the APIs, standards, and specifications that were part of the design.
2. Identify your goals.

   Consider what you want to accomplish by using web services. Identify applications and business logic that you want to make available as a service. Consider which existing services you want to migrate to newer technology.
3. Determine how web services fit into your current topology, applications, and programming model.

   Determine how your current web services process requests on the server and how the clients manage and use the web service. Keep these factors in mind when planning for new or migrated web services.
4. Design your web services for non-functional requirements to fit your business solution.

   In other words, design your web services for reliability, availability, manageability, and security. For example, you may want your web services to process a transaction in a reasonable amount of time at all hours of the day and provide users with optimal security, such as authentication mechanism.

   Have a dialogue with your security organization on what business functions should be exposed over the Internet and what precautions should be taken to protect them.

Choosing between SOAP and REST web services

This section offers guidance in helping architects make informed decisions about the architectural style or styles to be used when designing an application. It is important to note that REST-style and SOAP-style web services are not mutually exclusive architectural styles. There may be many circumstances in which an application will want to take advantage of both styles.

The advantages of SOAP-style web services include:

- Provides a standard for exchanging data in XML format, for example, the parameters used in a program call (for the inbound message) and the data resulting from the call (for the outbound message).
- Are independent of protocol, platform, operating system, and programming language.
- Are flexible and extensible.
- Enables the use of web services quality of service extensions such as WS-Security and WS-ReliableMessaging.

The advantages of REST-style web services include:

- They are prescriptive in terms of implementation patterns and security options, leading to a uniform approach that is intuitive for consumers and providers alike.
- The barrier of entry for mobile application programmers is set low; JavaScript application programmers can handle HTTP connections and JSON data without requiring additional specialist libraries (for example, for parsing)
- REST interfaces for server assets are familiar to mobile application programmers, and can be consumed in the same way as industry-standard APIs.
- They are independent of platform, operating system, and programming language.
- They are flexible and extensible.
- JSON can often represent data more concisely than XML. Every element in the tree has a name, and the element must be enclosed in a matching pair of tags. JSON expresses trees in a nested array format similar to JavaScript. This way can enable the same data to be expressed in a relatively smaller data package than with XML, which can be a factor for mobile applications.

REST-style web services is a good option:

- When URLs for end users are needed that you can send in mail or embed in web sites
- Client or proxy caching of resource representations is used
- Direct access to the service/resource representation from a web browser is needed
- The ability to manipulate the same resource interactively (e.g. using HTML forms) and programmatically is needed

SOAP-style web services is a good option in following cases:

- Support clients using SOAP protocol
- Multi-protocol exchanges (e.g. HTTP to MQ)
- Application-level security, long running transactions, and other sophisticated service policies

The choice of REST-style versus SOAP-style web services is nothing more than a choice over a design strategy based on business and application need. But it is a choice that can profoundly impact how an application is used and evolves over time. Where both SOAP-style and REST-style APIs are offered, REST APIs are more widely used due to the fact that REST APIs are often easier to consume, especially with scripting languages, and browser-based experimentation is easy.

It should be noted that external web API protocol usage overwhelming REST-based. However, many web APIs implement SOAP and it will continue to be an important building block for enterprise service integration. One of the roles of web APIs and REST in such cases is complementary to the existing services, to allow services to be exposed to external consumers in a manner more suitable for consumption.

Leading practices for developing web services

This section discusses leading practices to incorporate into the development cycle for web services.

Common best practices

Basic common practices that you must always consider when developing your web services are:

- Use simple data types.

  Even though web services were designed with interoperability in mind, it is a good practice to use simple data types where possible. By simple, we mean integers and strings. Compound types and arrays of simple types are also considered simple data types. Anything that does not fall into this pattern should be used carefully.
- Avoid nillable primitives.

Nillable primitive types (indicating that an element can be null) are allowed for web services, but there are interoperability issues when using them. The best advice is not use them at all, and use dedicated flags to control the condition that a value does not exist.

- Use short attribute, property, and tag names.

  Because each attribute, property, and tag name is transmitted verbatim, the length of a message is directly dependent on the length on the attribute and property names. The general guideline is that the shorter the attribute, property, and tag names are, the shorter the transmitted message and the faster the communication and processing.

- Avoid deep nesting of XML structures or JSON objects.

  Because parsing of deeply nested XML structures or JSON objects increases the processing time, deeply nested compound data types should be avoided. This also increases comprehension time of the data type itself.

- Choose the appropriate data format for your environment.

  Both XML and JSON are appropriate formats for web services. However, typical use cases for XML and JSON are as follows:

  – XML is suitable for data exchange or sharing between independent entities, systems, or applications, particularly where the domain is regulated.

  – JSON is suitable for use for data exchange or sharing within an application, and is typically used with human interfaces and mobile applications.

- Use web services caching as provided by the platform if possible.

  A caching framework allows for caching of information at various levels, thus save processing time.

- Don’t send data that is not needed.

  Sending data that is not needed still requires the client application to process the data in order to get at the more meaningful data in the response.

SOAP-based web services best practices

SOAP-style web services that use quality of service extensions to the base SOAP-style web services standards may need to avoid fine-grained web services that require you to issue multiple SOAP requests before you can complete a task.

Web services use a simple, but powerful format to exchange data using the SOAP protocol: XML. While reading and structuring XML documents with a simple text editor eases the use of SOAP, the process of automatically creating and interpreting XML documents is more complex. Without careful design, you can end up in a situation where the complexity of dealing with the SOAP protocol has a higher performance cost than performing the actual computation.

Design course-grained web services that perform more complex business logic. This allows the web service to return more data in response to a single request, rather than having multiple requests to retrieve smaller portions of data. Working with coarser grained services also allows a single service to be reused, instead of creating multiple fine-grained services.

REST-based web services best practices

Basic common practices that you must always consider when developing REST-based web services are:

- Follow REST principles.

  REST-style web services (i.e. web APIs) is a fast growing business channel. If you want applications developers to use an API it must be simple to use, access and understand. If an API does not follow REST principles or does not work in a consistent fashion, developers will move on to another API that does.

- URI design matters.

URIs should be predictable and consistent. By considering issues with URI patterns early in the application design, the RESTful service increases its usability and value over an extended time.

Part 2. Integrated web services server concepts

This part of the book introduces integrated web services server concepts and architecture, including installation details.

Chapter 4. Integrated web services server overview

In support of web services and SOA, the IBM i operating system integrates a software technology that support the externalization of an integrated language environment (ILE) program object as a web service. The technology is the integrated web services server. This integration opens the IBM i system to a variety of web service client implementations, including RPG, COBOL, C, C++, Java, .NET, PHP, WebSphere® Process Server, Enterprise Service Bus, mobile, and Web 2.0.

The features of the integrated web services server include:

- Easy to use via centralized configuration and control. The web services server focuses on making the deployment of ILE-based web services as painless as possible by hiding the complexities of the web services server behind an-easy to use and intuitive web administrative GUI frontend that allows you to manage and monitor the server and any deployed web services.
- Leading edge technologies. Even though the focus is on ease-of-use for the deployment of ILE-based services, the web services server is built on the powerful, yet lightweight, integrated application server and best-of-breed technologies in support of web services.
- Small footprint. The integrated web services server uses ILE programming architecture for minimal consumption of IBM i resources.

This chapter will give an overview of integrated web services server, including what specifications and standards are currently supported, the server architecture, and the programming model.

Supported specifications and standards

The integrated web services server has the following capabilities:

- Supports web services based on SOAP. The following standards are supported:
  - WSDL 1.1
  - SOAP 1.1 and SOAP 1.2
  - The request-response and one-way message exchange patterns are the only supported message exchange patterns.

  The integrated web services server does not support any SOAP quality of service extensions, such as WS-Security and WS-Policy.
- Supports web services based on REST principles.

Server architecture

The web services server is simply an integrated web application server [4] that is running a web services engine.

The integrated web application server addresses the need for a minimal footprint, easy to configure, secure infrastructure for hosting web applications and web services. The current version of the server is based on the IBM WebSphere Liberty profile, a highly composable, dynamic application server.

Figure 12 on page 50 illustrates the underlying architecture of the integrated web services server.

[4] More information on the server can be found at URL http://www.ibm.com/systems/power/software/i/iwas/.

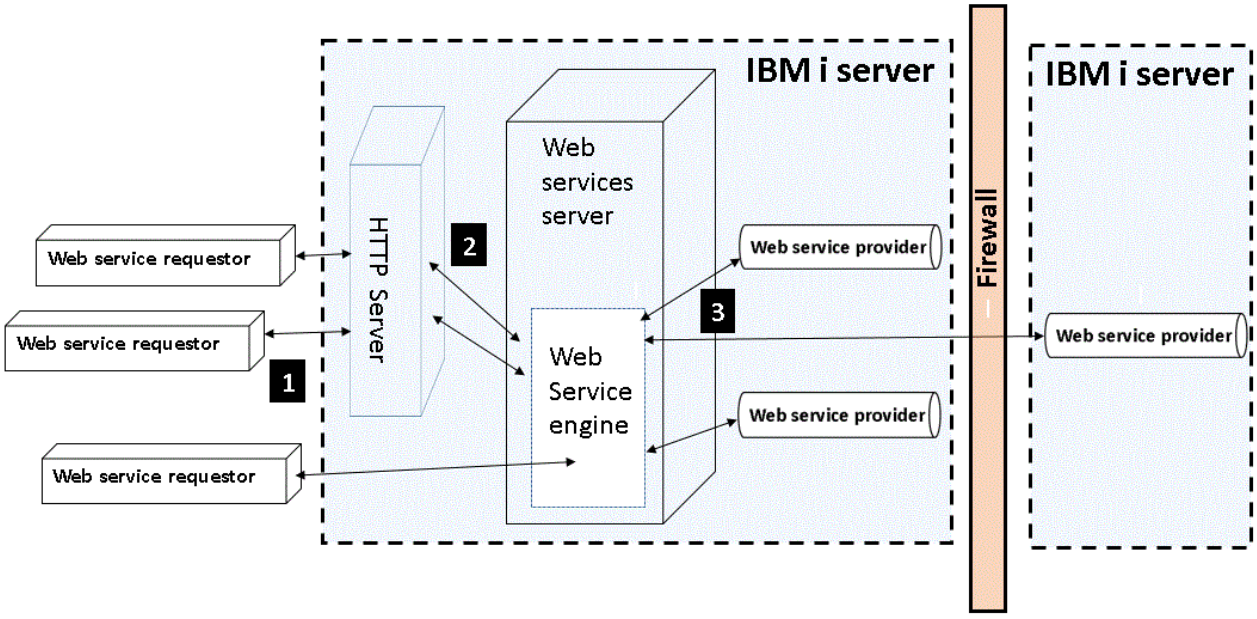


Figure 12: Flow between web service requestors and web service providers

The flow is as follows:

1. A client application invokes a web service. The invocation is via a URL and the request may either go through the HTTP server or directly to the integrated web services server.
2. Requests going to the HTTP server are forwarded to the integrated web services server plug-in[5], which sends the request on to the integrated web services server.
3. The web services engine running in the integrated web services server maps the request to a deployed web service and passes the data to the web service. Note that the web service implementation code (i.e. ILE program or service program) may reside on the same system as the integrated web services server or on a remote system. This is discussed in detail in "Two-tier web services" on page 50. Once the web service completes by returning output parameters, the web services engine serializes the data into the proper format and sends the response back to the client.

One might ask whether the HTTP server is needed? Well, it depends. Prior to the IBM Web Administration GUI interface adding support to enable SSL in the integrated web services server, the only way to enable SSL was to do it by using the HTTP server. You would enable SSL within the HTTP server and have clients go through the HTTP server. You no longer need to use the HTTP server if you want to enable SSL. However, if you wanted to enable HTTP basic authentication, or do URL mapping in order to simplify URLs, or if you do not want to directly expose the integrated web services server to the external world, you will need the HTTP server.

Two-tier web services

You have the ability to separate the integrated web services server and the web service implementation code. You can have the server on one IBM i system and the web service implementation code on another system. The primary use case would be an environment where the integrated web services server is in front of a firewall and the web service implementation code is not able to be run in the demilitarized zone (i.e. in front of the firewall) as shown in Figure 12 on page 50.

The steps to do this is as follows:

1. Deploy the web service.

[5] The plug-in is the glue between the web server and the integrated web services server, and its primary responsibility is to forward HTTP requests to the integrated web services server.

2. Specify the host of the web service implementation code for the web service. You may also indicate that the connection between the web services server and the web service implementation code should be secure. Go to web service properties, and within the **Connection Pool** tab, specify the host of the web service implementation code for the web service. You may also indicate that the connection between the web services server and the web service implementation code should be secure. You also are able to set the host by using the "setWebServiceProperties.sh command" on page 119 or the "installWebService.sh command" on page 110. The following example sets the host for web service `QIWSSAMPLEr`, and appending `:SECURE` to the host indicates that a secure connection should be used between the web services server and the remote system hosting the web service implementation code:

```
setWebServiceProperties.sh -server WSREMOTE -service QIWSSAMPLEr
       -host lp59ut29:SECURE
```

3. On the system hosting the web service implementation code, configure the Remote Command Server and the Signon Server applications to use SSL as documented in the support page http://www.ibm.com/support/docview.wss?uid=nas8N1010449.
4. Save the server using "saveWebServicesServer.sh command" on page 118.
5. Restore the server on the server that will not contain the web service implementation code using the "restoreWebServicesServer.sh command" on page 116.
6. If using a secure connection between the web services server and the web service implementation code, go to the **Server Properties->Properties** panel and set the default key store for the web services server that was restored, in addition to importing the certificate to the system store. The specified key store must be the path to the SYSTEM key store, `/QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB`. You may also use the QShell command "setWebServicesServerProperties.sh command" on page 122 to do this. Here is an example:

```
setWebServicesServerProperties.sh  -server WSREMOTE
       -defaultKeystore /QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB
       -defaultKeystorePassword xxxxxxx
```

   Note that the user profile used to run the web services server must have *RX (read, execute) authority to all parts of the SYSTEM key store path. In addition, the server user profile must have *USE authority to the user profile the web service is running under on the remote system.

Server programming model

Let us take a closer look at how the web services engine invokes web services. Figure 13 on page 52 shows what actually happens when a web service request is received by the web services engine.

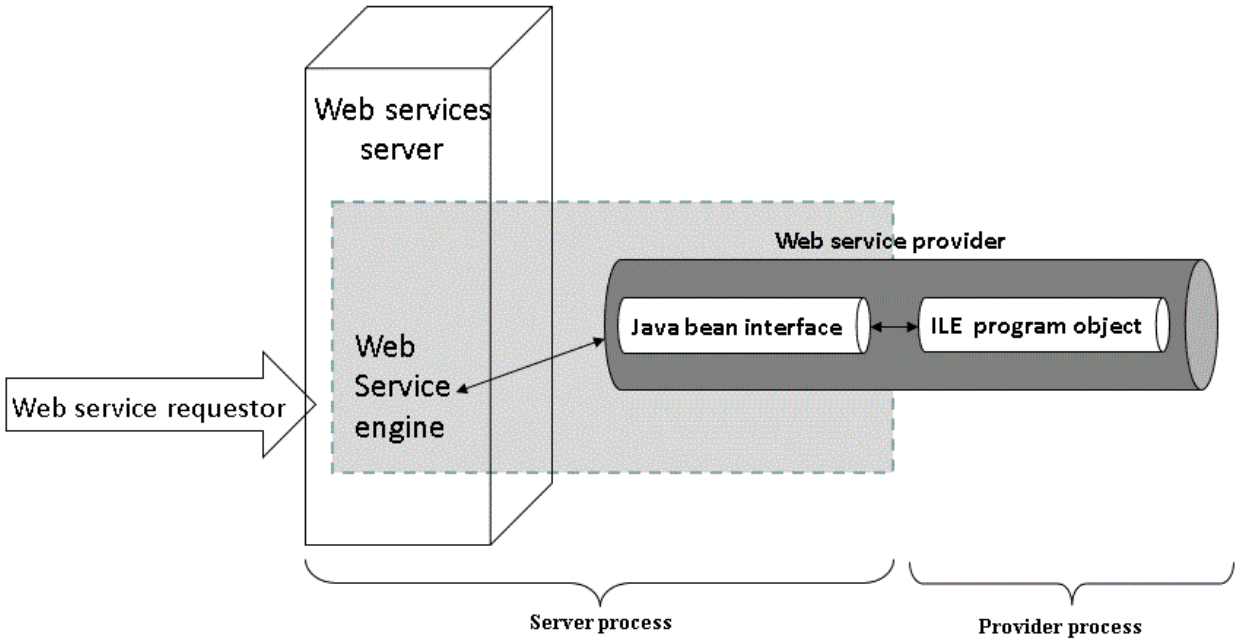


Figure 13: Web service engine invoking web service

When you deploy an ILE program or service program as a web service, underneath the covers a Java bean interface is created that acts as a proxy for the ILE program object that is being deployed. As requests come in, the web services engine invokes the Java bean, which in turn invokes the program object (i.e. ILE program or procedure in ILE service program) via the IBM Toolbox for Java classes. The Java proxy runs in the same process as the web services server, while the web service implementation program object that actually handles the web service request runs in a remote command host server job (QZRCSRVS). Simultaneous requests to the same web service will result in multiple host server jobs. The Java proxy contains a connection pool of host server jobs so that host server jobs can be reused.

The Java proxy is generated during deployment time. Deployment can be done using either the Web Administration for i GUI or integrated web services server QShell scripts. During deployment, a user will specify the program object that will be deployed as a web service, the interfaces to export as a web service operation, and several properties, such as host server job connection pool properties, and the user profile and library list that the ILE web service will run under. From there the necessary artifacts will be generated and deployed to the server instance, transparent to the user. The user will have the ability to query the list of deployed web services that have been deployed on an integrated web services server instance and manage the web services.

For a complete look at the GUI interfaces, see Chapter 6, “Administration console,” on page 57. Information on the integrated web services server QShell scripts can be found in Chapter 7, “Command line tools,” on page 105.

There is one important detail that has not yet been discussed, and that is how is the web service Java proxy code generated from a program or service program? That is where PCML comes into the picture.

Program Call Markup Language (PCML)

Program Call Markup Language (PCML) is a tag language that is based on XML and that enables the calling of program objects with less Java code.

Ordinarily, extra code is needed to connect, retrieve, and translate data between an ILE program objects and IBM Toolbox for Java objects. However, by using PCML, calls to the program object with the IBM Toolbox for Java classes are automatically handled. PCML class objects are generated from the PCML tags and help minimize the amount of code needed to be written in order to call program objects from an application (in this case, the web service proxy Java code).

During the deployment of a web service, the integrated web services server deployment code retrieves the PCML data associated[6] with the program object and uses the PCML data to generate web service proxy Java code with the proper data types.

More information on PCML can be found in the IBM Knowledge Center.

[6] The module(s) of the selected program object whose interfaces are to be externalized must have been compiled so PCML information is generated and stored as part of the module. If no PCML data is found in the program object, a path to a PCML file would need to be supplied.

Chapter 5. Integrated web services server installation details

This chapter describes the integrated web services server package, including what you need to do to install integrated web services server the package and a description of the various components that make up the integrated web services server package.

Installing server support code

The integrated web services server support code is included in option 3 (Extended Base Directory Support) of the base operating system (e.g. 5770SS1 for i 7.1, etc.). In addition to installing base option 3 of the operating system, the following prerequisite products will also need to be installed:

- Qshell - base option 30 of operating system
- PASE - base option 33 of operating system
- Host Servers - base option 12 of operating system
- Digital Certificate Manager - base option 34 of operating system
- IBM HTTP Server for IBM i
- IBM Java SE 7 32 bit must be loaded for Qshell script support. In addition, it is a good idea to also load Java SE 8 64 bit so servers that are created use the latest supported runtime. This will ensure minimal disruption since Java 7 is scheduled to go out of service sometime in 2019.

Note: After installing the various license product options, you should load the latest HTTP Group PTF since all fixes and enhancements are packaged as part of the HTTP Group PTF. It would also be wise to load the latest Java group PTF. The various group PTFs for an IBM i release may be found at the IBM Support Portal.

Server product package

The installation directory for integrated web services server support code is `/QIBM/ProdData/OS/WebServices`. In this chapter, and throughout this documentation, the installation directory is shown as `<install_dir>`.

When the package has been installed, the installation directory (`<install_dir>`) contains the following directory structure shown in Figure 14 on page 56:

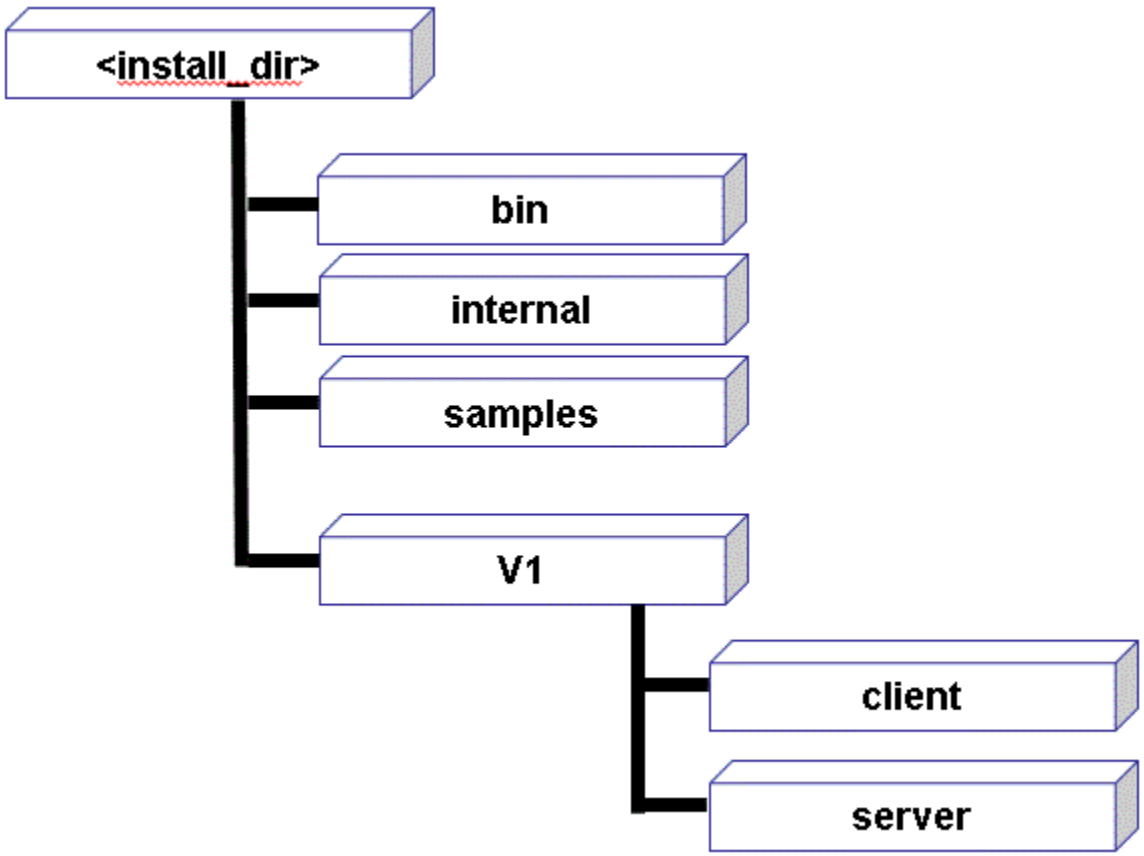


Figure 14: Install directory structure

The following table gives an overview of the contents of each directory:

Table 6: Contents of installed directories	
Installed directory	**Contents**
`<install_dir>/bin`	Contains the QShell scripts in support of the integrated web services server. The scripts are described in Chapter 7, "Command line tools," on page 105.
`<install_dir>/internal`	Contains internal code in support of IBM Web Administration GUI. In addition, the directory contains the web service engine in support of integrated web services server versions 1.3 and 1.5.
`<install_dir>/samples`	Contains source code for the samples that accompany integrated web services server.
`<install_dir>/V1`	Contains the integrated web server for ILE client directory (`<install_dir>/V1/client`). The directory also includes a server directory `<install_dir>/V1/server` that is used internally in support of the integrated web services server versions 1.3 and 1.5.

Updating integrated web services server support code

To update the server support code, ensure you load the latest HTTP group PTF for the operating system release installed on the IBM i system.

Chapter 6. Administration console

The integrate web services server is managed through the IBM Web Administration for i interface. The Web Administration for i interface is an application that is loaded in the HTTP Administration server, and accessed from a web browser, that is typically accessed using the following URL:

```
http://<server>:2001/HTTPAdmin
```

Note: If you are unable to connect to the server, the Web Administration server may not be started. You can start the server using the following CL command:

```
QSYS/STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
```

The Web Administration for i interface combines forms, tools, and wizards to create a simplified environment to set up and manage many different servers on your system. The wizards guide you through a series of advanced steps to accomplish a task. With a few clicks of a button, you can have an integrated web services server running in no time at all.

This chapter will give a detailed description of the features of the Web Administration for i interface that pertain to the integrated web services support. It is assumed that you are generally familiar with the Web Administration for i interface. If not, please read about the interface under the HTTP Server topic in the IBM Knowledge Center.

Attention: If you do not have latest HTTP Group PTF you may see Web Administration for i interface panels or fields on a panel that are not available to you. You can either choose to load the latest HTTP Group PTF (assuming the missing feature is available for your release) or ignore the panel or missing field.

User profile requirements to use the Web Administration for i interface

By default, only users with *ALLOBJ and *IOSYSCFG special authorities can manage and create web-related servers on the system through the use of the IBM Web Administration for i interface. web-related servers include instances of IBM HTTP Server and integrated web services server. A user without the necessary IBM i special authorities to manage or create web-related servers requires an administrator to grant that user permission to a server or group of servers.

To be able to access the Web Administration for i interface, the IBM i user profile used to sign on must meet at least one of the following conditions:

- The user profile has *ALLOBJ and *IOSYSCFG special authorities.
- The user profile has been granted permission to an entire class of servers, or a specific server.
- The user profile has been granted permission to create servers.

For example, if a user wants to create an integrated web services server using the Web Administration for i interface, the user profile must either have *ALLOBJ and *IOSYSCFG special authorities, or have permission to create integrated web services servers.

Only users with *ALLOBJ and *IOSYSCFG special authority are allowed to grant, revoke, or manage user permissions. The granting of permissions to a user profile is done through the Web Administration for i interface by giving user profiles that need to access the Web Administration for i interface roles to specific servers or a class of servers.

Note: Granting *ALLOBJ authority to a user profile or using the QSECOFR user profile to access the Web Administration for i interface is not recommended.

Roles

Roles define a set of permissions that define what operations a user is allowed to perform on a server. The Web Administration for i interface defines the following roles:

Administrator
Any IBM i user profile with *ALLOBJ and *IOSYSCFG special authority is identified with the role of Administrator. An Administrator has unrestricted use of every feature in the Web Administration for i interface, including the ability to manage user permissions. An Administrator cannot be assigned any other role.

Note: A user profile cannot be assigned this role.

Developer
Is allowed to view and modify a server, including the ability to delete a server.

Operator
Is allowed to view a server, including the capability to start and stop a server. In addition, an Operator is allowed to modify trace settings for a server.

If a user with a role of Developer or Operator has no role assigned to them for a server, they are not allowed to view the server or any of its attributes.

Permissions

A *permission* is the ability to perform an operation on a server. The ability for a user to perform operations on a server is determined by the role they have been assigned for the server. The Web Administration for i roles are defined with the following permissions (only permissions relating to the integrated web services server are shown):

Table 7: Permissions corresponding to each role..

	Roles		
Permissions	**Administrator**	**Developer**	**Operator**
Start/Stop server	x	x	x
Delete server	x	x	
Install/Remove web services	x	x	
Start/Stop web services	x	x	x
Modify server attributes	x	x	
Modify web service attributes	x	x	
Modify server tracing	x	x	x
Use Web Performance Advisor	x	x	
Use Web Log Monitor	x	x	
Create server[Note 1]	x		

Notes®:

1. An administrator granting permissions to a user profile needs to explicitly grant the create-server permission.

Only an Administrator can grant permissions. The granting of permissions to a user profile is done through the Web Administration for i interface by giving user profiles that need to access the Web Administration for i interface roles to specific servers or a class of servers.

Note: If a user creates a server, they are automatically assigned the role of Developer to the newly created server.

Permissions can be granted to a specific server or to all servers of a certain type. When granting permissions, you should be aware of the following points:

- If you grant a user permission to create a web services server, then you must also grant the user permission to create HTTP Servers. This is due to the association between an HTTP Server and the web services server.
- If you grant a user permissions to a web services server, and you do not explicitly grant the user permissions to the associated HTTP Server(s), the user is automatically granted the same permissions to the associated HTTP Servers(s). This is also true in reverse. If you grant a user permissions to an HTTP Server, and you do not explicitly grant the user permissions to the associated web services server, the user is automatically granted the same permissions to the associated web services server.

  Note: A warning message is displayed on the Web Administration for i interface when permissions are implicitly granted to a user.
- If you attempt to grant a user different permissions to an HTTP Server and the associated web services server, the user is granted the higher permission and both servers get assigned that permission.

  Note: A warning message is displayed on the Web Administration for i interface when permissions to servers are upgraded.

If a user has no permissions to any servers, and no permission to create any type of server, then the user is not allowed to access the Web Administration for i interface.

Creating an integrated web services server

The Create Web Services Server wizard provides a convenient way to create a web services server that allows for the externalization of programs running on IBM i, such as RPG or COBOL ILE programs, as web services. The goal of the wizard is to create a recommended production level server configuration while requiring minimal information from the user. Alternatively, you can create an integrated web services server by using the QShell script `createWebServicesServer.sh` (see “createWebServicesServer.sh command” on page 106 for more information). The main difference between the two is that the QShell script gives you the opportunity to specify the directory in which the server is to be created (the wizard creates all servers in the `/www` directory).

The following sections will walk you through the steps necessary to create an integrated web services server using the Create Web Services Server wizard. Prior to following the steps, sign on to the Web Administration for i interface by specifying the following URL: `http://<hostname>:2001/HTTPAdmin`, where hostname is the host name of your server (note that if SSL has been configured for the Web administration server the URL would be `https://<hostname>:2010/HTTPAdmin`) and sign on. You must have `*ALLOBJ` and `*IOSYSCFG` special authorities to create a web services server, or you must have been given permission to create web services servers as described in “User profile requirements to use the Web Administration for i interface” on page 57. If a connection to the server fails, execute the following CL command to ensure the server is started:

```
QSYS/STRTCPSVR *HTTP HTTPSVR(*ADMIN)
```

If you continue to experience issues accessing the Web Administration for i console, please open a Service Request (PMR) or call 1-800-IBM-SERV.

Step 1. Launch the Create Web Services Server wizard

Launch the Create Web Services Server wizard by either clicking on the link in the navigation bar under the **Common Tasks and Wizards** heading, or on the main page of the **Setup** tab (see Figure 15 on page 60).

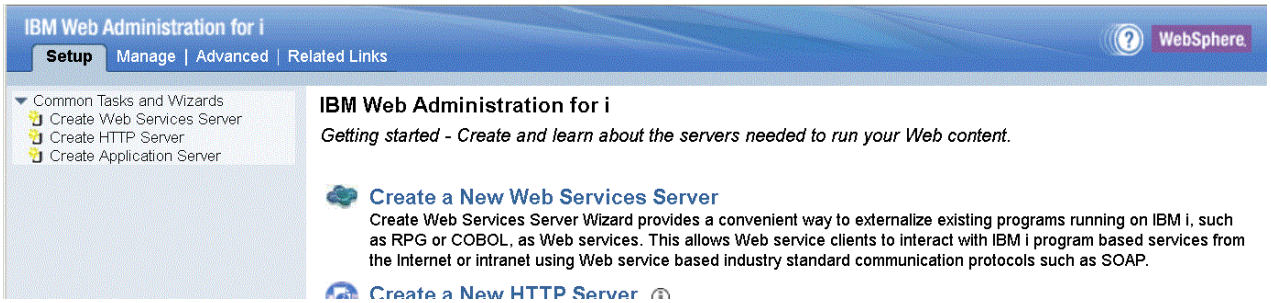


Figure 15: Link to Create Web Services Server wizard

Step 2. Specify web services server name

When you bring up the Create Web Services Server wizard, the first panel you see is a form with a default server name and description. You have the option of naming (see Figure 16 on page 60) the web services server that is to be created and provide a short description if you so choose.

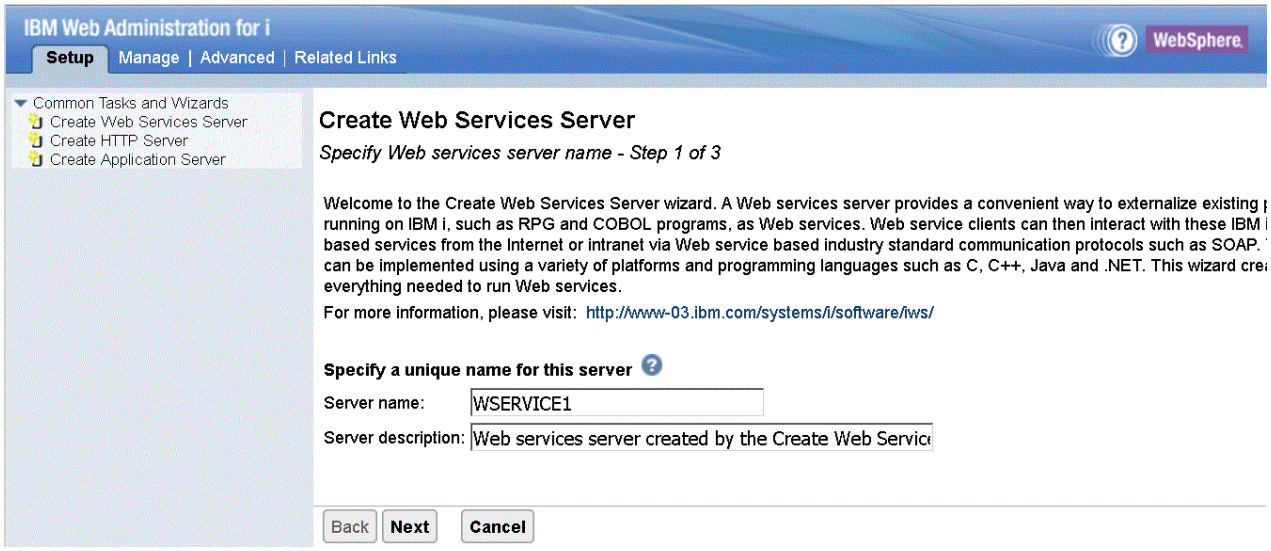


Figure 16: Specify web services server name

Click on the **Next** button at the bottom of the form.

Step 3. Specify web services server user ID

The next form that is shown allows you to specify the user ID to run the jobs associated with the server (see Figure 17 on page 61). You have the option of specifying an existing user ID, creating a new user ID, or using the default user ID (the default user ID is QWSERVICE).

Note: Any user ID specified for the server must be enabled and the password set to a value other than *NONE. Ensure this is true for the specified user ID.

Figure 17: Specify web services server user ID

Click on the **Next** button at the bottom of the form.

Step 4. Confirm creation of web services server

The wizard shows you a summary page (see Figure 18 on page 61), giving you the chance to see the details relating to the web services server before it actually kicks off the task of creating the web services server.

Figure 18: Summary – Servers tab

Let us examine the summary panel in a little more detail. The **Servers** tab on the summary page gives you details about the web services server and the associated HTTP server (anytime a web services server is created a corresponding HTTP server is also created). Here you will find information about ports, the location of the web services server instance, etc.

If you click on the **Services** tab, you will see information about the services being deployed (see Figure 19 on page 62). Here you will find the sample services being deployed.

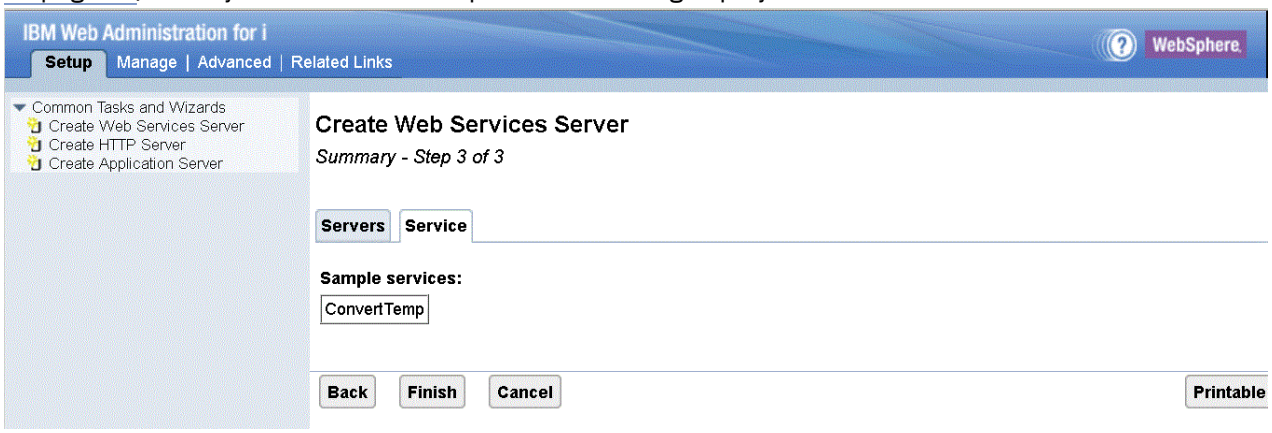


Figure 19: Summary – Services tab

Clicking on the **Finish** button at the bottom of the summary page will result in the panel shown in Figure 20 on page 62 being displayed by the wizard:

Figure 20: Server creating

After the server is created, the wizard will start the web services server and associated HTTP server. After a short time, you will see the server in **Running** state and the deployed[7] services active (green dot to the left of service name) as in Figure 21 on page 63:

[7] A sample web service, `ConvertTemp`, gets deployed in all newly created web services servers. This web service converts temperatures from Fahrenheit to Celsius.

Figure 21: Server created

Server directory structure

When you create an integrated web services server, the directory structure shown in Figure 22 on page 64 is created:

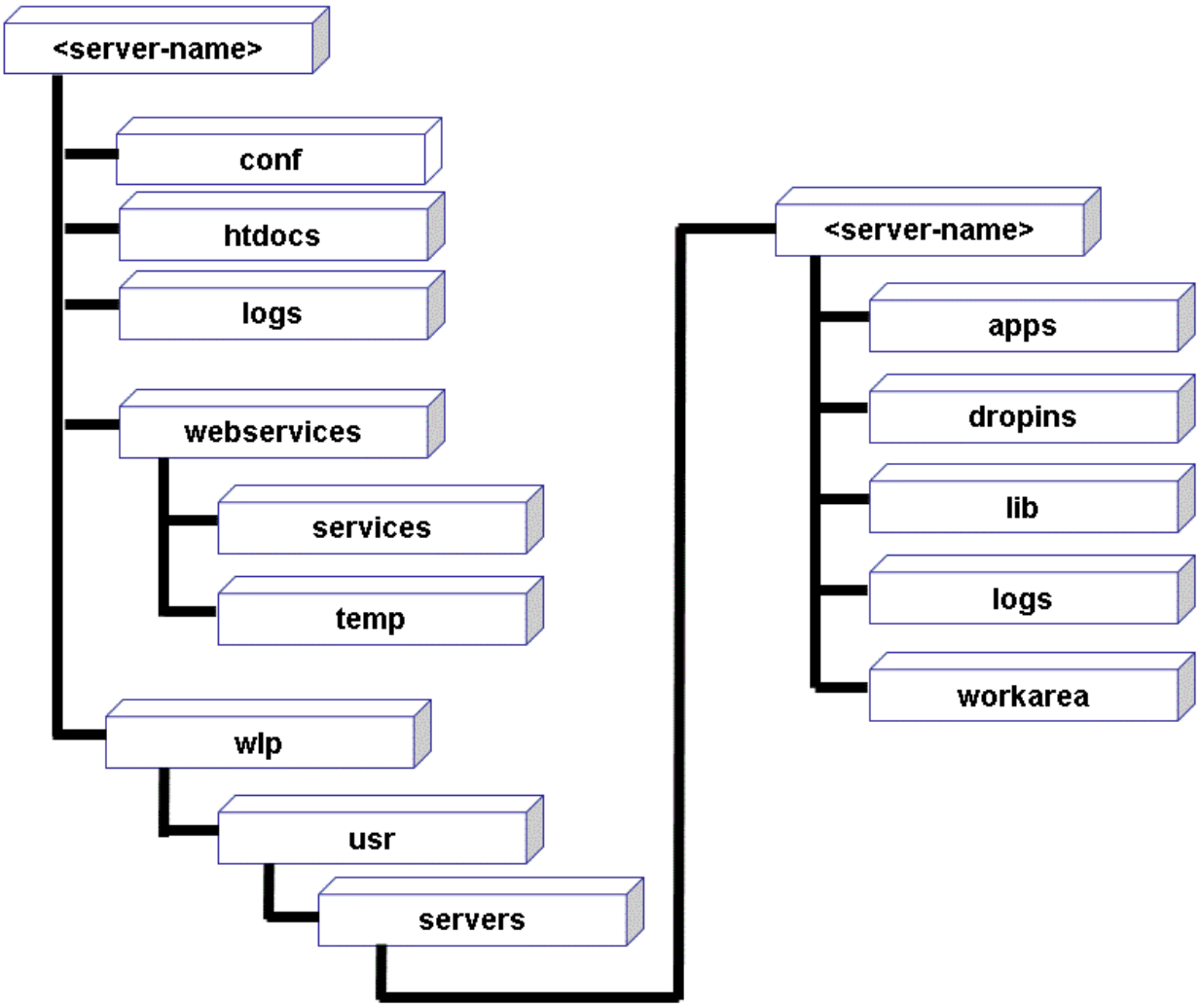


Figure 22: Directory structure of created server

The directories can be categorized into three groups. The first group is the directories that contain the HTTP server objects assuming that an HTTP server is associated with the integrated web services server. These objects include everything in the `conf`, `htdocs`, and `logs` directories. The second group are those directories that contain objects relating to deployed web services, which are stored within the `webservices` directory. The third group are objects relating to the application server instance, and includes everything in the `wlp` directory.

Note: You should never manually modify any objects in the integrated web server instance directory.

Server runtime environment default port numbers

When you create an integrated web services server, the server "reserves"[8] ten ports for its use, although in actuality the server currently uses four ports:

- Integrated web services server HTTP port
- Integrated web services server administration port
- Integrated web services server JMX port
- Apache web server HTTP port

[8] On creation, a port block of 10 ports is assumed to be available for the server, although only four ports are used by default.

The HTTP ports may be configured by using the Web Administration for i interface or the `setWebServicesServerProperties.sh` command. The JMX port is an ephemeral port that cannot be configured. The administration port may be changed using the `setWebServicesServerProperties.sh` command.

Exploring the Web Administration for i interface

The Web Administration for i interface provides you the capability to start, stop, create and delete integrated web services servers. In addition, you have the ability to change server properties. Figure 23 on page 65 shows the panel that is shown when managing an integrated web services server.

Figure 23: Manage server panel

If you are not in the proper context the navigation frame will not contain web services-related navigation links. To get into the proper context, you need to have selected the **Application Servers** subtab and then selected the server you want to manage. The web services-related navigation links include:

1. Web service wizards
2. Server properties
3. Web services
4. Problem determination

Web service wizards

The Web Administration for i interface includes wizards, which are graphical user interfaces for setting up the integrated web services server. The arrow shown Figure 24 on page 66 in points to the available wizards.

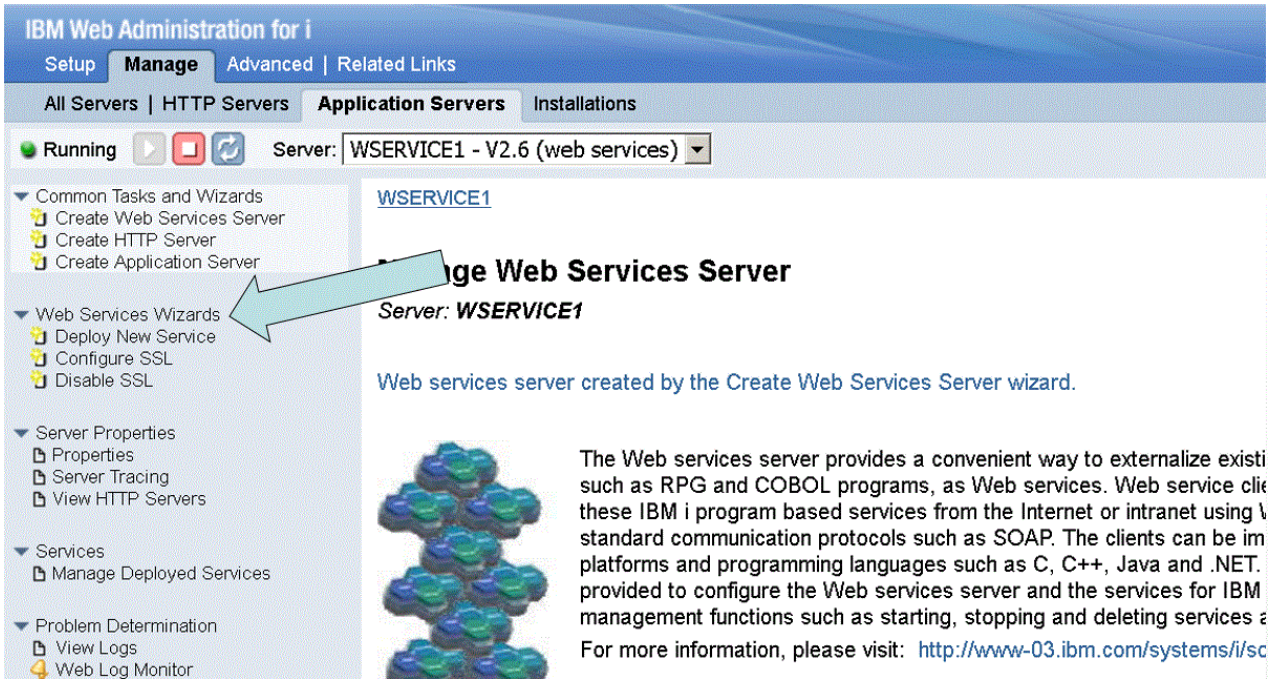


Figure 24: Manage server panel - wizards

The following wizards are available:

- Deploy New Service
- Configure SSL
- Disable SSL

The deploy new service wizard

The **Deploy New Service** wizard enables you to externalize programs running on IBM i, such as RPG or COBOL ILE programs, as web services.

You need to ensure you are in the proper context by selecting the web services server that will contain the web service. If you are not in the proper context the navigation frame will not contain web services-related navigation links. To get into the proper context, you need to have selected the **Application Servers** subtab and then selected the web services server that will contain the web service as shown in Figure 25 on page 67.

Figure 25: Setting proper context

Once you have set the proper context, you can begin deploying a web service by launching the **Deploy New Service** wizard by clicking on the link in the navigation bar under the **Web Services Wizards**.

The following sections will walk you through the panels you will see when deploying a web service to an integrated web services server. Because you can deploy web services based on SOAP or REST, some panels will be designated as SOAP-ONLY or REST-ONLY.

Panel: Specify Web service type

When you launch the wizard, you should see the panel shown in Figure 26 on page 67.

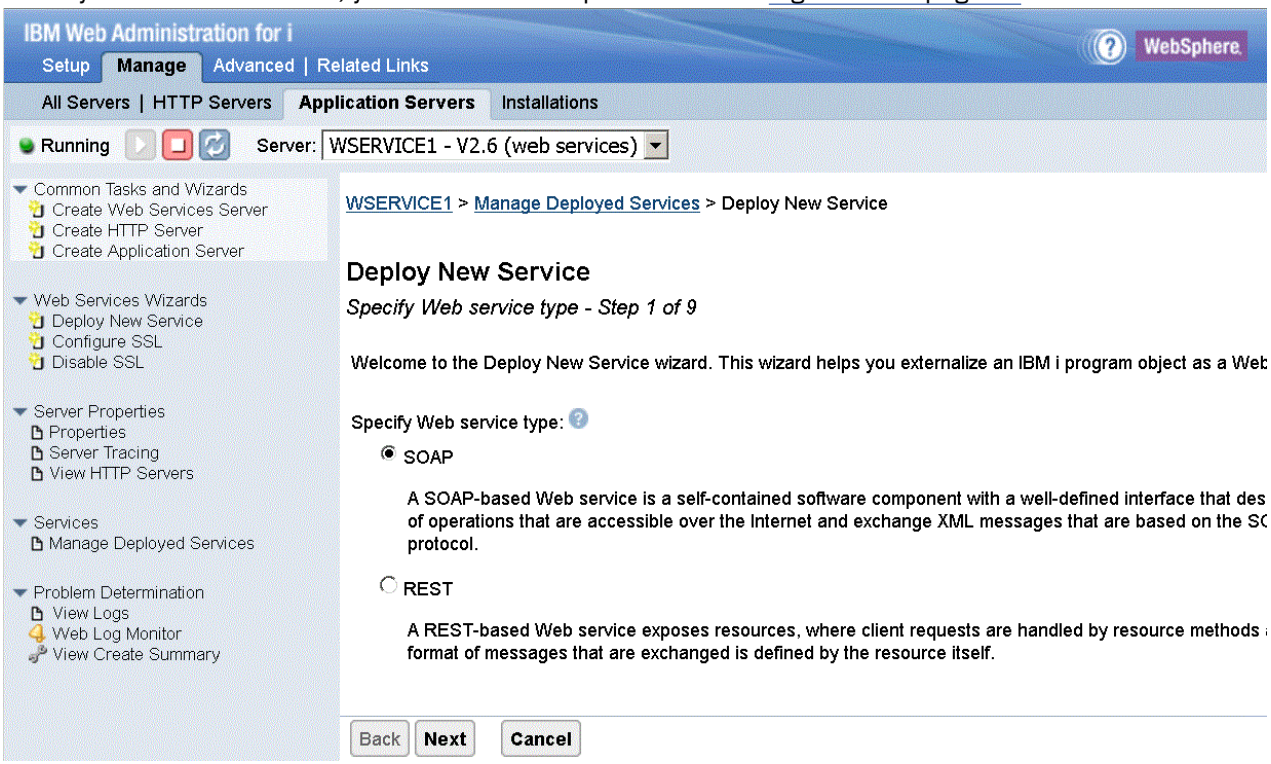


Figure 26: Specify Web service type

Click on the **Next** button at the bottom of the form.

Panel: Specify location of IBM i program object

We now need to specify (see Figure 27 on page 68) an ILE program object name from which the web service is generated.

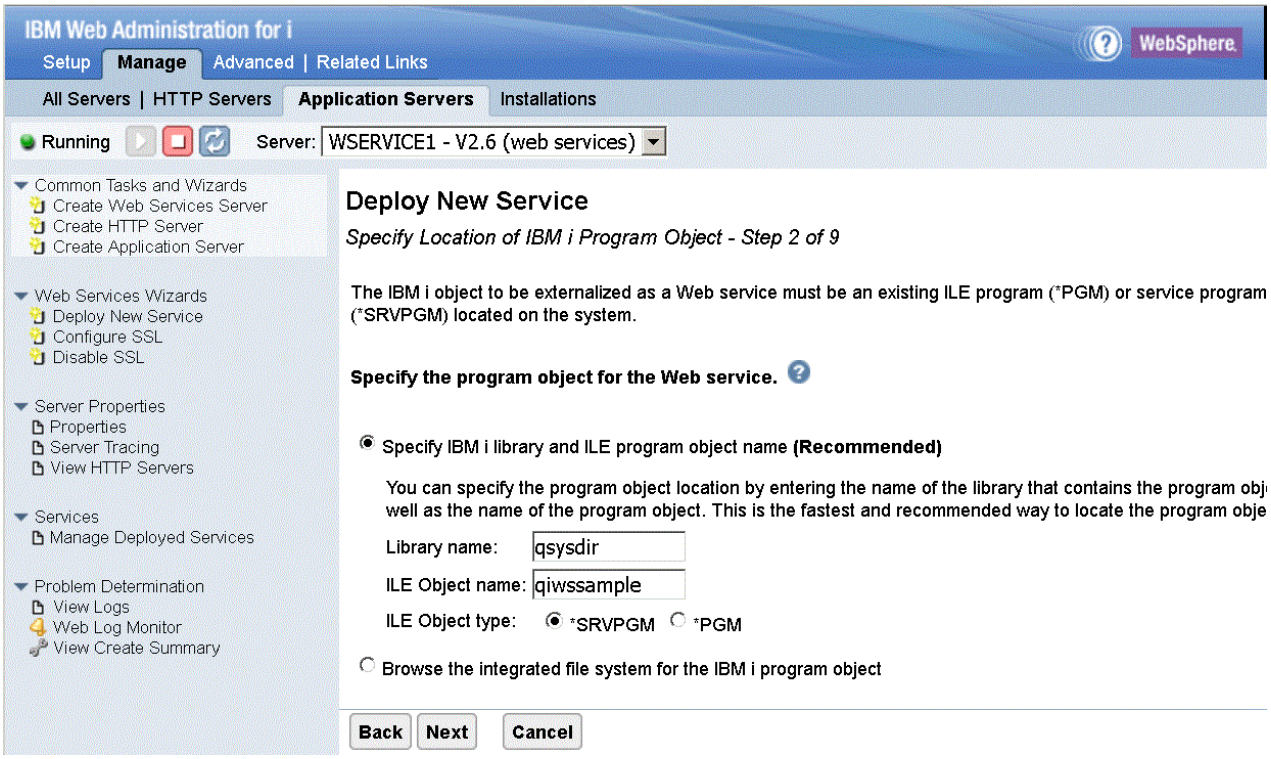


Figure 27: Specify location of IBM i program object

There are two ways to locate the program object on the system. The default way is to specify the program and library names by selecting the option **Specify IBM i library and ILE object name (Recommended)**. Another way is to search for the program object by browsing the integrated file system (IFS), which could take a while if a directory is specified that contains a lot of objects, such as `/QSYS.LIB`. You will need to use this method to specify an IFS path to a program object residing on an independent auxiliary storage pool (ASP). For example, if an independent ASP was mounted on directory `/XSM`, and the program `MYIASPPGM` resides on the independent ASP in library `MYIASPLIB`, then you would need to specify the following path: `/XSM/QSYS.LIB/MYIASPLIB.LIB/MYIASPPGM.PGM`.

For each program object to be deployed there should be a corresponding PCML document. As was discussed previously in this document, PCML is based upon XML, and is a tag syntax that is used to fully describe the input and output parameters of the program object.

The wizard determines whether the program object has any PCML data stored in the object. If PCML data is not found, the wizard shows a prompt in which you can specify a path to a PCML document that describes the program object. The PCML document is then validated and processed so that only information for exported functions or procedures is shown.

The PCML information can be generated two ways:

1. Manually, using an editor.
2. Recompiling ILE modules so that the PCML information is stored as part of the module (`*MODULE`) object.

The generation of PCML information is done by using the Program Interface Information (`PGMINFO`) parameter on the `CRTBNDRPG`, `CRTRPGMOD`, `CRTBNDCBL` and `CRTCBLMOD` commands. Since PCML is generated on a per-module basis, the wizard combines the generated PCML documents into one

document so that the resultant document has the following format (not all elements and attributes are shown):

```
<pcml version="4.0">
   <program name="p1" ... >  ...   </program>
   <program name="p2" ... >  ...   </program>
 </pcml>
```

The recommended approach for users not experienced with PCML, or users who do not need to customize the PCML document, is to store the PCML as part of the module object as follows:

- For i 6.1 and higher, one is able to use the PGMINFO parameter as follows:

```
RPG:   CRTRPGMOD PGMINFO(*PCML *MODULE)
       CRTBNDRPG PGMINFO(*PCML *MODULE)
COBOL: CRTCBLMOD PGMINFO(*PCML *MODULE)
       CRTBNDCBL PGMINFO(*PCML *MODULE)
```

- For all releases, the following line must be in the source code to automatically generate PCML:

```
RPG:   H PGMINFO(*PCML:*MODULE)
COBOL: PROCESS OPTIONS PGMINFO(PCML MODULE)
```

In this example, we are deploying the service program QIWSSAMPLE in library QSYSDIR. This service program is shipped as part of integrated web services server support and contains one exported procedure, CONVERTTEMP, that is written in the ILE RPG programming language. The procedure converts the temperature from Fahrenheit to Celsius. The source for the procedure can be found in:

```
/QIBM/ProdData/OS/WebServices/samples/server/ConvertTemp/CNVRTTMP.RPGLE
```

Click on the **Next** button at the bottom of the form.

Panel: Specify name for the web service

Now we need to give the web service a meaningful service name and description. By default, the service name and description is set to the name of the selected program object (see Figure 28 on page 70). It would make sense to set the name of the web service to ConvertTemp, but that name is already being used for the sample service that is deployed on all newly created integrated web services servers. So let us set the name to ConvertTemp2. You can also change the description if you so choose.

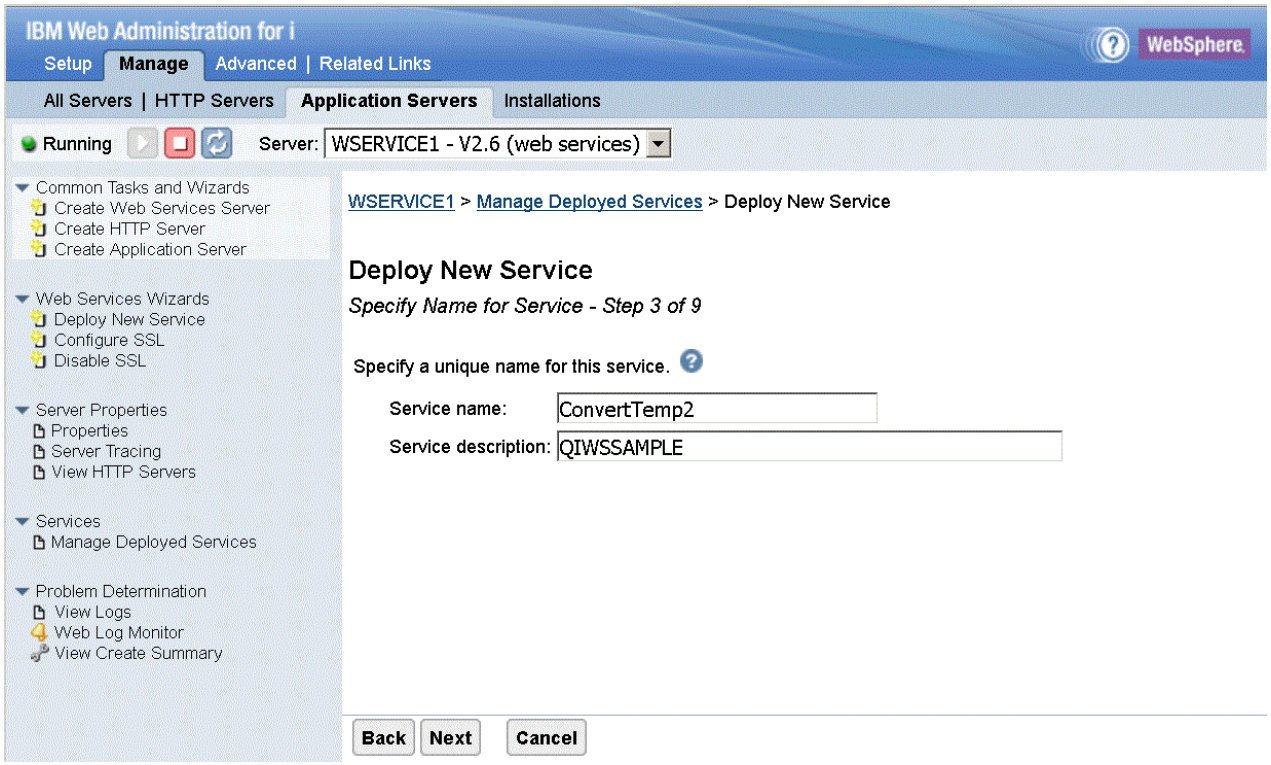


Figure 28: Specify name for the web service (SOAP-ONLY panel)

If deploying the program object as a REST web service, there will be an additional field that will be displayed as shown in Figure 29 on page 70.

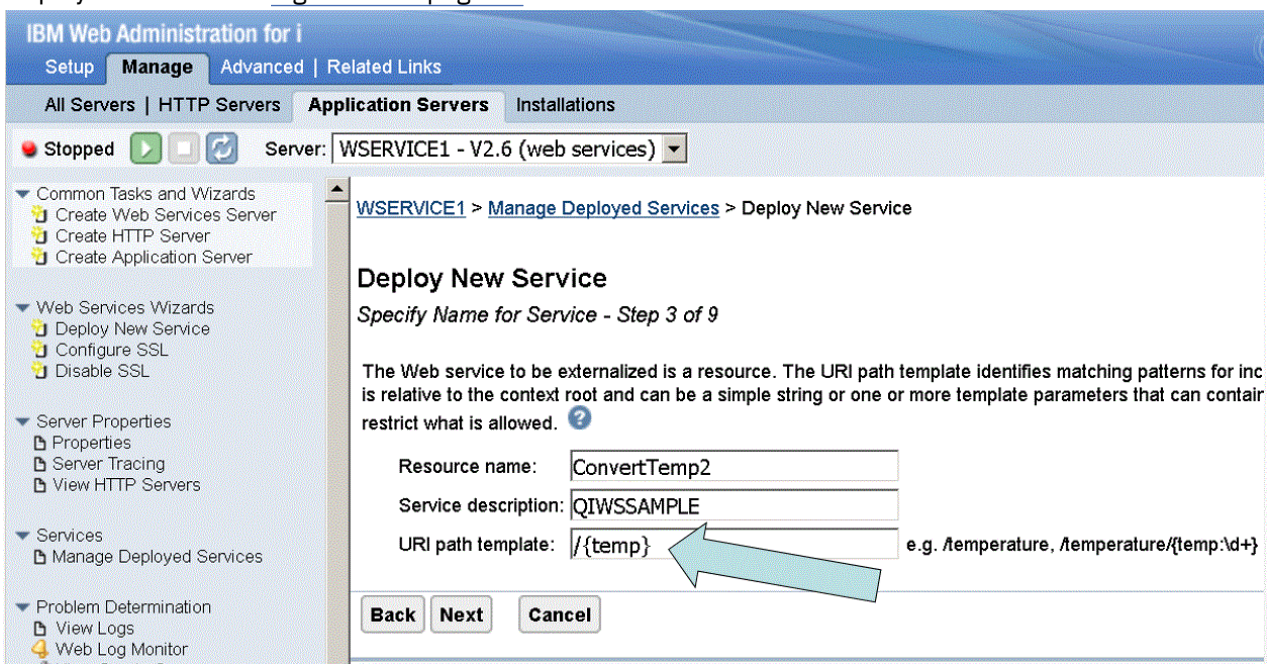


Figure 29: Specify name for the web service (REST-ONLY panel)

The *URI path template* is a partial URI that further qualifies how requests are mapped to resources and resource methods. The path is relative to the context root and can be a simple string or one or more template parameters that can contain regular expressions to further restrict what is allowed. Template parameters are denoted by braces ({ and }) and must start with a forward slash (/). The format of a parameter with a regular expression is as follows: `{var [: regExpr] }`. The default value is forward

slash (/). If you use a path parameter, you must specify the parameter when you reference the resource, otherwise the resource is not found and an error is returned. Here are some examples:

```
/temperature
/temperature/{temp}
/temperature/{temp : \d+}
```

In the example above, `{temp}` is a template parameter that can be referenced and passed to a procedure parameter. The `\d+` is a regular expression that limits the template parameter `{temp}` to one or more digits. So this would be a valid REST request: `http://host:port/web/services/ConvertTemp2/34`. This would be an invalid URL: `http://host:port/web/services/ConvertTemp2`.

Note: If you have problems referencing a deployed RESTful web service and a regular expression is used in the URI template, ensure that the regular expression is correct. More often than not that will be the source of the problem.

Click on the **Next** button at the bottom of the form.

Panel: Select export procedures to externalize as a web service

The wizard will show a list of exported procedures as shown in Figure 30 on page 71. For service programs (object type of *SRVPGM), there may be one or more procedures. For programs (object type of *PGM), there is only one procedure, which is the main entry point to the program. Expanding the procedure row shows the parameters for the procedure and various parameter attributes.

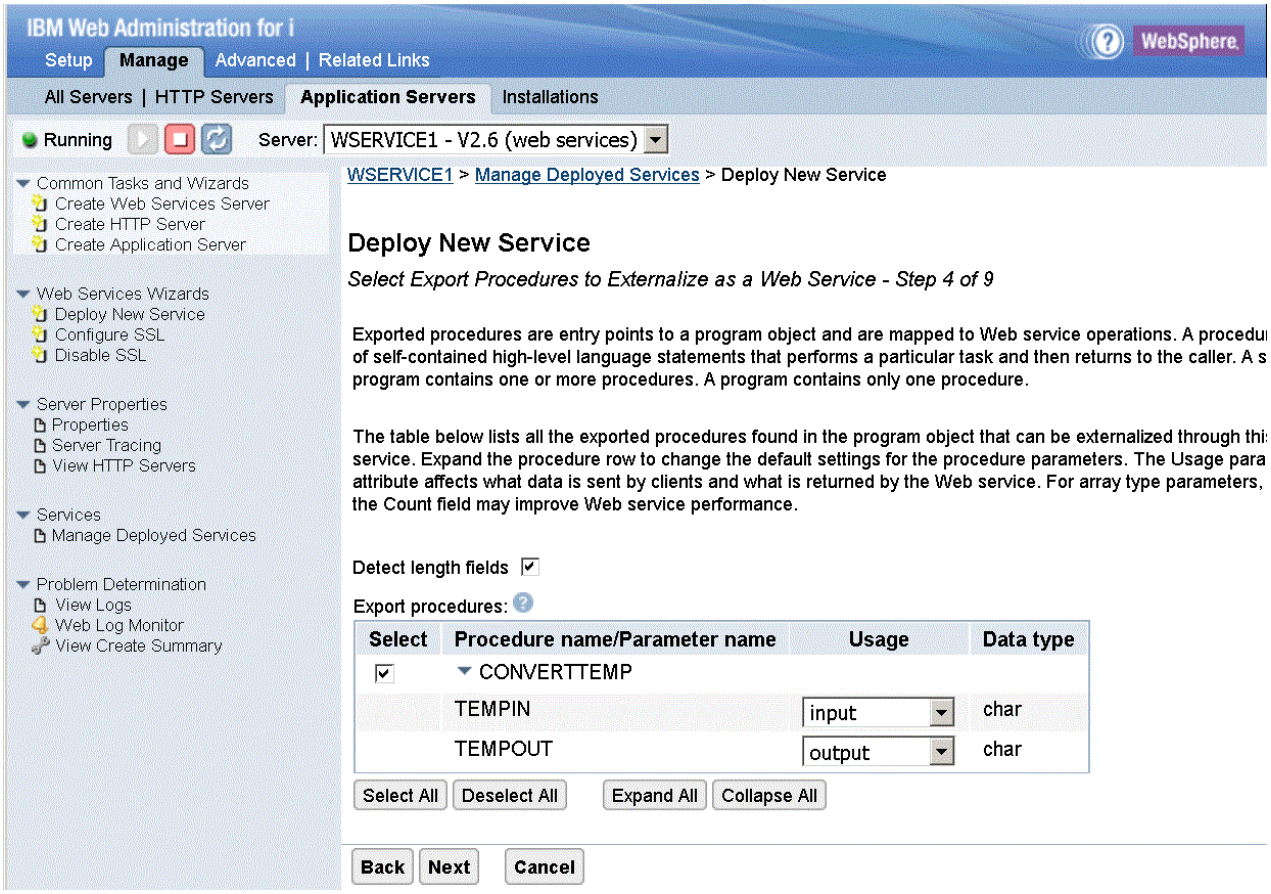


Figure 30: Select export procedures to externalize as a web service

The parameter attributes are modifiable. In most cases you want to modify the parameter attributes to control what data is to be sent by web service clients and what data is to be returned in the responses to the client requests.

The **Detect length fields** should always be selected. The only reason that you have the option to deselect is for web services that were deployed prior to this support. The benefits of length detection include the following:

- Support of nested output arrays in an efficient manner. This is done by assuming that any numeric field that immediately precedes an array field with the same name as the array field appended with _LENGTH is a length field that will be used to indicate the actual number of elements in the array. Without this support, empty elements would be returned in the response.
- Improves the processing of very large output character fields. This is done by assuming that any numeric field that immediately precedes a character field with the same name as the character field appended with _LENGTH is a length field that will be used to indicate the actual number of characters in the field. Without this support, the length of the string is determined by traversing the field a byte at a time, from right to left, looking for the first non-blank character.
- Preserves case sensitivity of identifiers used in the program object. Prior to this support, identifier generation was left up to the web services engine being used.
- Preserves field ordering in the input and output schema. Prior to the detect length fields support, the ordering of parameter fields and fields within structures was not preserved, and in most cases the ordering is based on alphabetic ordering.

The **Export procedures** table shown in Figure 30 on page 71 contains the following information:

- The **Select** column specifies which procedures are to be externalized as a service operation. You should select only those procedures that are to be exposed as web service operations. At least one procedure must be selected. A procedure may not be selectable, that is, disabled, if the procedure definition contains parameter types that are not supported[9].
- The **Procedure name/Parameter name** column identifies the program object's procedures and the parameters for each procedure.
- The **Usage** column indicates which of the procedure's parameters is input, output, or both input/output. The designation affects what a web service client needs to specify on a request to the web service. Designating a parameter as input or input/output means that a web service client has to pass data corresponding to the parameter and the data is passed to the program object. Designating a parameter as output or input/output means that after the program object is invoked, data corresponding to the parameter is returned as part of the response to the web service client request.
- The **Data** type column indicates the type of the data.

In this example, the TEMPIN parameter is an input parameter, and the TEMPOUT parameter is the output parameter. This means that a web service client will need to only pass data corresponding to the TEMPIN parameter, and the response to the client request will be returned in the TEMPOUT parameter.

Click on the **Next** button at the bottom of the form.

Panel (REST-ONLY): Specify resource method information

The panel shown in Figure 31 on page 73 is only displayed when deploying a RESTful web service. This panel will be shown for each procedure that was selected to be externalized as a resource method.

[9] See "PCML considerations" on page 132 for information regarding product restrictions and limitations.

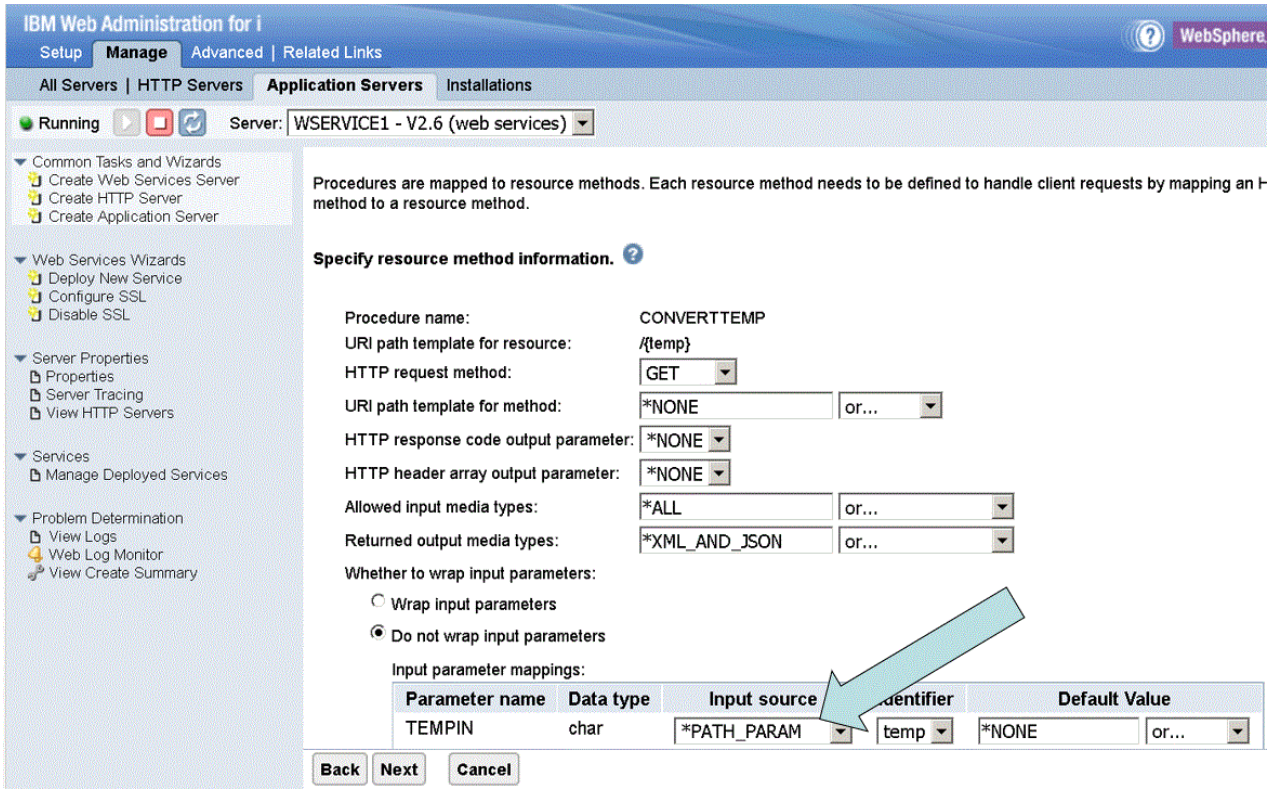


Figure 31: Specify resource method information

Let us examine the fields on the panel:

- **Procedure name**: Is the name of the procedure for which resource method information is to be set.
- **URI path template for resource**: Is the URI path template that is specified for the resource (see Figure 29 on page 70). Any path to the resource must include a value that is allowable by what is specified for the resource URI path template.
- **HTTP request method**: The HTTP method that the procedure will handle. The four methods are GET, PUT, DELETE, and POST. Resources are not required to permit all HTTP methods for all clients.

  The HTTP GET method retrieves a resource representation. It is safe and should be idempotent. Repeated GET requests do not change any resources.

  The HTTP PUT method is often used to update resources. When a resource must be updated, an HTTP PUT method is issued at the resource URL with the new resource data as the request entity, also known as the message body. The HTTP PUT method should be idempotent so multiple identical PUT requests with the same entity to the same URL yields the same result as if only one PUT request was issued.

  The HTTP DELETE method removes a resource at a specific URL.

  The HTTP POST method is often used while creating a resource. It is not safe and is not idempotent.
- **HTTP response code output parameter**: You can designate an output parameter that will contain the HTTP response code returned to the client by the resource method. The output parameter must be of type integer. After invoking the web service implementation code, the status code is checked to determine whether it is an error status code, and if it is the error is returned to the client without processing any output parameters.

  Note: Output parameters used for HTTP status codes are not returned as part of the response. That is, the parameter is hidden from the client.
- **HTTP header array output parameter**: You can designate an output parameter that will contain HTTP headers to be returned in the client response by the resource method. The output parameter must be an array of type character.

Note: Output parameters used for HTTP headers are not returned as part of the response. That is, the parameter is hidden from the client.

- **Allow input media types**: Specify or select the MIME types the resource method will accept. You can select the type from the second select list field or specify one or more comma-delimited MIME types. Specifying types is allowed only when input parameters are unwrapped. If there is an input parameter that is a structure, then the only input media type allowed is *XML, *JSON, or *XML_AND_JSON[10].
- **Returned output media types**: Specify or select the MIME types the resource method will return. What is actually returned to the client is dependent on the content negotiation that is performed between the client and the server and is dependent on what is set in the client HTTP headers. You can select the type from the second select list field or specify one or more comma-delimited MIME types. Specifying user-defined types is allowed as long as there is zero or one primitive (parameters that are not structures) output parameter.
- **Whether to wrap input parameters**: Specify whether to wrap all input parameters in a structure or not. The decision whether to wrap or not wrap input parameters is a based on whether you have more than one input parameter and whether or not you are planning on injecting a value in one of the input parameters.
  - If you choose to wrap input parameters, then all input parameters will be wrapped in a structure, the injection of values into parameters is not allowed, the HTTP method for the resource must be POST or PUT, and the HTTP requests must have a structured messages payload with type XML or JSON.
  - If you choose to not wrap input parameters, then there can only be at most one parameter that is not injected with a value from an input source. This option is not allowed if there exists input parameters that are arrays, or more than one parameter that is a structure, or there is a primitive input parameter that has a type other than "int", "char", "byte", "float", "packed" or "zoned".

    Only parameters that have a primitive type may be injected with a value from an input source. You can inject input parameters with values from the following input sources:

    - `*QUERY_PARAM`: The value is retrieved from a query parameter extracted from the request URL. Query parameters are appended to the URL after a "?" with name-value pairs. For instance, if the URL is `http://example.com/collection?itemID=itemIDValue`, the query parameter name is `itemID` and `itemIDValue` is the value. Query parameters are often used when filtering or paging through HTTP GET requests.
    - `*PATH_PARAM`: The value is retrieved from the URI path parameter extracted from the request URI. Path parameters are part of the URL. For example, the URI path template can include `/collection/{item}`, where `{item}` is a path parameter that identifies the item. If there are no parameters set in the URI path templates, you will not be allowed to specify `*PATH_PARAM` as an input source.
    - `*FORM_PARAM`: The value is retrieved from a HTML form parameter extracted from HTML form elements. Form parameters are used when submitting a HTML form from a browser with a media type of `application/x-www-form-urlencoded`. The form parameters and values are encoded in the request message body in the form like the following: `firstParameter=firstValue&secondParameter=secondValue`. In order for you to specify `*FORM_PARAM` as an input source, the HTTP request method must be set to PUT or POST.
    - `*COOKIE_PARAM`: The value is extracted from the HTTP cookie value in the incoming HTTP request. Cookie parameters are special HTTP headers. While cookies are associated with storing session identification or stateful data that is not accepted as RESTful, cookies can contain stateless information.
    - `*HEADER_PARAM`: The value is extracted from the HTTP header value in the incoming HTTP request. Headers often contain control metadata information for the client, intermediary, or server.
    - `*MATRIX_PARAM`: The value is retrieved from the URI matrix parameter. Matrix parameters are part of the URL. For example, if the URL includes the path segment, /

[10] The keyword *XML_AND_JSON indicates that the method will accept a payload of either XML or JSON. If you specify *XML or *JSON, you are indicating that the payload must be the specified media type; otherwise, the request will be rejected.

`collection;itemID=itemIDValue`, the matrix parameter name is `itemID` and `itemIDValue` is the value.

For all the input sources except for `*PATH_PARAM`, you may specify a default value for the input source that will get injected in the parameter if the input source variable is not found in the client request.

Let us take a look at some examples. Assume that we have an exported procedure named `CONVERTTEMP` that has one input parameter named `TEMPIN` of type character. If we choose to wrap the input parameters, then the XML request would be as follows:

```
<CONVERTTEMPInput><TEMPIN>2337</TEMPIN></CONVERTTEMPInput>
```

and the JSON request would be:

```
{"TEMPIN": "2337"}
```

Now assume that we had chosen to unwrap the input parameters and that we specified a URI path template of `/{temp}`. Then the parameter `TEMPIN` can be injected with an input source of `*PATH_PARM` with identifier of `temp`. If the client sent in a request with the URL `http://host:port/web/service/CONVERTTEMP2/2337`, the 2337 in the path will be inserted into the parameter `TEMPIN`.

Let us take a look at a more complicated example, where the exported procedure named `CONVERTTEMP` has two input parameters: one is `TEMPIN` of type character and another is a structure `Struct1` which has two fields - `FLD1` and `FLD2` - of type integer. If we choose to wrap the input parameters, then the XML request would be as follows:

```
<CONVERTTEMPInput>
   <TEMPIN>23</TEMPIN>
   <STRUCT1>
     <FLD1>00</FLD1>
     <FLD2>01</FLD2>
   </STRUCT1>
</CONVERTTEMPInput>
```

and the JSON request would be:

```
{
   "TEMPIN": "23",
   "STRUCT1": {
       "FLD1": "00", "FLD2": "01"
   }
}
```

Now assume that we had chosen to unwrap the input parameters. `TEMPIN` must be injected with a value from an input source since the other input parameter, `Struct1`, is a structure. `TEMPIN` will be injected from a query parameter (`*QUERY_PARAM`) with identifier `temp`. The input data for the structure parameter must be carried in the request body. If client sent in a request using URL `http://host:port/web/service/CONVERTTEMP2?temp=2337`, then the value 2337 is injected in `TEMPIN`, and the XML request would look like the following:

```
<STRUCT1>
     <FLD1>00</FLD1>
     <FLD2>01</FLD2>
</STRUCT1>
```

and the JSON request would be:

```
{
   "FLD1": "00",
```

```
   "FLD2": "01"
}
```

In Figure 31 on page 73, we have chosen to inject the input parameter with a variable (`temp`) from the URI template path.

Click on the **Next** button at the bottom of the form.

Panel: Specify user ID for the service

We now need to specify the user ID that the service will run under. As shown in Figure 32 on page 76, you can run the service under the server's user ID or you can specify an existing user ID that the service will run under.

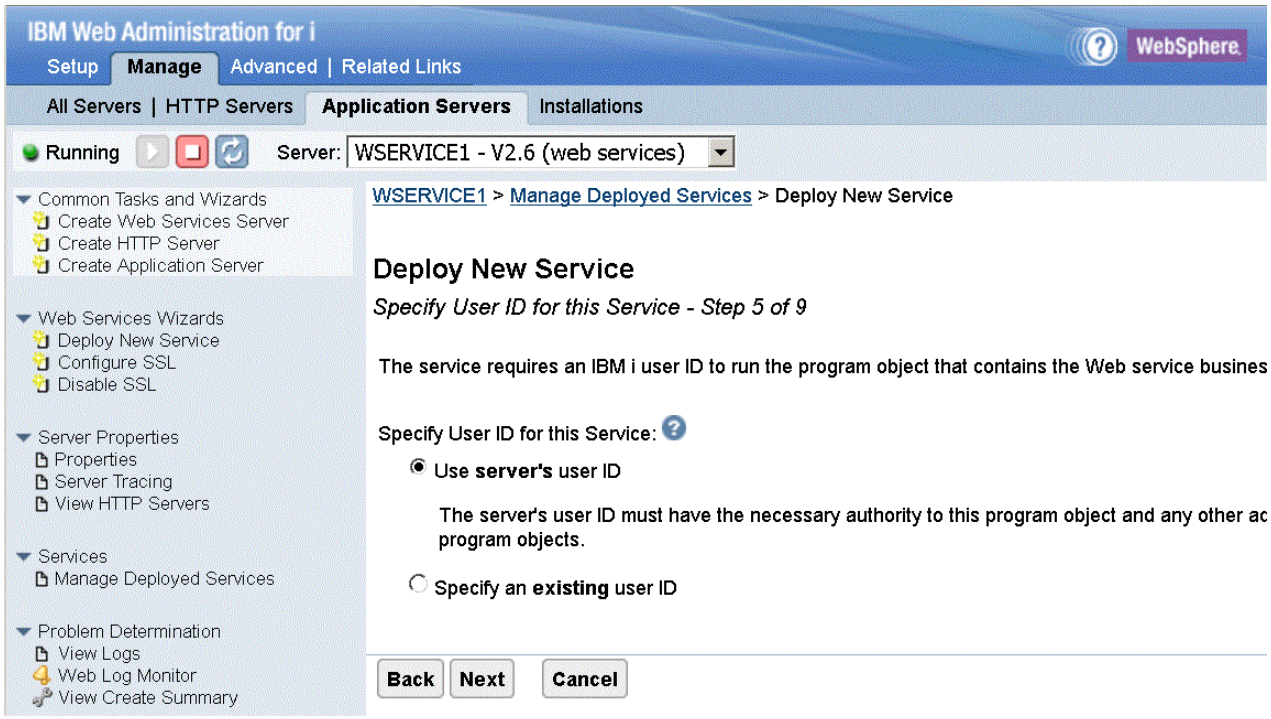


Figure 32: Specify web services server user ID

In order for the web service to run correctly, the user ID status must be set to *ENABLED and the password must be set to a value other than *NONE. If a user ID is specified that is disabled or has a password of *NONE, a warning message is displayed and the service may not run correctly. In addition, ensure that the specified user ID has the proper authorities to any resources and objects that the program object needs, such as libraries, databases and files.

Click on the **Next** button at the bottom of the form.

Panel: Specify library list

Specify any libraries that the program object needs to function properly (see Figure 33 on page 77). You have the option of putting the libraries at the start of the user portion of the library list or at the end of the user portion of the library list.

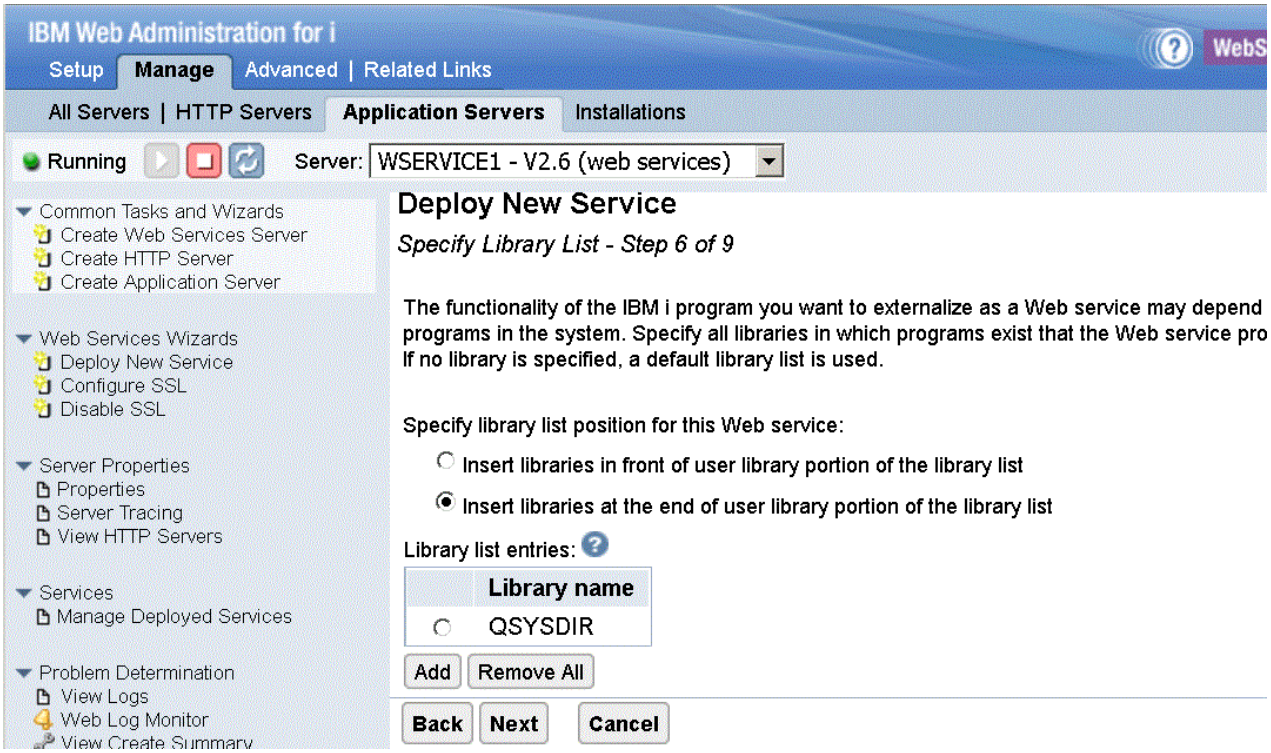


Figure 33: Specify library list

Click on the **Next** button at the bottom of the form.

Panel: Specify transport information to be passed

Specify[11] what transport information related to the client request is to be passed to the web service implementation code (see Figure 34 on page 78). The information is passed as environment variables.

[11] The **Request Information** panel will not be shown if the deployed web service does not have the ability to handle transport information. If this is the case, you will have to redeploy the web service to get the **Request Information** panel to be displayed.

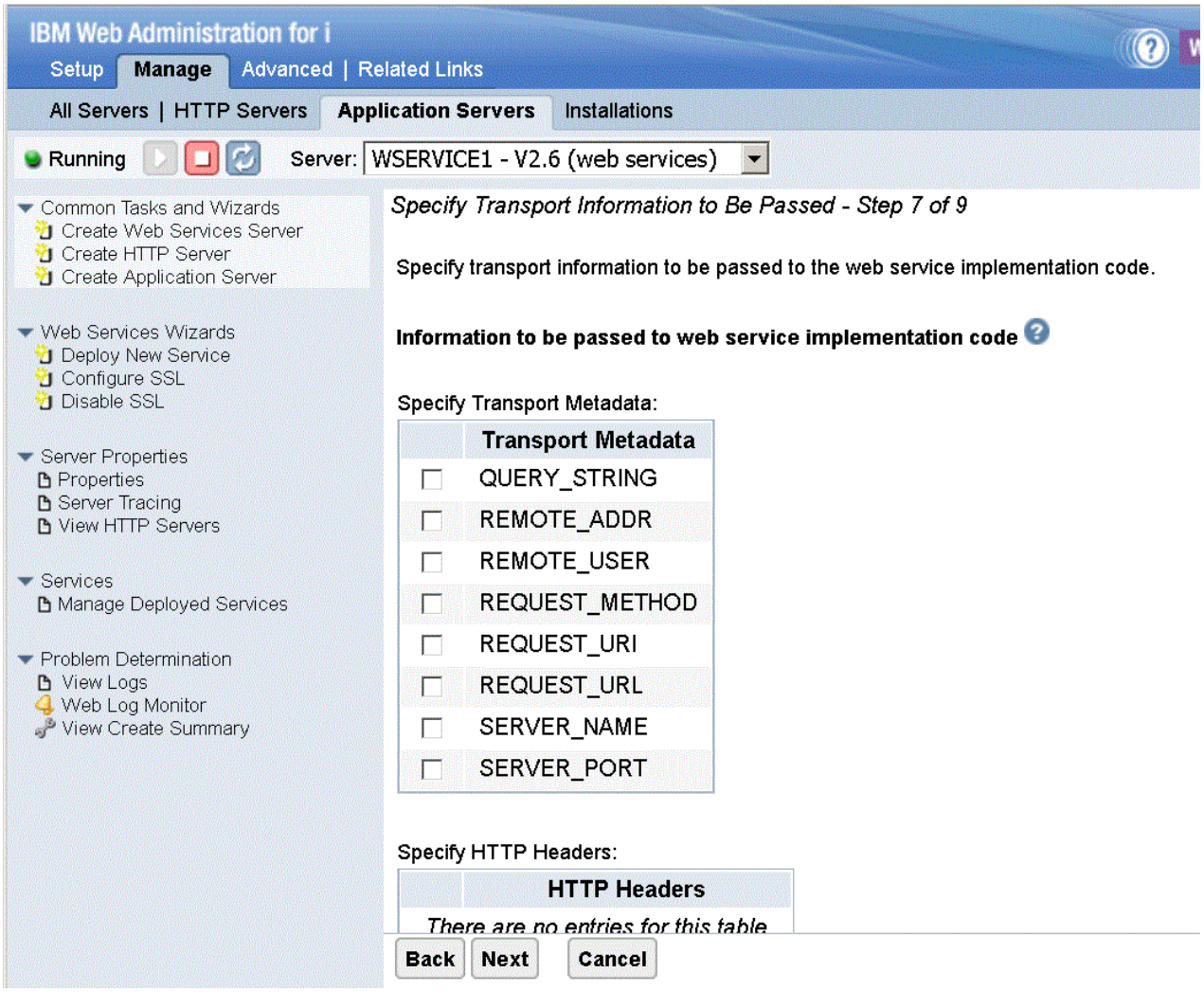


Figure 34: Specify transport information to be passed

For example, if the transport metadata REMOTE_ADDR is selected, it will be passed to the web service implementation code in an environment variable named REMOTE_ADDR.

HTTP headers indicates what transport headers (e.g. HTTP headers) to pass to the web service implementation code. Transport headers are passed as environment variables. The environment variable name for HTTP headers is made up of the specified HTTP header prefixed with HTTP_, all upper-cased. For example, if `Content-type` is specified, then the environment variable name would be `HTTP_CONTENT-TYPE`. If an HTTP header was not passed in on the web service request, the environment variable value will be set to the null string.

Click on the **Next** button at the bottom of the form.

Panel (SOAP-ONLY): Specify WSDL options

Specify options that affect the generated WSDL file associated with the web service (see Figure 35 on page 79).

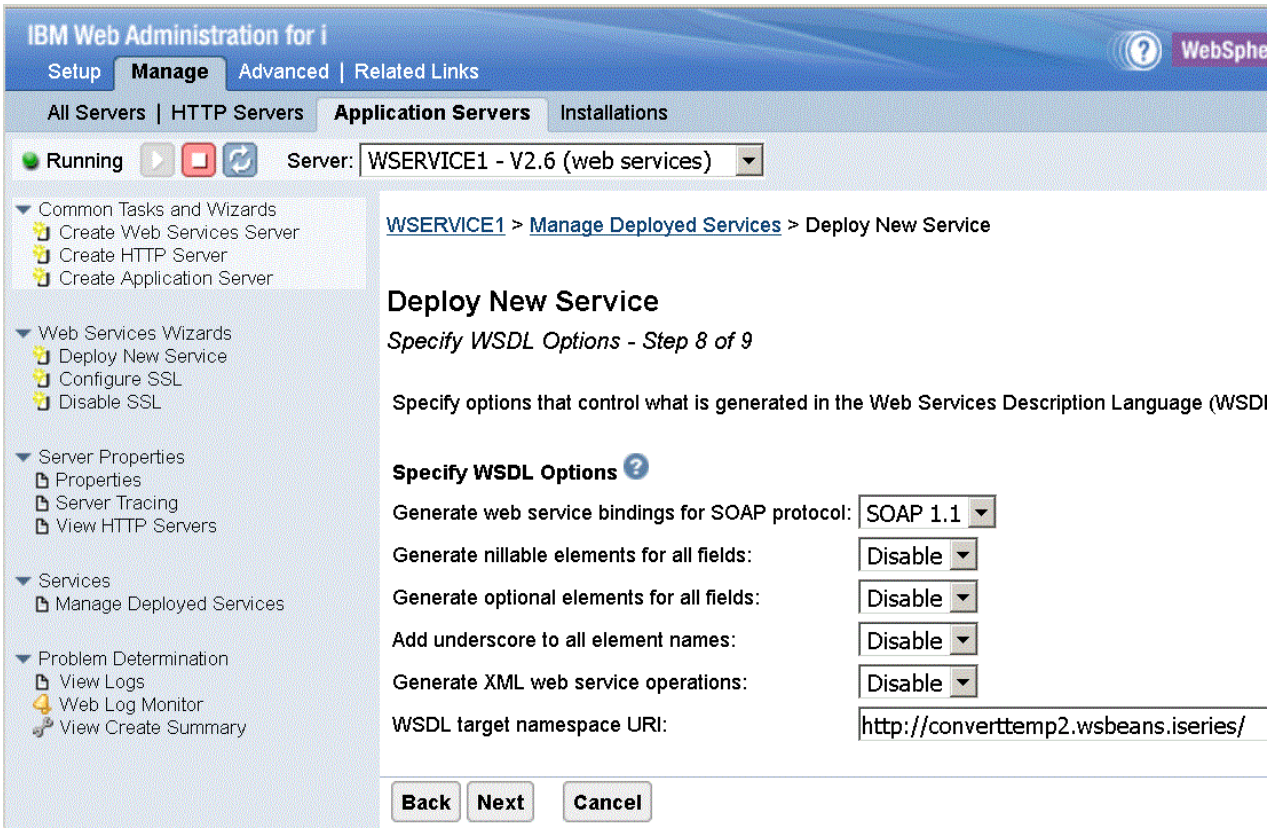


Figure 35: Specify WSDL options

Let us examine the fields on the panel:

- **Generate web service bindings for SOAP protocol**: You can select which version of SOAP to use, either SOAP 1.1 or SOAP 1.2.
- **Generate nillable elements for all fields**: Specifies whether to generate nillable elements for all fields when generating a new WSDL file for the web service. Nillable fields allows a client to sent a request with no value. If a value is not sent for a field, the web service implementation code will either receive the field with all blanks if the field is a character field, or a value of zero for numeric fields.
- **Generate optional elements for all fields**: Specifies whether to generate optional elements for all fields when generating a new WSDL file for the web service. Optional elements can be eliminated from the client request. If an element is not received that corresponds to a field, the web service implementation code will either receive the field with all blanks if the field is a character field, or a value of zero for numeric fields.
- **Add underscore to all element names**: Specifies whether to add underscore to all elements names when generating WSDL file for the web service. This is a legacy option and should only be used when moving web services from version 1.3 of the integrated web services server, which always added the underscore to element names.
- **Generate XML web service operations**: Specifies whether to generate XML web service operations when generating a new WSDL file for the web service. This is a legacy option which basically generated an additional operation that return the SOAP response as string. In most cases this should not be used since it adds complexity to the web service.
- **WSDL target namespace URI**: The WSDL target namespace URI. Target namespace is the namespace for the WSDL file that is generated for the web service. The target namespace is used for the names of messages and the port type, binding and service defined in the WSDL file. The value must take the form of a valid URI (for example, `http://www.mycompany.com/myservice`).

Click on the **Next** button at the bottom of the form.

Panel: Confirm deployment information for web service

The wizard shows you a summary page (see Figure 36 on page 80), giving you the chance to see the details relating to the web service being deployed before it actually kicks off the task of deploying the service. There will be different tabs shown depending on whether the web service is SOAP-based or REST-based web service.

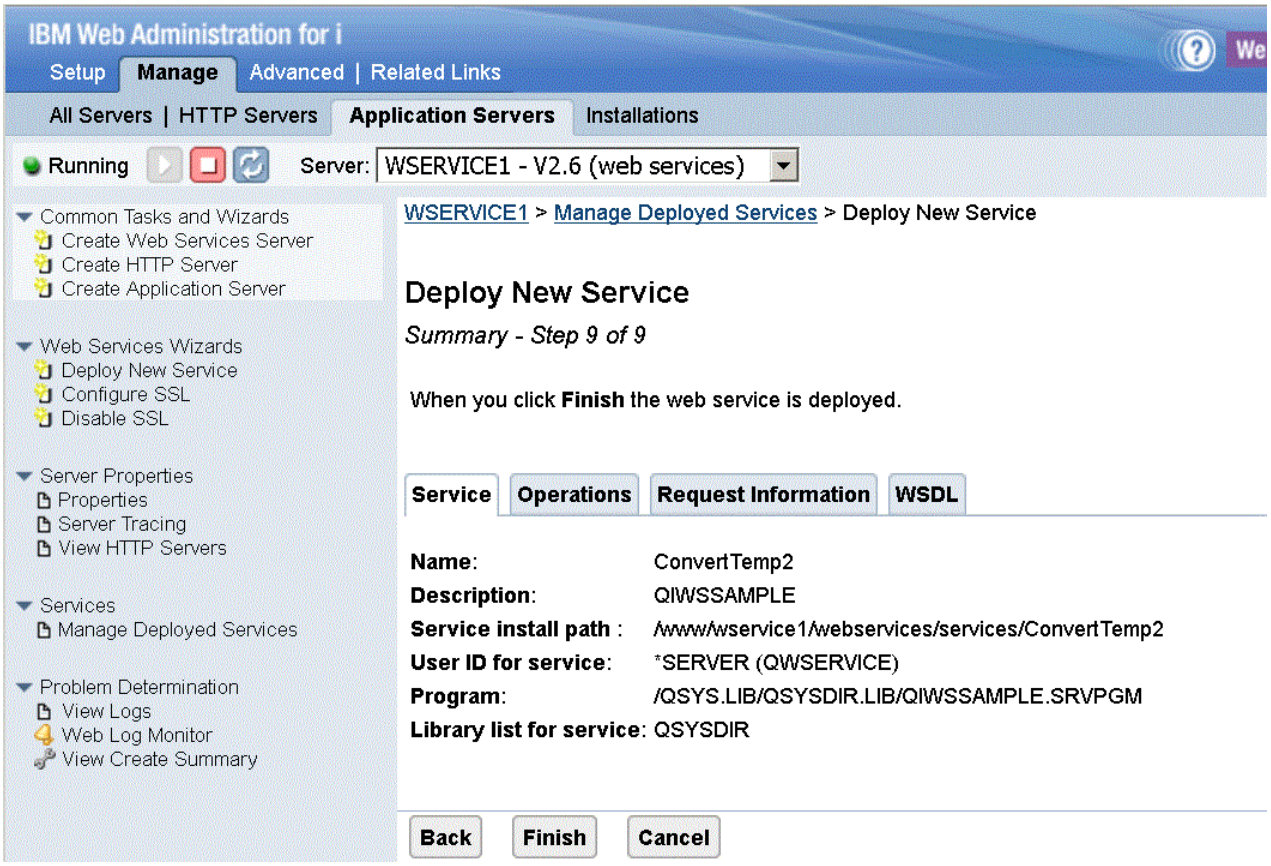


Figure 36: Summary – Services tab

Let us examine the summary panel in a little more detail. On the **Services** tab, you will see information about the service being deployed.

If you click on the **Operations** tab, you will see the web service operations that correspond to the procedure that was selected to be deployed as a web service operation. Figure 37 on page 81 shows the panel for a SOAP-based web service.

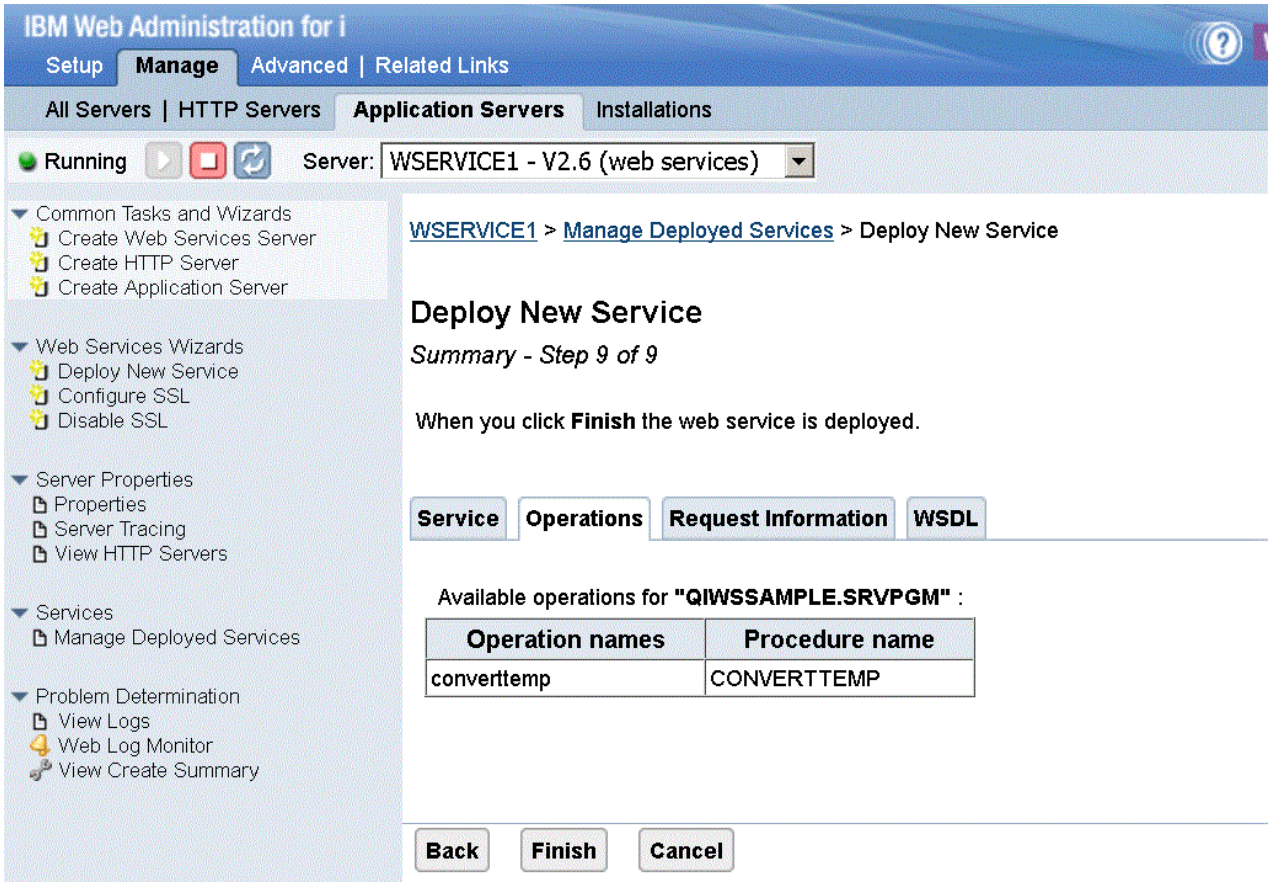


Figure 37: Summary – Operations tab (SOAP-ONLY)

For REST-based web services, the tab that will be shown is the **Methods** tab as shown in Figure 38 on page 82.

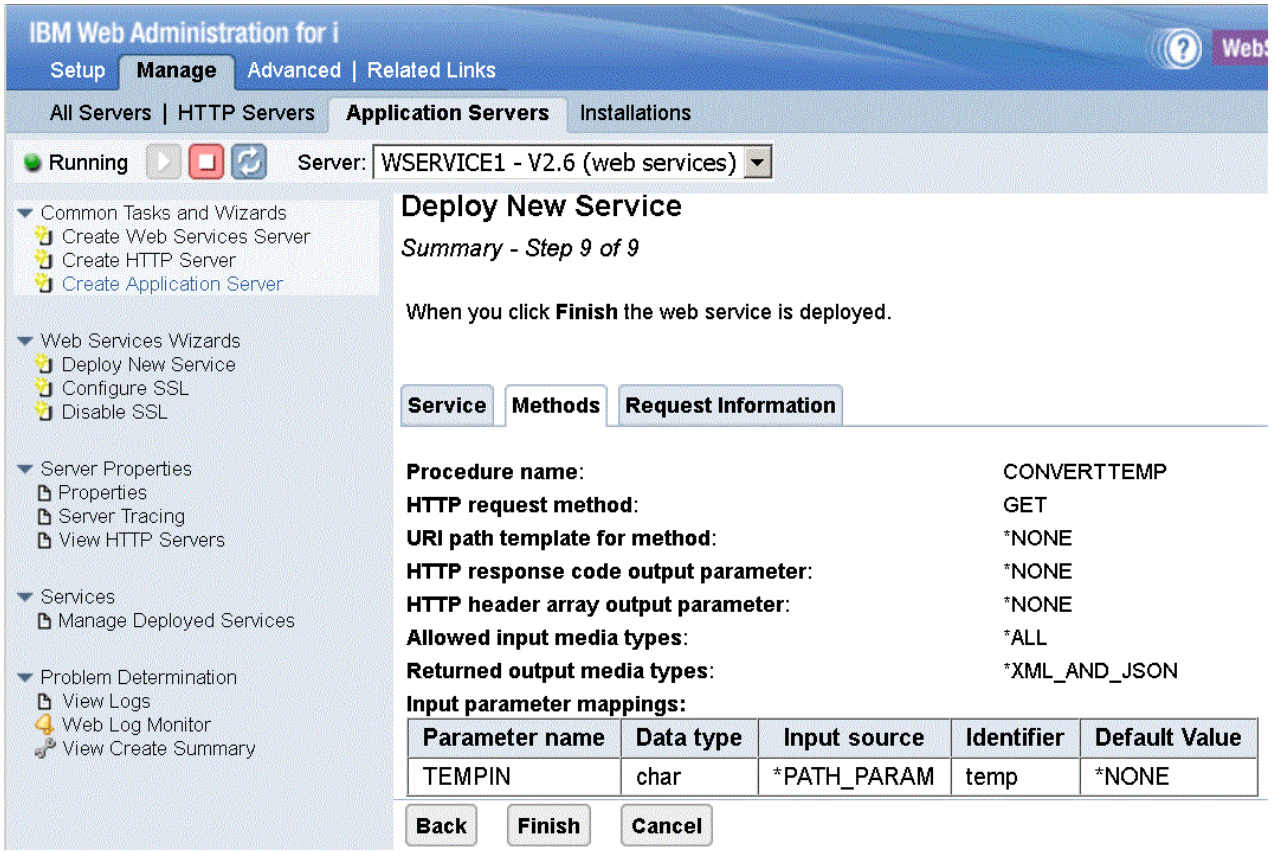


Figure 38: Summary – Methods tab (REST-ONLY)

If you click on the **Request Information** tab, you will see the transport information to be passed to the web service implementation code (see Figure 39 on page 83).

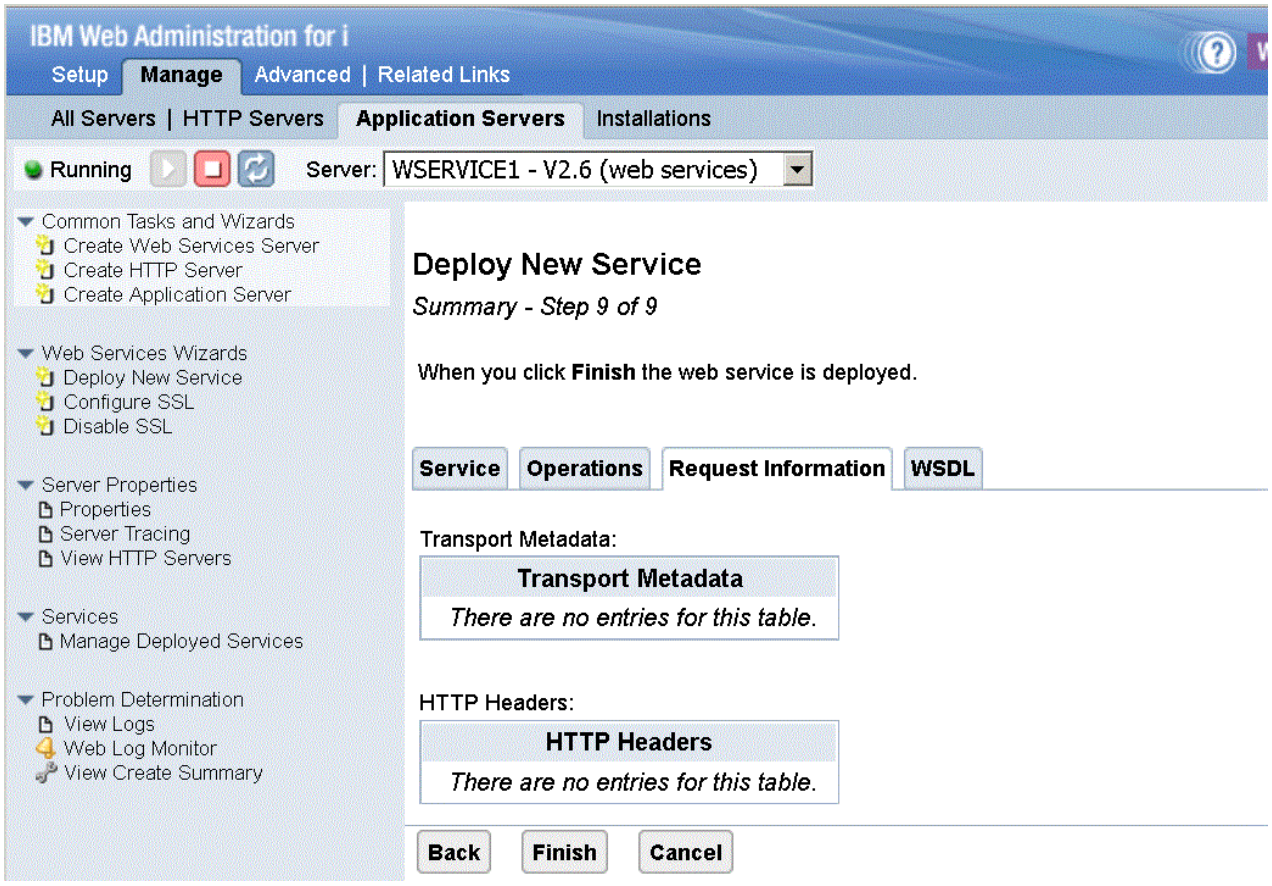


Figure 39: Summary – Request Information tab

If the web service that is being deployed is SOAP-based, there will be a **WSDL** tab where you will see the WSDL options that have been specified for the web service (see Figure 40 on page 84).

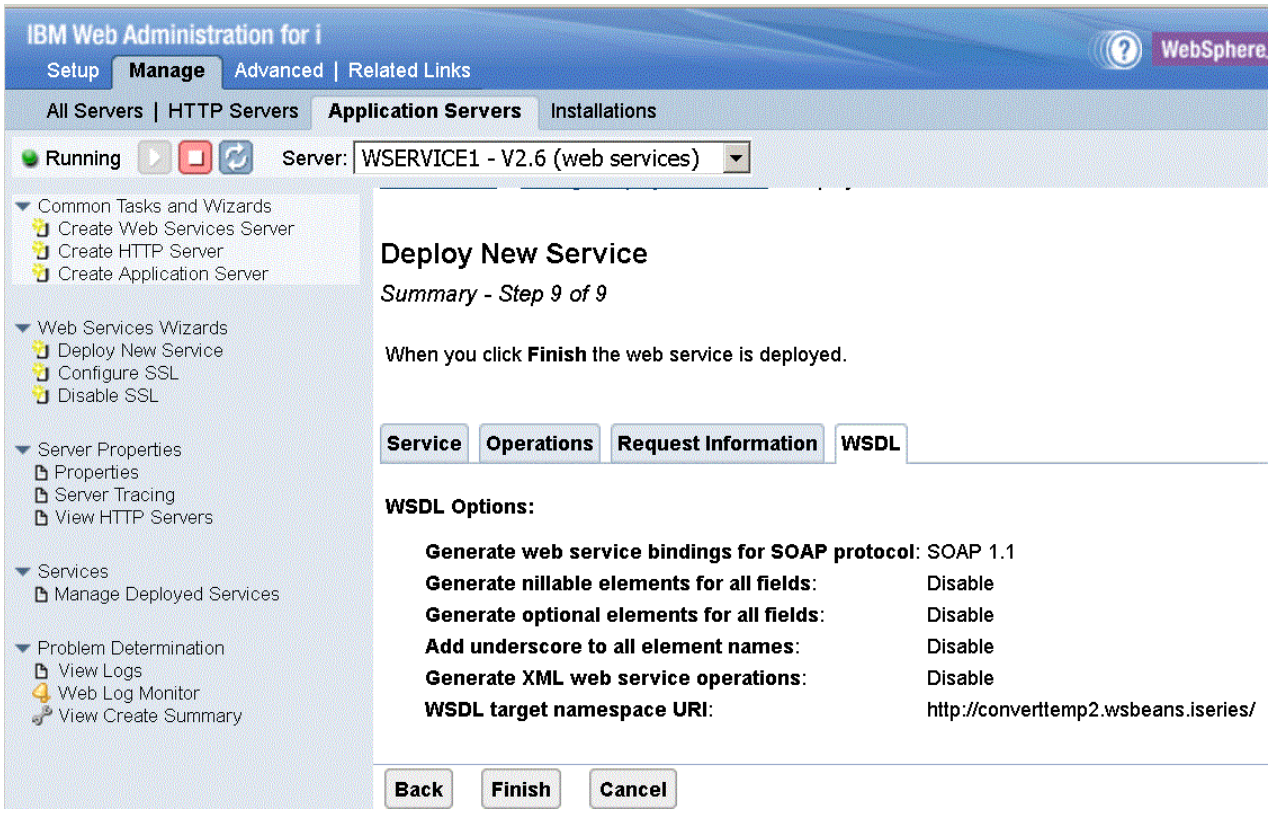


Figure 40: Summary – WSDL tab (SOAP-ONLY)

Clicking on the **Finish** button at the bottom of the summary page will result in the panel shown in Figure 41 on page 84 being displayed by the wizard:

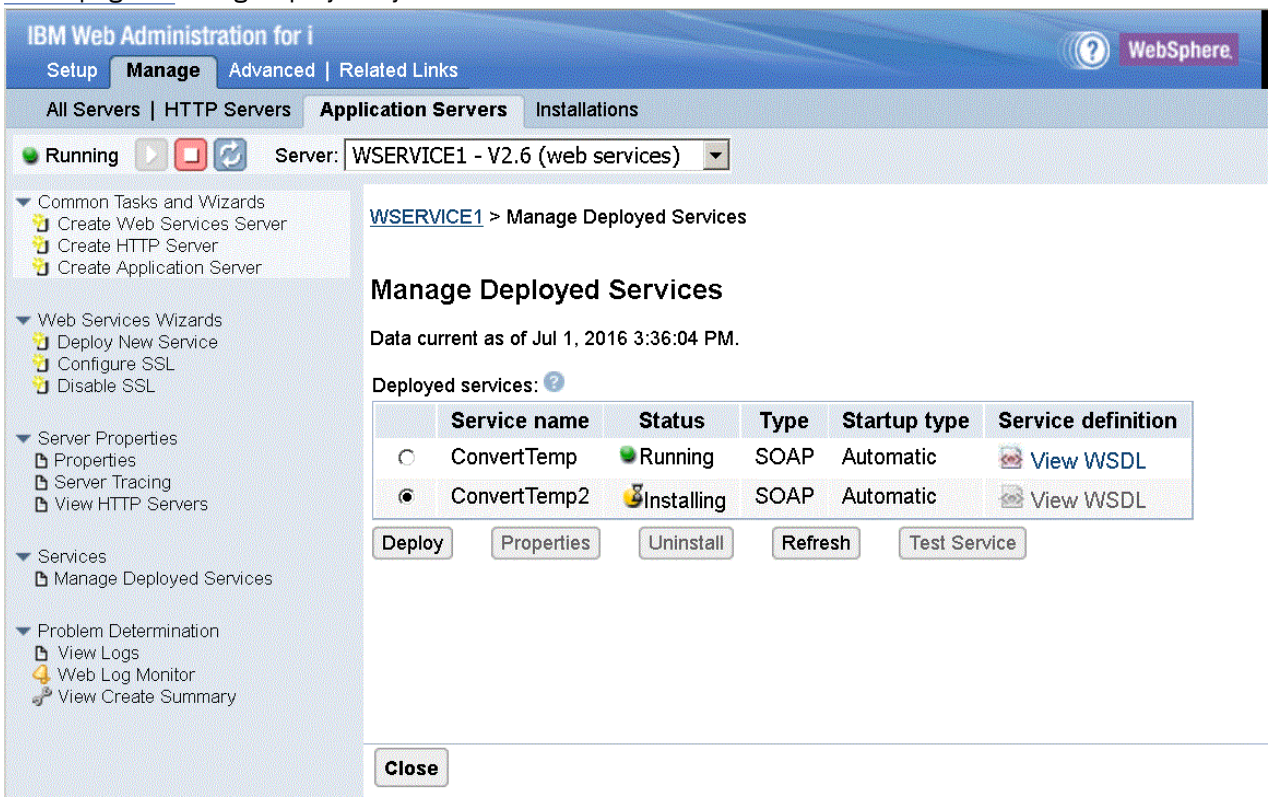


Figure 41: Web service being deployed

When the web service is deployed the deployed service becomes active (green dot to the left of service name) as in Figure 42 on page 85.

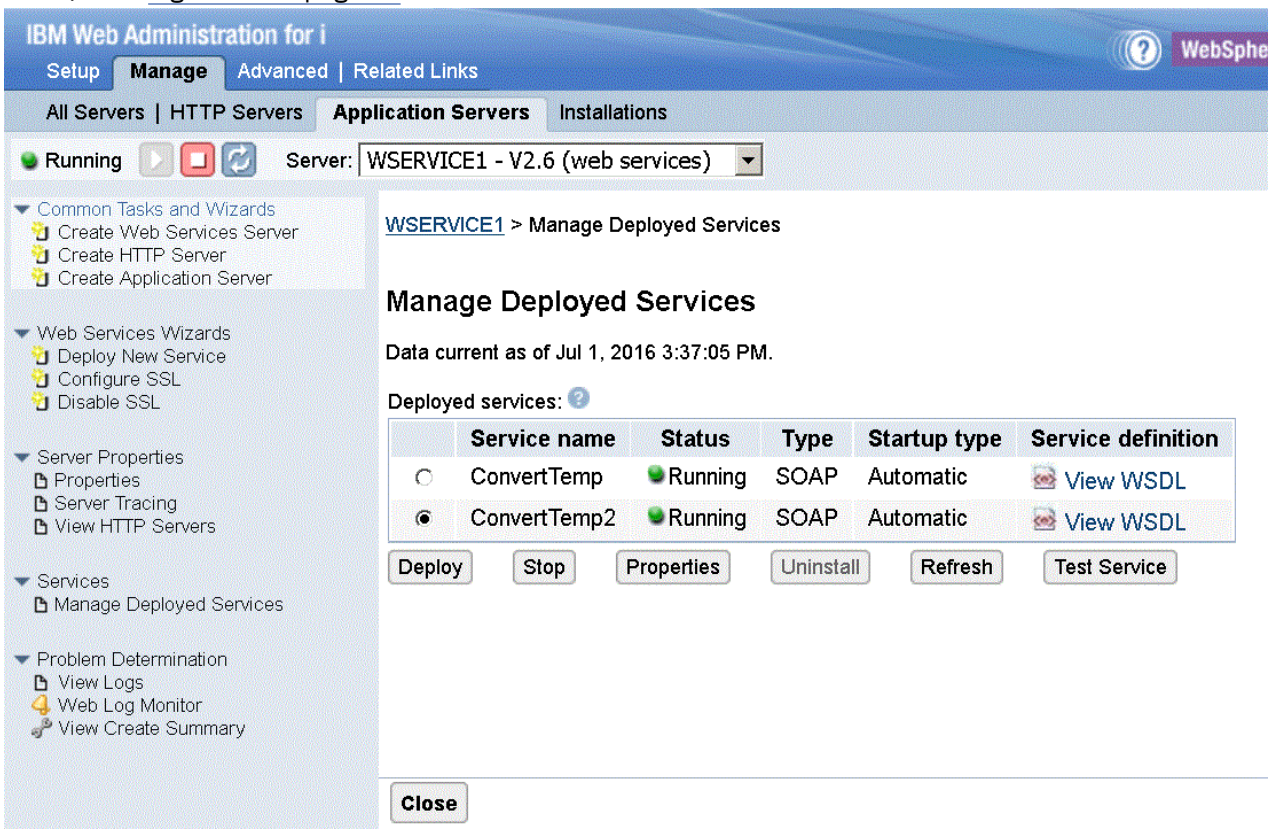


Figure 42: Web service is active

The configure SSL wizard

The **Configure SSL** wizard provides a way to configure SSL for integrated application server (version 8.5 and above), integrated web services server (version 2.6 and above) and stand-alone Liberty servers running on IBM i.

To use the wizard, you will need the following:

- Digital certificate manager (option 34 of the base operating system) is required when the *SYSTEM store is used for SSL.
- The user profile accessing the Web Administration for i interface must have access to the application server and the associated HTTP server or can be a user profile with the *ALLOBJ, *IOSYSCFG and *SECADM authorities, or a user who has been granted permission through the permissions support discussed previously in this document.

If you enable SSL in the application server and the server is associated with an HTTP server, the wizard ensures that the HTTP server plug-in file is updated so that the HTTP server can communicate with the application server.

You can use the wizard to enable SSL in the HTTP server associated with the web application server or the application server itself.

The disable SSL wizard

The **Disable SSL** wizard enables you to remove SSL from an application server. If there is associated HTTP server with the application server, the corresponding HTTP plug-in configuration may also be updated by the wizard.

In order to use the wizard, the user profile accessing the Web Administration for i interface must have access to the application server and the associated HTTP server or can be a user profile with the *ALLOBJ, *IOSYSCFG and *SECADM authorities, or a user who has been granted permission through the permissions support discussed previously in this document.

Server properties

The next group of navigation links relate to the integrated web services server. The arrow shown in Figure 43 on page 86 points to the location of server property links.

Figure 43: Manage server panel - server properties

There are properties specific to the server, tracing and showing associated HTTP servers. Let us take a look at each.

Properties relating to the server

If you click on the **Properties** link, you will see the panel shown in Figure 44 on page 87

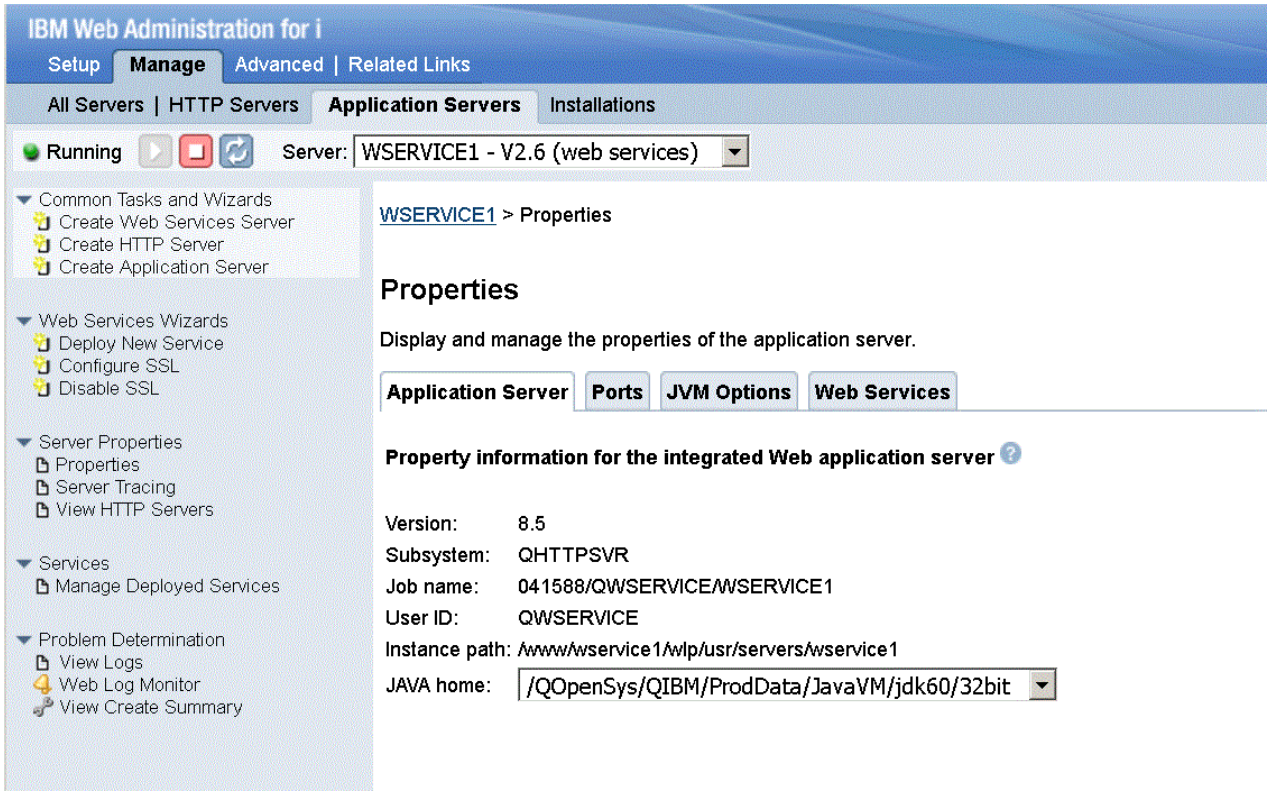


Figure 44: Server properties - application server tab

The first tab that is shown is the **Application Server** tab, which contains information about the application server, including the version, subsystem server is running in, the job name of the server (if running), the user profile the server is running under, the path to the server and the Java runtime environment that is being used to run the server. You have the ability to choose a Java runtime environment if you so choose.

The **Ports** tab contains the HTTP and HTTP SSL ports that the application server is listening on (see Figure 45 on page 88) .

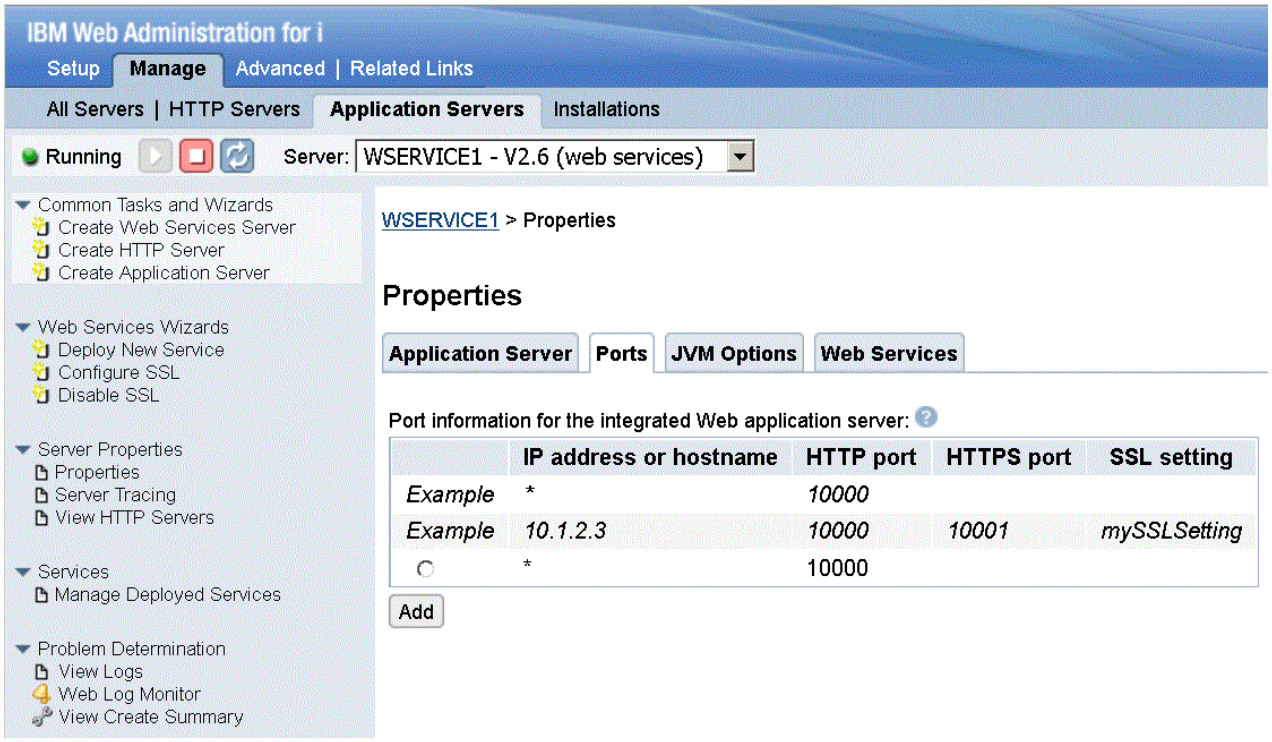


Figure 45: Server properties - ports tab

You can add ports easily from within this panel, but if want more control on adding SSL ports you should use the **Configure SSL** wizard to walk you through the process.

Figure 46 on page 89 shows the **JVM Options** tab that contains the default JVM options for the server. Any updates to the JVM options requires a server restart.

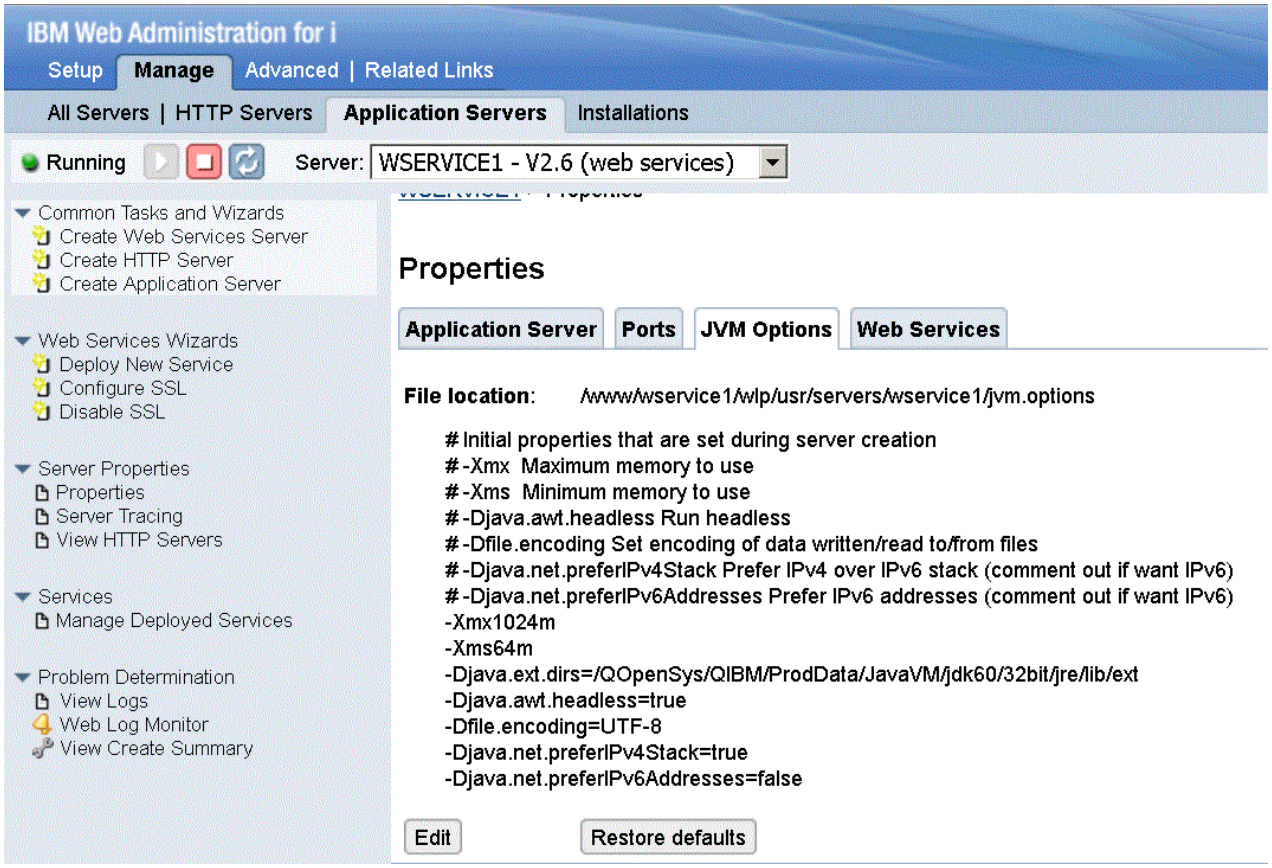


Figure 46: Server properties - JVM Options tab

Figure 47 on page 90 shows the **Web Services** tab that contains web services-specific information.

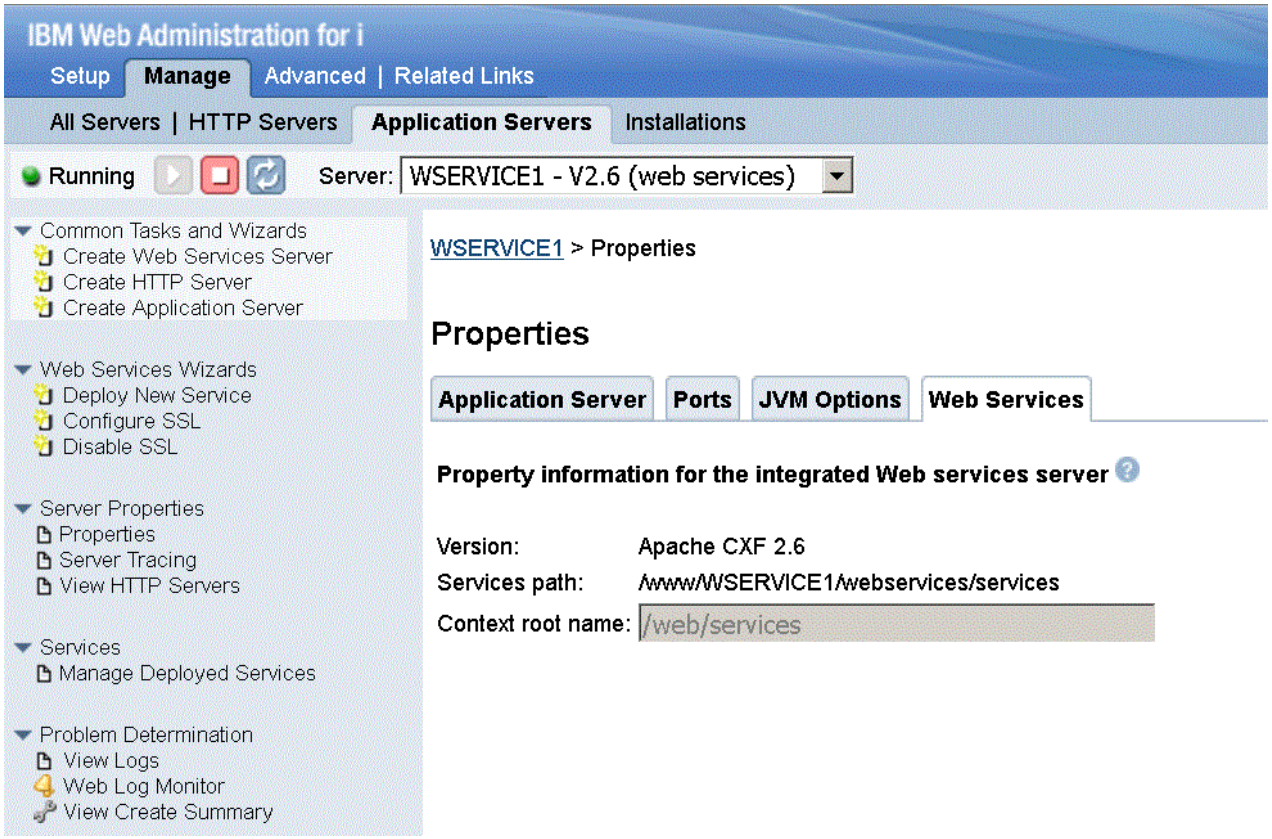


Figure 47: Server properties - Web Services tab

The web service information includes the version of the web services engine, the path to the directory where web services are located, and the context root for all web services.

Whatever is specified in the context root is what must be specified in the URL. For example, `http://host:port/web/services/ConvertTemp`.

The server must be inactive if you want to change the context root.

Server tracing

If you click on the **Server Tracing** link, you will see the panel shown in Figure 48 on page 91

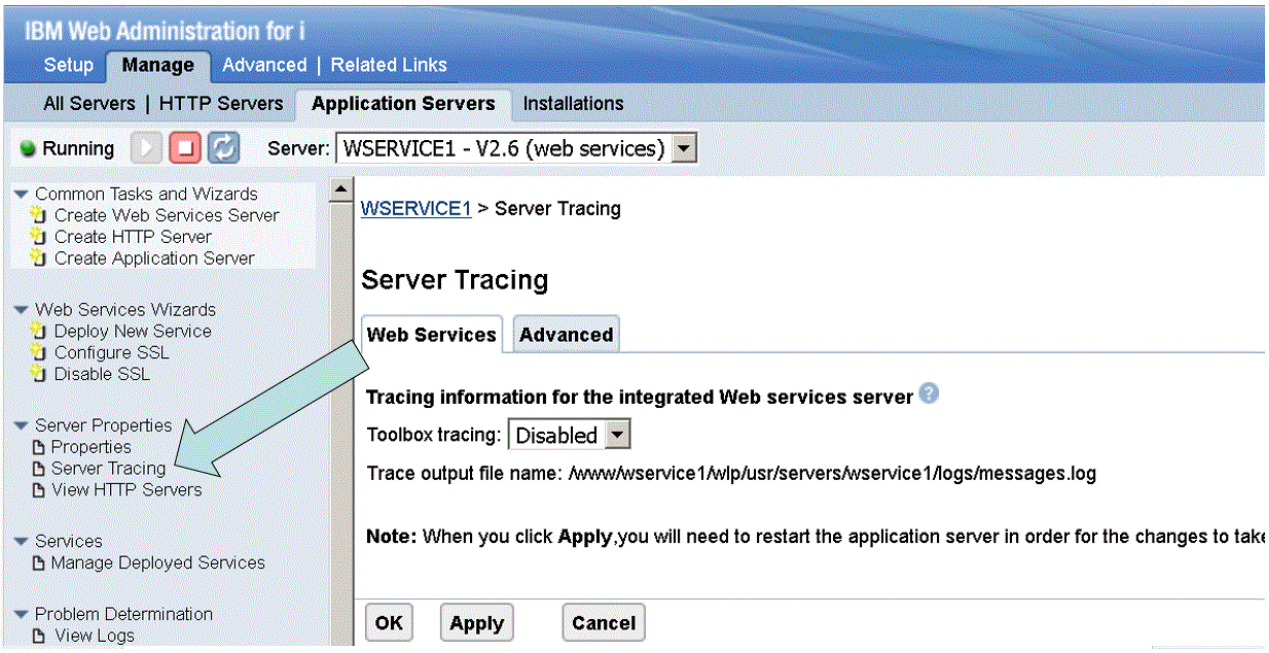


Figure 48: Server tracing properties

The first tab that is shown is **Web Services** tab, which contains tracing properties directly relating to web services. You can enable or disable Java toolbox tracing. Recall that Java toolbox is used to invoke ILE program objects running within a host server job. The Java toolbox tracing enables you to see the data flow between the integrated web services server and the host server job in which the ILE program object is running.

This trace produces lots of information and will affect the performance of the web service, so you do not want to enable the trace in a production environment unless you really need to. The output of the trace is written to the file shown on the panel, and may be viewed from with the Web Administration for i interface by clicking on the **View Logs** link in the navigation panel, which is discussed in “Problem determination” on page 100.

The **Advanced** tab shown in Figure 48 on page 91 allows you to enable tracing for the integrate web services server runtime, and should not be enabled unless instructed to by IBM service personal.

For more information on tracing, see “ Tracing” on page 139.

Viewing HTTP servers associated with integrated web services server

If you click on the **View HTTP Servers** link, you will see the panel shown in Figure 49 on page 92, which lists the HTTP servers associated with the integrated web services server.

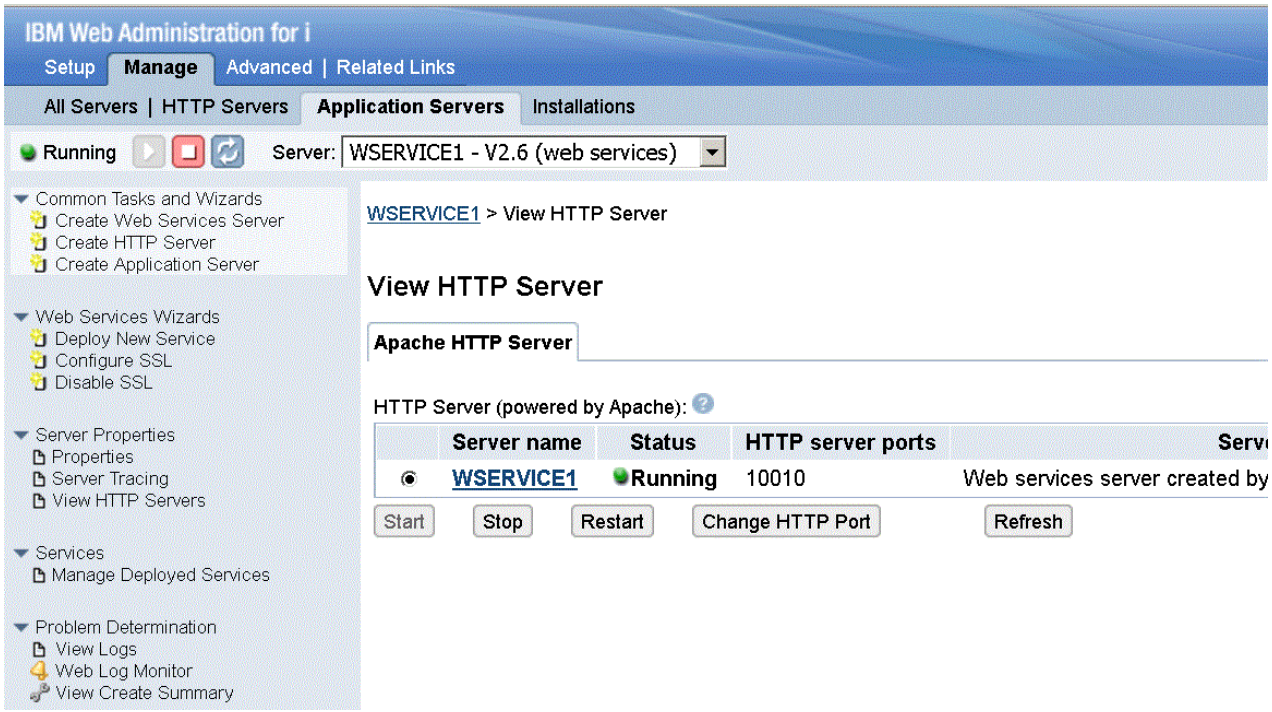


Figure 49: HTTP servers associated with server

If you click on a HTTP server link in the table, the view will change to the HTTP server view, where you can set and manage the HTTP server. This is shown in Figure 50 on page 92.

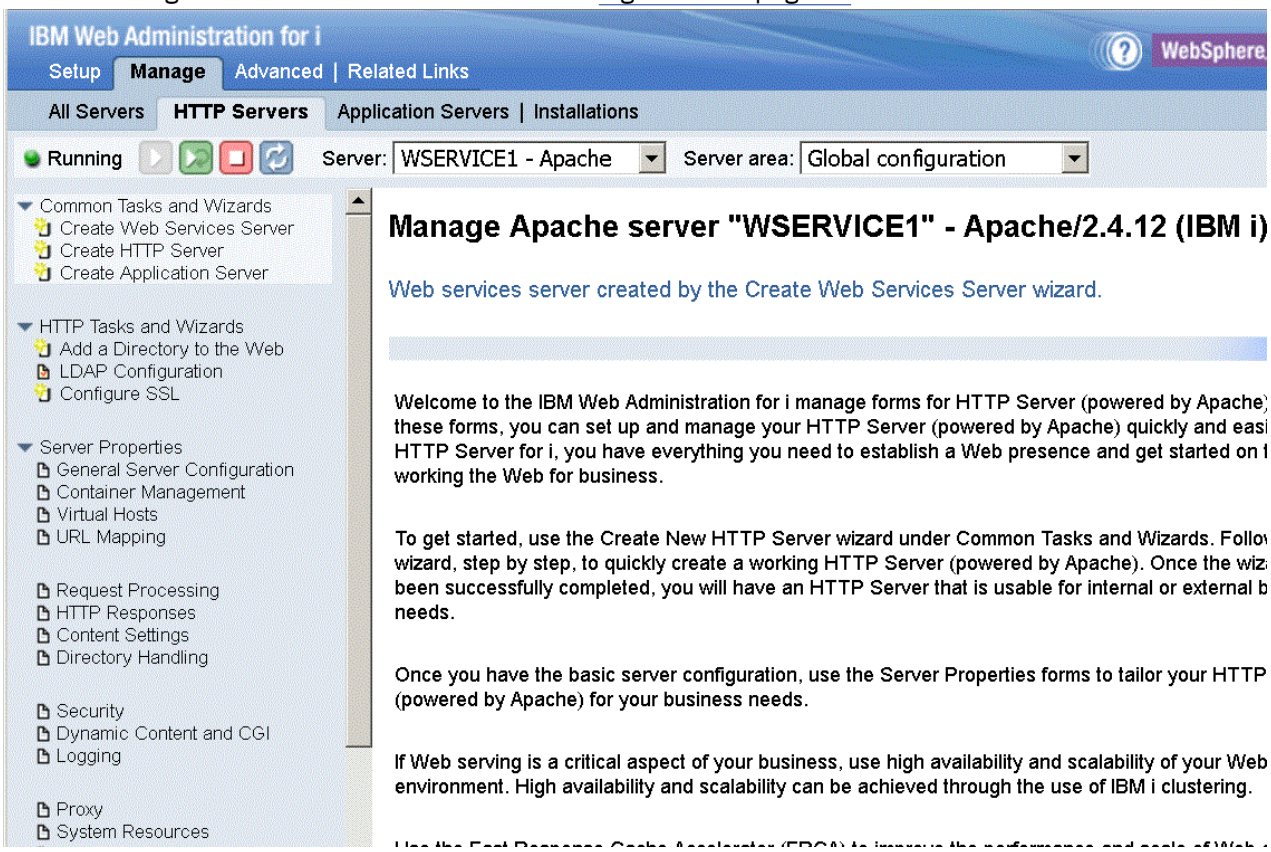


Figure 50: HTTP server view

Notice that the navigation bar has changed to reflect options relating to the management of the HTTP server.

Managing web services

The next group of navigation links relate to the management of web services. The arrow shown in Figure 51 on page 93 points to the location of links relating to web services.

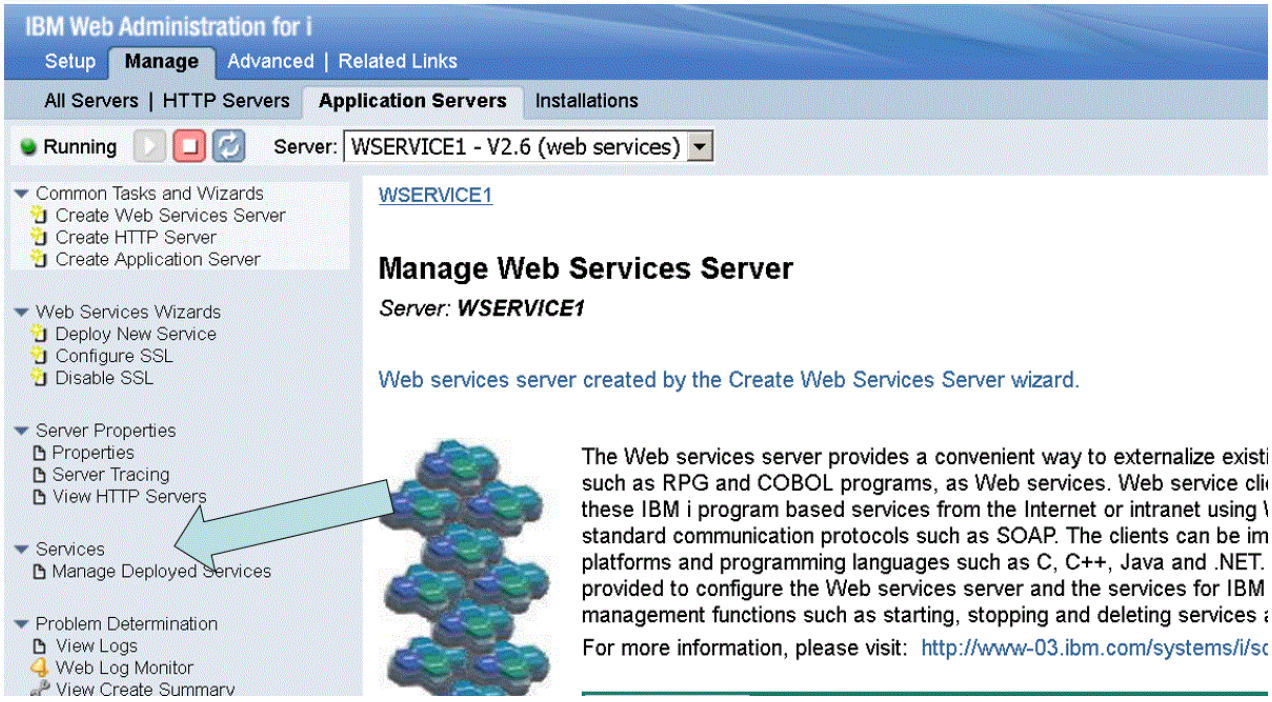


Figure 51: Manage server panel - manage services

Managing deployed services

If you click on the **Manage Deployed Services** link, you will get the panel shown in Figure 52 on page 94. The panel shows all the deployed web services.

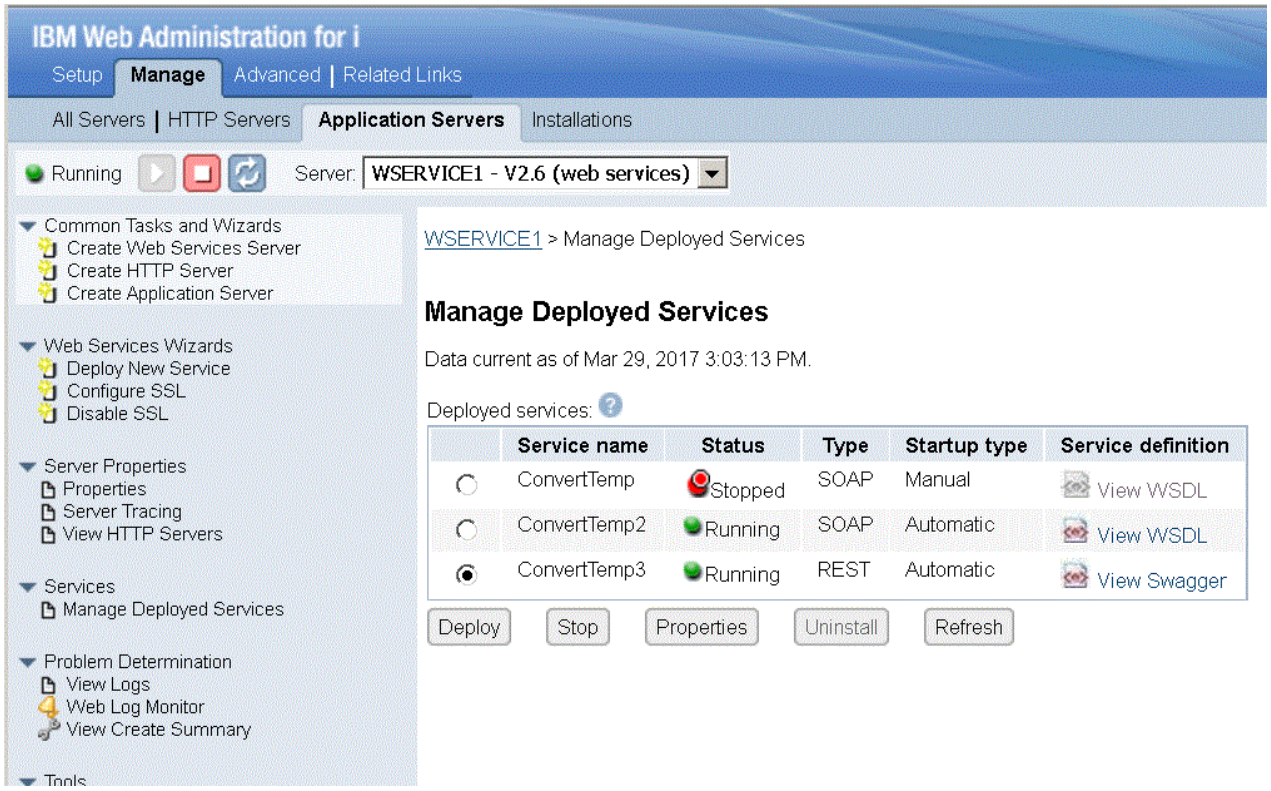


Figure 52: Manage server panel - manage services

The table of deployed web services includes the name the service, the status of the service (running or stopped), the type of the service (SOAP or REST), the startup type of the service - whether the service starts when the server is started (automatic) or whether the service needs to be explicitly started (manual), and a link to the service definition (WSDL or Swagger) for the service. The service definition link is clickable only if the web service is active.

Underneath the table are buttons:

- **Deploy**: Brings up **Deploy New Service** wizard.
- **Start**: Starts an inactive web service. If the service is active, the button will be disabled.
- **Properties**: Brings up a panel that allows you to see various web service properties. You will also be able to modify some options.
- **Uninstall**: Allows you to uninstall a web service. If the web service is active, the button will be disabled.
- **Refresh**: Refreshes the table.

So let us now look at the web service properties panel. If you select the radio button for the `ConvertTemp2` SOAP web service and click on the **Properties** button, the panel in Figure 53 on page 95 is shown.

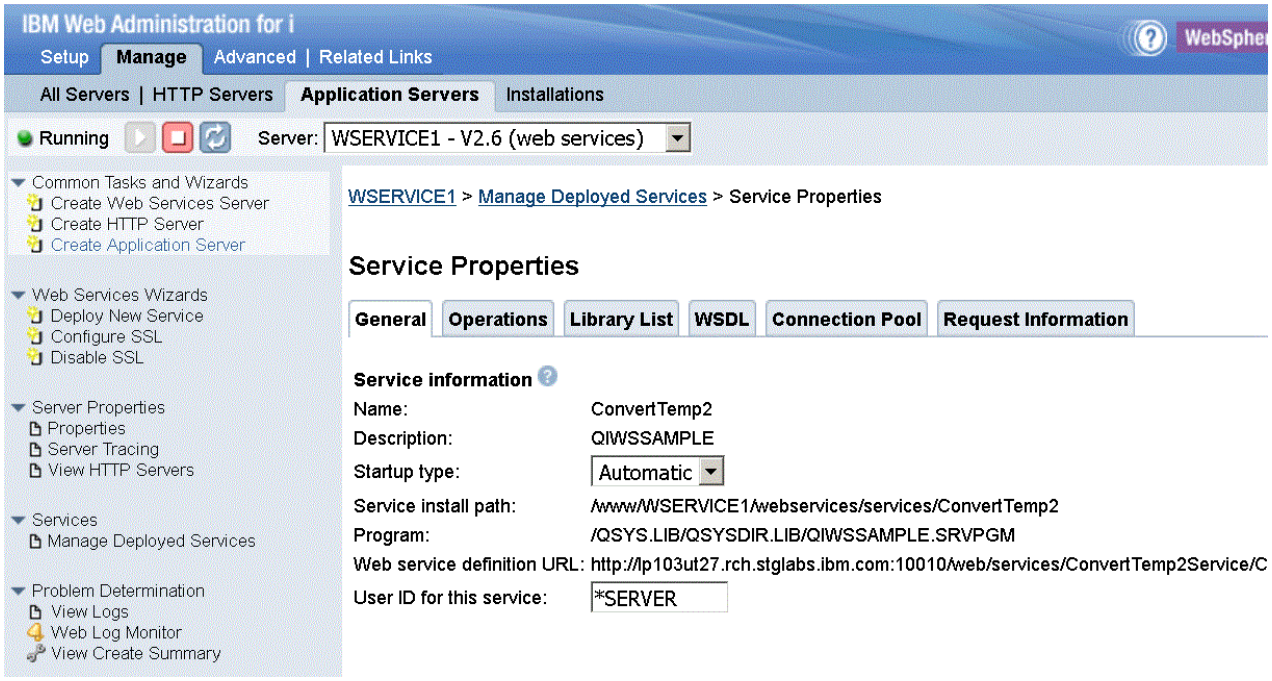


Figure 53: Properties panel - General

On the **General** tab of the property panel, there is information about the web service and related program object. Here is where you would modify the startup type and the user ID that the web service is run under.

Clicking on the **Operations** tab will show a list web service operations as shown in Figure 54 on page 95.

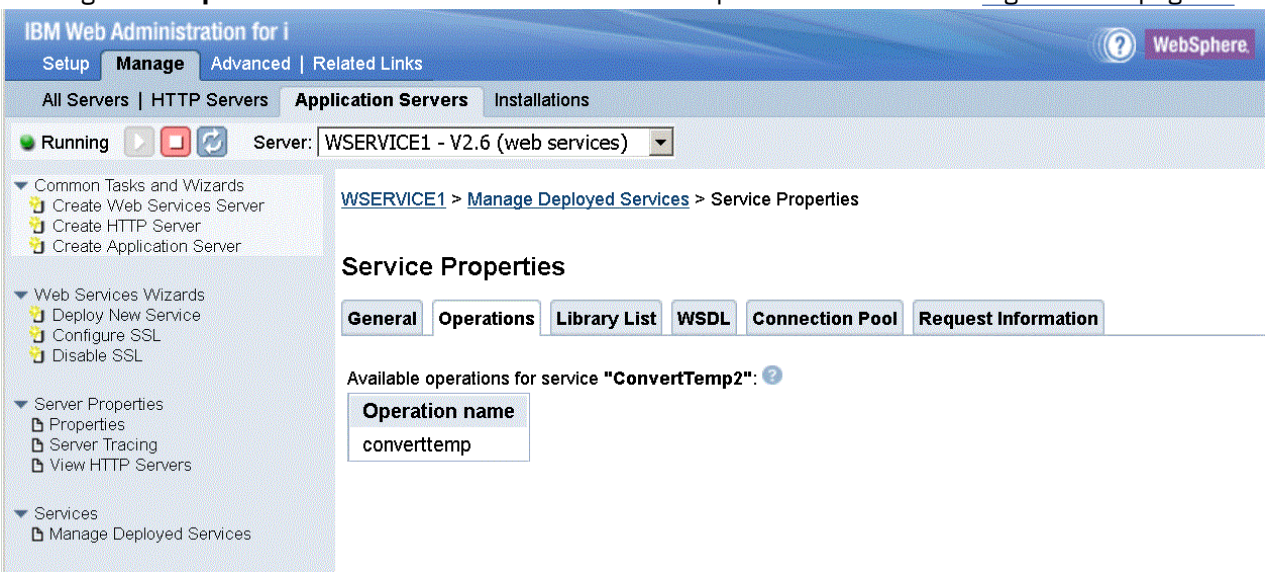


Figure 54: Properties panel - Operations (SOAP-only)

The **Operations** tab is for SOAP web services only. If this was a REST-based web service, you would see a **Methods** tab as shown in Figure 55 on page 96. The tab shows all the REST information for all the procedures associated with the REST resource.

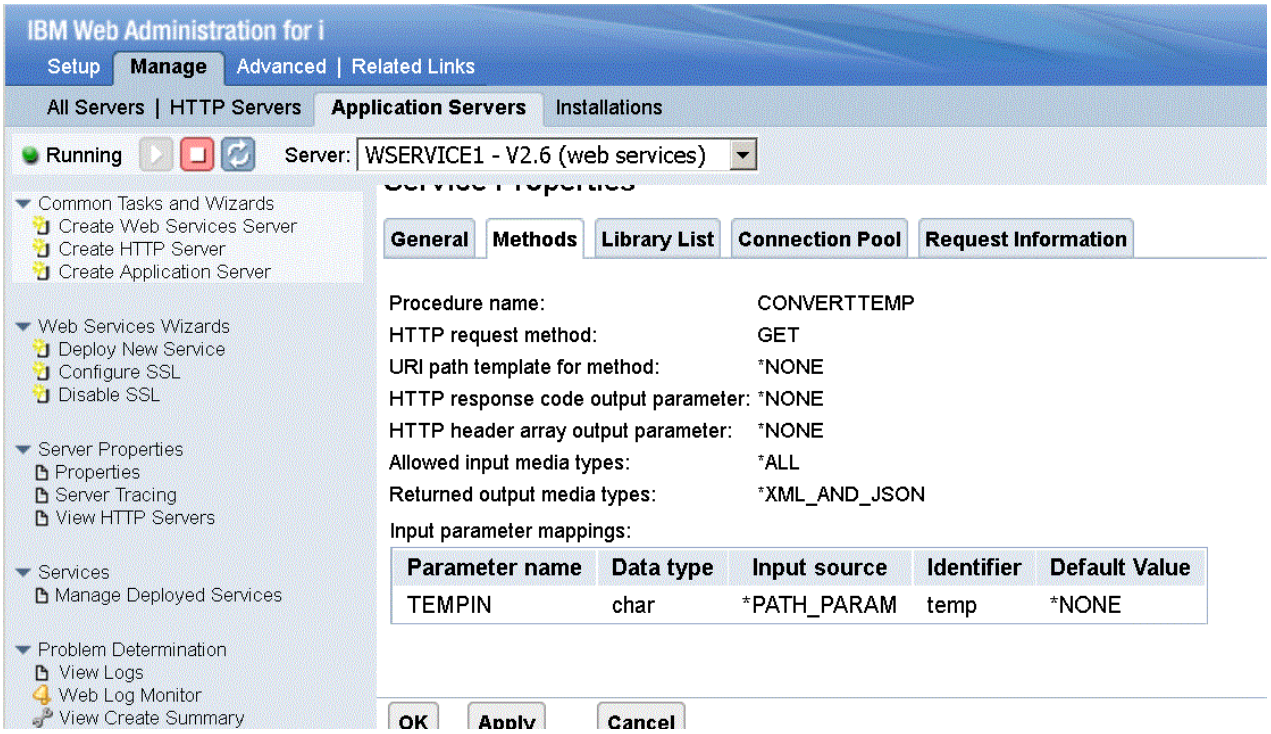


Figure 55: Properties panel - Methods (REST-only)

The **Library List** tab of the properties panel (shown in Figure 56 on page 96) allows you to display the libraries that will be added to the library list when the ILE program implementation for the web service is invoked. You can update, remove, and modify the list of library entries available to the program object.

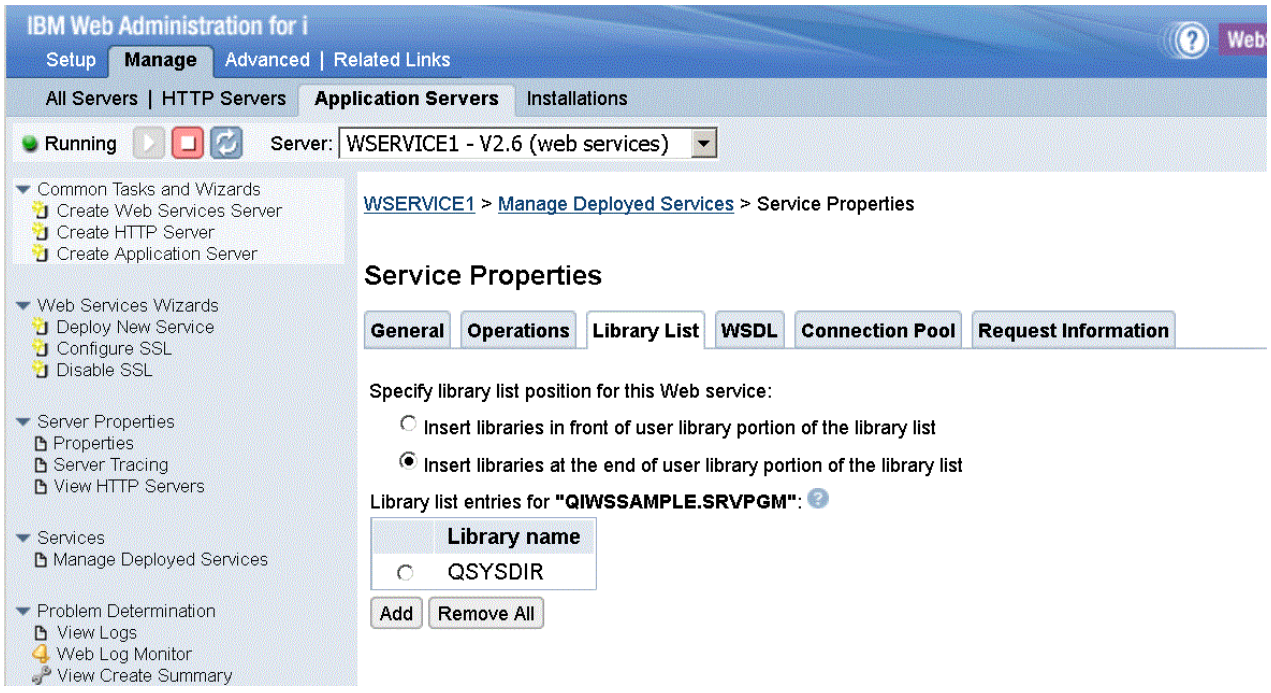


Figure 56: Properties panel - Library List

The **WSDL** property tab in Figure 57 on page 97 is only shown for SOAP-based web services. You can edit the file, but you cannot change namespaces or element names. You can also indicate whether the endpoint should be dynamically generated based on the incoming request URL, context root and web service name; or you can specify a URL to use. The URL you specify will automatically be inserted in the WSDL file as the SOAP address when the WSDL is requested by a client. Note that if you do choose to

specify a URL, you need to ensure that it eventually gets mapped to the actual URL that the server expects. You also have the ability to reset the WSDL file to its original form by clicking on the **Restore Static WSDL** button.

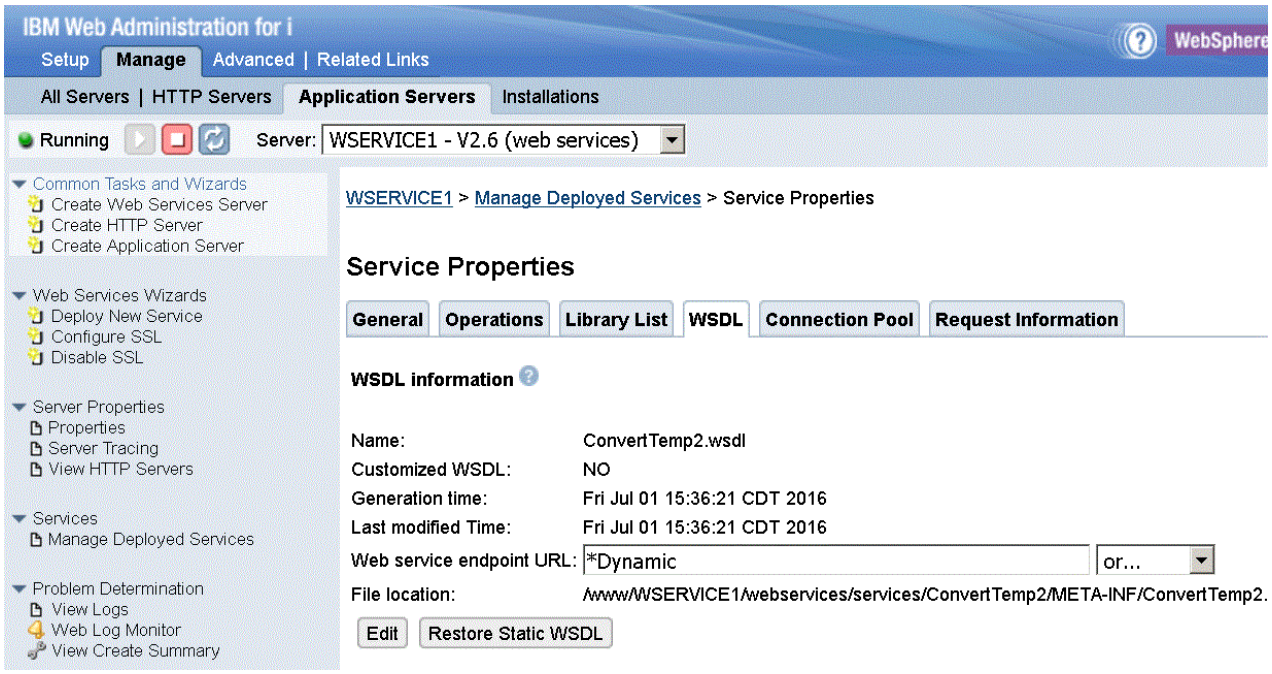


Figure 57: Properties panel - WSDL

The **WSDL** tab is for SOAP web services only. If this was a REST-based web service, you would see a **Swagger** tab as shown in Figure 58 on page 97. You can edit the file. You also have the ability to reset the Swagger file to its original form by clicking on the **Restore Static Swagger** button.

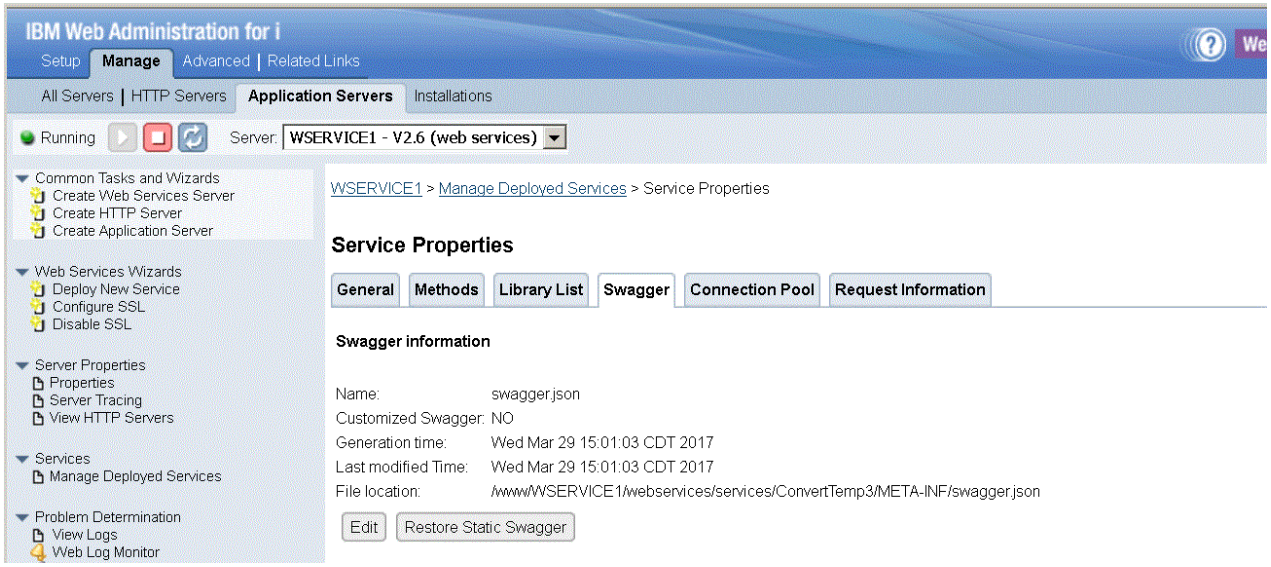


Figure 58: Properties panel - Swagger

The **Connection Pool** property tab in Figure 59 on page 98 allows you to view and update web service connection pool attributes. The connection pool contains a pool of connections to host server jobs that are used to handle web service requests.

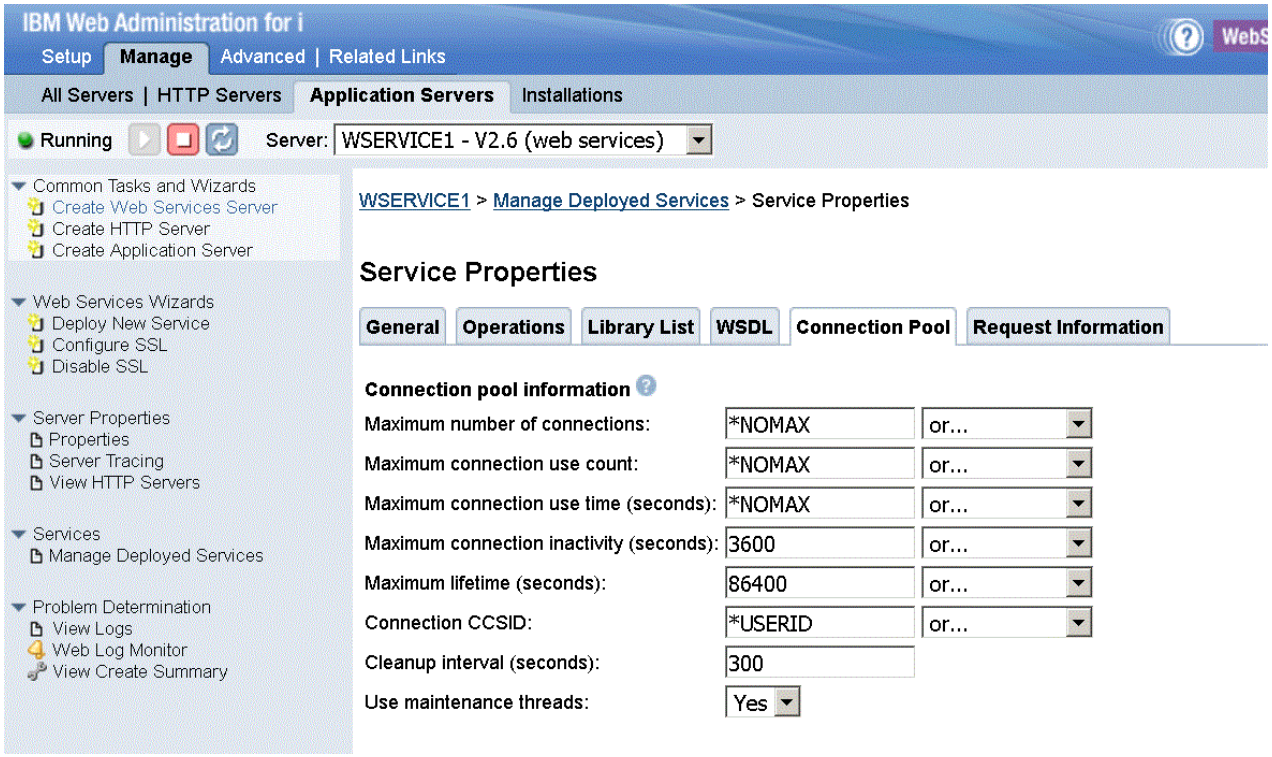


Figure 59: Properties panel - Connection Pool

More information on the connection pool attributes may be found in “Connection pools” on page 149.

The **Request Information** tab of the properties panel (shown in Figure 60 on page 99) allows you to display the transport and request information that is to be passed to the web service implementation code. You can update, remove, and modify the information that is to be passed to the program object. The information is passed to the program object in environment variables.

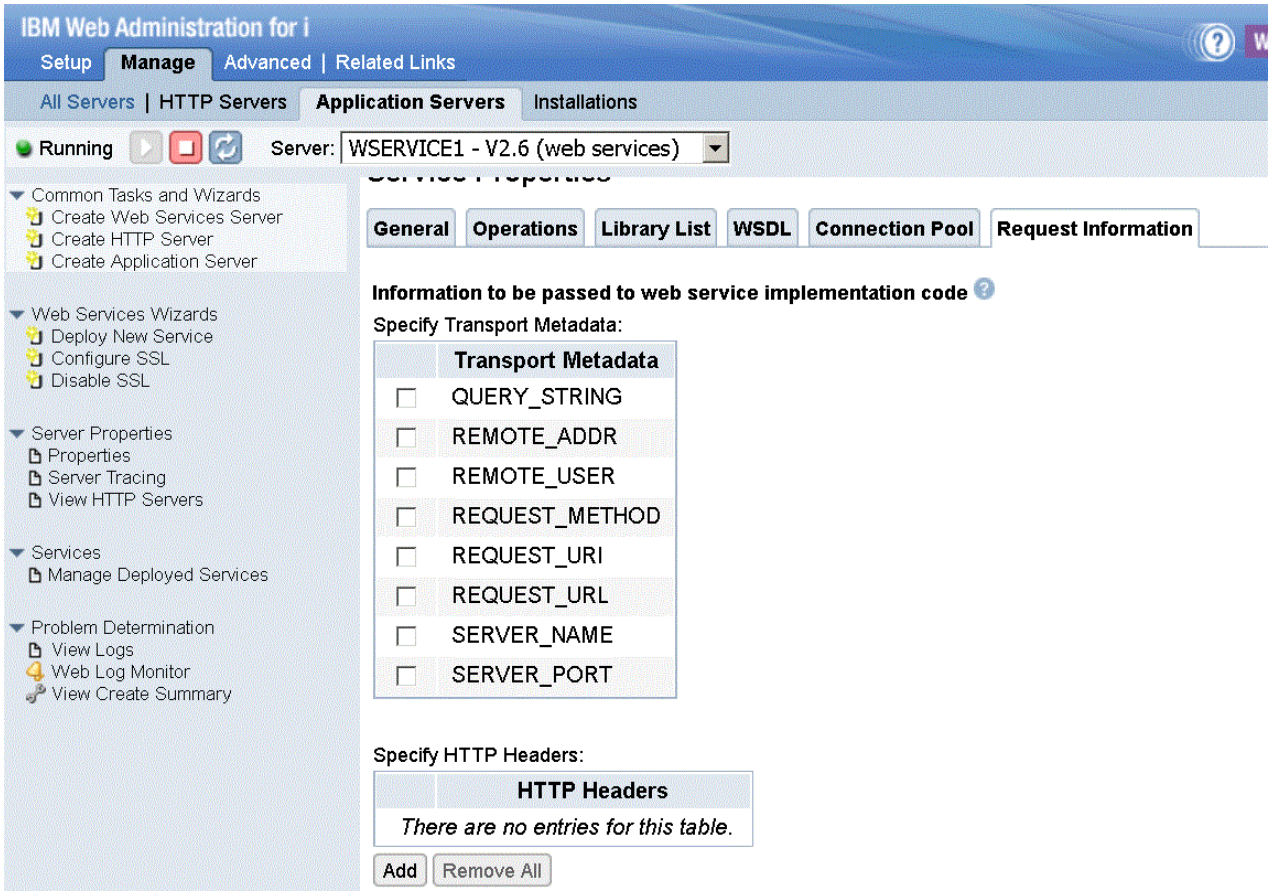


Figure 60: Properties panel - Request Information

There is two categories of information that can be passed: transport metadata associated with the client request and any HTTP headers in the client request. The supported transport metadata that may be passed to the web service implementation code is as follows:

QUERY_STRING
The query string that is contained in the request URL after the path. The value is not decoded by the server.

REMOTE_ADDR
The Internet Protocol (IP) address of the client or last proxy that sent the request.

REMOTE_USER
The login of the user making this request, if the user has been authenticated. If you want the REMOTE_USER to be passed to the web service implementation code, you would need to enable basic authentication in the associated HTTP server receiving the web service requests.

REQUEST_METHOD
The name of the HTTP method with which this request was made. For example, GET, POST, or PUT.

REQUEST_URI
The part of the client request's URL from the protocol name up to the query string in the first line of the HTTP request. The server does not decode the string.

REQUEST_URL
The URL the client used to make the request. The returned URL contains a protocol, server name, port number, and server path, but it does not include query string parameters.

SERVER_NAME
The host name of the server to which the request was sent. It is the value of the part before ":" in the Host header value, if any, or the resolved server name, or the server IP address.

SERVER_PORT
The port number to which the request was sent. It is the value of the part after ":" in the `Host` header value, if any, or the server port where the client connection was accepted on.

In addition to the transport metadata, HTTP headers in the client request may be passed to the web service implementation code. The environment variable name for HTTP headers is made up of the specified HTTP header prefixed with `HTTP_`, all uppercased. For example, if `Content-type` is specified, then the environment variable name would be `HTTP_CONTENT-TYPE`.

If there is no value associated with a transport metadata or HTTP header, the environment variable value will be set to the null string.

Problem determination

The next group of navigation links relate to problem determination. The arrow shown in points to the location of links relating to problem determination.

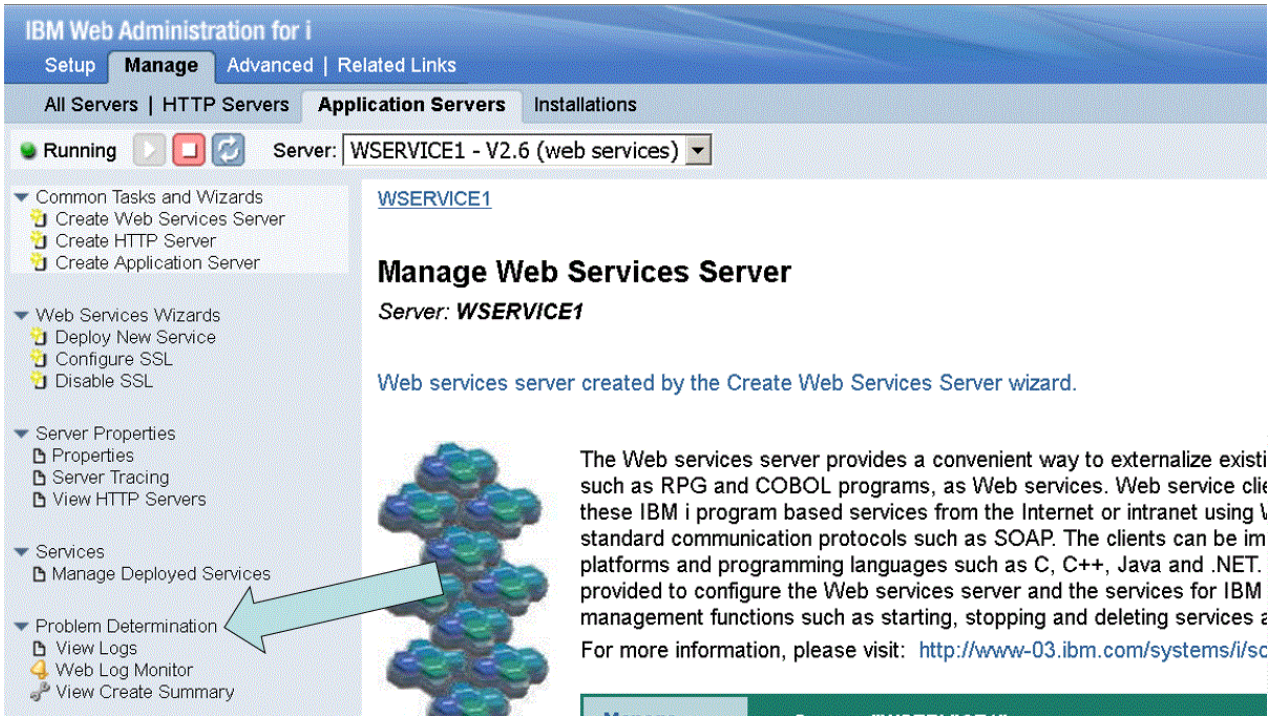


Figure 61: Manage server panel - problem determination

Viewing server logs

If you click on the **View Logs** link, you will get the panel shown in Figure 62 on page 101. The panel shows various log files associated with the server.

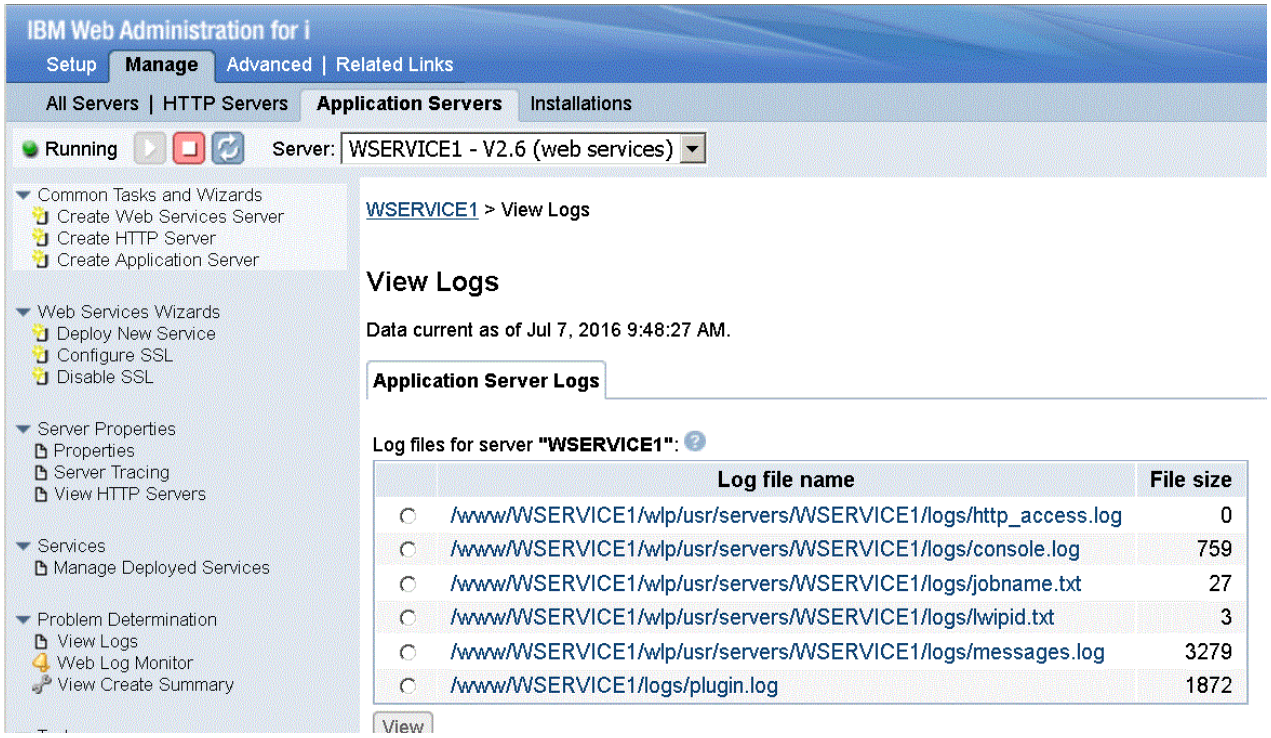


Figure 62: Problem determination - View logs

For more information on the various files and tracing, see “ Tracing” on page 139.

The web log monitor

If you click on the **Web Log Monitor** link, you will get the panel shown in Figure 63 on page 102. The Web Log Monitor inspects specified log files of any web related server. The log files are inspected for each keyword that is specified in the rule. When a match is found it automatically sends a notification to the system administrator via email or *QSYSOPR message queue. The Web Log Monitor interface allows you to customize the Web Log Monitor for the selected server, including the log file to be monitored, the notification text to be sent, the channel of sending notification (system message queue and/or email), the monitor interval, and the max number of notifications to be sent per hour.

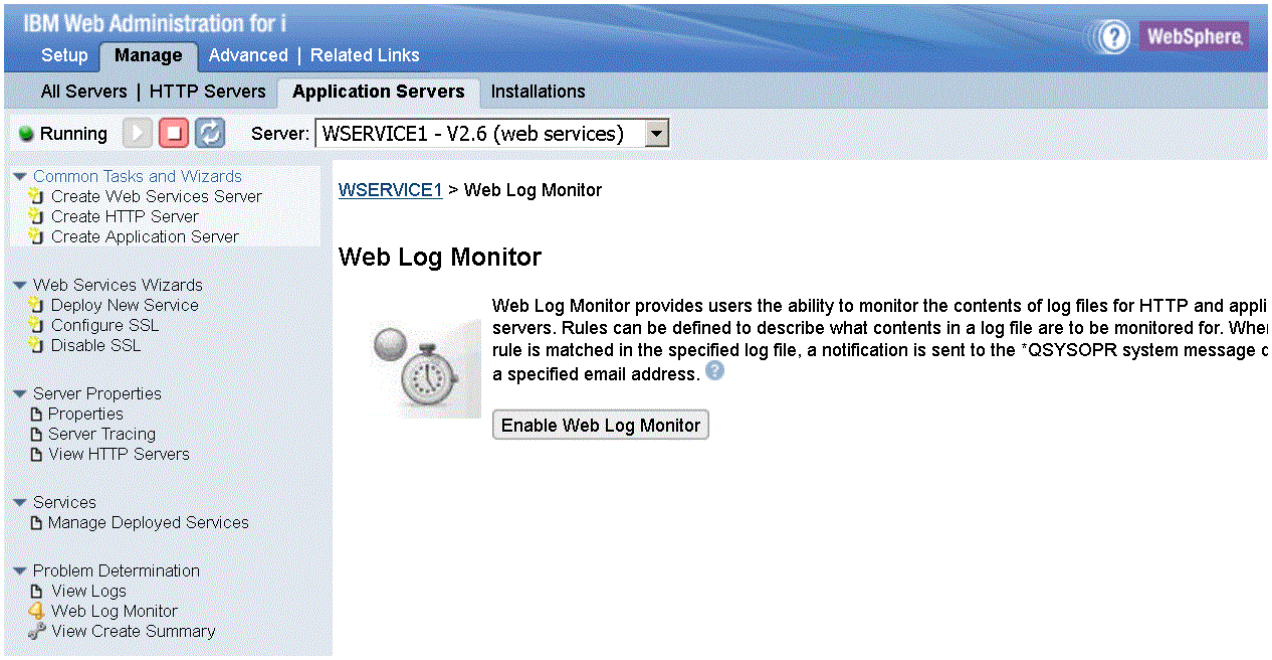


Figure 63: Problem determination - Web log monitor

Server creation summary

If you click on the **View Create Summary** link, you will get the panel shown in Figure 64 on page 103. What is shown is summary information about the integrated web services server and associated HTTP server that you can print out for your records.

Figure 64: Problem determination - Server creation summary

Chapter 7. Command line tools

There exists a collection of Qshell commands in support of the integrated web services server. The commands can be found in the `<install_dir>/bin` directory and are listed in table Table 8 on page 105.

Table 8: QShell commands in support of integrated web services server

Command	Description
createWebServicesServer.sh	Creates an integrated web services server.
deleteWebServicesServer.sh	Deletes an integrated web services server.
getWebServiceProperties.sh	Gets web service properties.
getWebServicesServerProperties.sh	Gets web services server properties.
installWebService.sh	Installs a web service.
listWebServices.sh	Lists all deployed web services in a web services server.
listWebServicesServers.sh	Lists all integrated web services servers.
restoreWebServices.sh	Restores web services from a save file.
restoreWebServicesServer.sh	Restores web services server from a save file.
saveWebServices.sh	Saves web services to a save file.
saveWebServicesServer.sh	Saves web services server to a save file.
setWebServiceProperties.sh	Sets web service properties.
setWebServicesServerProperties.sh	Sets web services server properties.
startWebService.sh	Starts a web service that is in a stopped state.
startWebServicesServer.sh	Starts an integrated web services server.
stopWebService.sh	Stops a web service that is in an active state.
stopWebServicesServer.sh	Stops an integrated web services server.
uninstallWebService.sh	Uninstalls a web service.

The commands must be run from within QShell. There are several ways to run QShell commands:

- Invoke the fully qualified path name of the command from within QShell (to enter the interactive shell session you would issue STRQSH CL command). For example,

```
<install_dir>/bin/startWebServicesServer.sh -server MyServer
```

- Invoke the script from the IBM i command line or from an IBM i CL program. To use this method, run the STRQSH CL command and specify the fully qualified path name of the script. For example:

```
STRQSH CMD('<install_dir>/bin/startWebServicesServer.sh -server MyServer')
```

The following sections gives more details of the supported commands.

createWebServicesServer.sh command

The **createWebServicesServer.sh** command creates an integrated web services server. An integrated web services server consists of an integrated application server running a web services engine and, optionally, an associated HTTP server.

Synopsis

```
createWebServicesServer.sh
    -server server-name  -startingPort starting-port
    [ -userid user-id ] [ -locationDirectory location-path ]
    [ -noHttp ]
    [ -defaultKeystore 'keystore' ] [ -defaultKeystorePassword 'password' ]
    [ -version '*DEFAULT|*CURRENT' ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server to be created.

-startingPort *starting-port*
Specifies the starting port number for generating ports required by the web services server. On creation, a port block of 10 ports is assumed to be available for the server, starting from the specified *starting-port*, although only four ports are used by default.

Optional arguments

-help
Displays a help message and exits.

-userid *user-id*
Specifies the user profile the web services server will run under. If not specified, QWSERVICE will be used, unless the caller of the command does not have authority to use the profile, in which case the caller's profile will be used. If a user ID is specified, the command will fail if the user profile of the caller of the command does not have *ALLOBJ or *USE authority to the specified user ID.

-locationDirectory *location-path*
Specifies the absolute path to the directory in which the web services server will be created. If not specified, the server will be created in the /www directory. The command will fail if the user profile of the caller of the command does not have authority to the specified directory.

-noHttp
Indicates that an associated HTTP server should not be created. If not specified, an associated HTTP server is created.

-defaultKeystore *keystore*
Specifies the path to the default keystore for the server. If specified, the value must be the path to the SYSTEM keystore, /QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB and the -defaultKeystorePassword parameter must be specified.

-defaultKeystorePassword *password*
Specifies the password to the keystore. If this parameter is specified, the parameter -defaultKeystore must also be specified.

-version
Indicating which version of web services server should be created. If a value of *DEFAULT is specified, the default version of a web services server for the installed operating system release will be created. If a value of *CURRENT is specified, the most current version of web services server for the installed release will be created. If not specified, a value of *DEFAULT will be used.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority or must have been given permission by an administrator using the Web Administration for i permission support to create integrated web services servers. To learn more about permissions, see "Permissions" on page 58.
2. If the default keystore is set, the user profile used to run the web services server must have *RX (read, execute) authority to all parts of the key store path. In addition, the server that is hosting the web service implementation code has to have the Remote Command Server and the Signon Server applications configures to use SSL as documented in the support page Configuring the SSL Telnet and Host Servers for Server Authentication for the First Time.

Example

The following command creates a web services server named `MyServer`:

```
createWebServicesServer.sh -server MyServer -startingPort 40001
```

deleteWebServicesServer.sh command

The **deleteWebServicesServer.sh** command deletes an integrated web services server.

Synopsis

```
deleteWebServicesServer.sh
    -server server-name  [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server to be deleted.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority or must have been given permission to delete the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command deletes a web services server named `MyServer`:

```
deleteWebServicesServer.sh -server MyServer
```

getWebServiceProperties.sh command

The **getWebServiceProperties.sh** command retrieves the properties of a web service.

Synopsis

```
getWebServiceProperties.sh
    -server server-name  -service service-name
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server in which the web service is deployed.

-service *service-name*
Specifies the web service for which properties will be displayed.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.

Example

The following command gets the properties for web service ConvertTemp that is deployed in the web services server named WSERVICE1:

```
getWebServiceProperties.sh -server WSERVICE1 -service ConvertTemp
```

The result of the command is as follows:

```
Name:                          ConvertTemp
Description:                   ConvertTemp
Startup type:                  Manual
Type:                          SOAP
Status:                        Stopped
Runtime user ID:               *SERVER
Install path:                  /www/WSERVICE1/webservices/services/ConvertTemp
Program object path:           /QSYS.LIB/QSYSDIR.LIB/QIWSSAMPLE.SRVPGM
PCML file path:
      /www/WSERVICE1/webservices/services/ConvertTemp/ConvertTemp.pcml
WSDL target namespace URI:     http://converttemp.wsbeans.iseries/
WSDL file path:
      /www/WSERVICE1/webservices/services/ConvertTemp/META-INF/ConvertTemp.wsdl
WSDL file generation time:     Wed Apr 09 09:44:37 CDT 2014
WSDL file modification time:   Thu Jun 09 18:11:52 CDT 2016
Operations:                    converttemp (Enabled); converttemp_XML (Enabled);
Library list:                  QSYSDIR;
Library list position:         *LAST
Connection pool properties
  Default CCSID:                       *USERID
  Use maintenance threads:             true
  Cleanup interval (seconds):          300
  Maximum number of connections:       *NOMAX
  Maximum inactivity time (seconds):   3600
  Maximum life time (seconds):         86400
  Maximum use count:                   *NOMAX
  Maximum use time (seconds):          *NOMAX
Transport metadata:
Transport headers:
```

getWebServicesServerProperties.sh command

The **getWebServicesServerProperties.sh** command retrieves the properties of a web services server.

Synopsis

```
getWebServicesServerProperties.sh
    -server server-name  [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server for which properties will be displayed.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.

Example

The following command gets the properties for the web services server named WSERVICE1:

```
getWebServicesServerProperties.sh -server WSERVICE1
```

The result of the command is as follows:

```
Instance path: /www/WSERVICE1
Application server: Integrated Application Server 8.5.5.9
Application server ports: 10000;
Subsystem: QHTTPSVR
Job name: WSERVICE1
Runtime user ID: QWSERVICE
JVM version: 1.6
JVM type: IBM Technology for Java VM
Web services runtime: Apache CXF 2.6
Web services install path: /www/WSERVICE1/webservices/services
Context root: /web/services
Java toolbox tracing: Disabled
Trace output file name:
     /www/WSERVICE1/wlp/usr/servers/WSERVICE1/logs/messages.log
HTTP server name: WSERVICE1
HTTP server ports: 10010;
```

installWebService.sh command

The **installWebService.sh** command deploys a web service to the specified web services server.

Synopsis

```
installWebService.sh
    -server server-name -programObject program-object
    [ -service service-name ] [ -pcml pcml-file ]
    [ -userid '*SERVER|*AUTHENTICATED|userid' ]
    [ -detectFieldLengths ] [ -serviceType '*SOAP11|*SOAP12|*REST' ]
    [ -host host-server ]
    [ -targetNamespace target-namespace ]
    [ -parameterUsage 'parameter-list' ] [ -propertiesFile 'property-file' ]
    [ -libraryList library-list ] [ -libraryListPosition *FIRST|*LAST ]
    [ -transportMetadata *NONE|metadata-list ]
    [ -useParamNameAsElementName ]
    [ -transportHeaders *NONE|header-list ]
    [ -soapOptions option-list ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server in which the web service will be deployed.

-programObject *program-object*
Specifies the integrated file system path to the ILE program or service program that will be deployed as a web service.

Optional arguments

-service *service-name*
Specifies the name of the web service. If not specified, the program object name will be used.

-pcml *pcml-file*
Specifies the path to the PCML file. If not specified, the program object must contain the PCML data.

-userid *SERVER | *AUTHENTICATED | *userid*
Specifies the user profile the web service will run under. If not specified, the web service will run under the web services server user profile. A value of *SERVER will ensure the web service runs under the same user profile as the web services server. the same user profile as the web services server. A value of *AUTHENTICATED will ensure the web service runs under an authenticated user profile.

Note: The web service server's user profile needs to have *USE authority to the user profile(s) specified.

-detectFieldLengths
Indicates that field lengths for array and character fields will be detected. Specifying a field length for output arrays ensures only those elements that are set in the array are returned. Specifying a length field for large character fields improves the processing of the character field. If not specified, length field processing is not performed.

-serviceType *SOAP11|*SOAP12|*REST
Specifies the type of service to be installed. A value of *SOAP11 or *SOAP12 indicates that the program object should be installed as a SOAP 1.1 or SOAP 1.2 service, respectively. A value of *REST indicates that the program object should be installed as a REST service. If not specified the default value of *SOAP11 is used.

-host *host-server*
Specifies the host name or internet protocol (IP) address of the server hosting the web service implementation code (i.e. ILE program object). Specify `localhost` if the web service implementation code resides on the same server as the integrated web services server. A secure channel between the server and the server hosting the web service implementation code may be requested by appending `:SECURE` to the host value. For example, `-host iserver.com:SECURE`.

Note: This option is not supported on version 1.3 and 1.5 of the server.

-parameterUsage *parameter-list*
Specifies a colon delimited list containing parameter usage values corresponding to the procedures or program to be deployed. For each program or procedure you will need to specify the procedure or program name, followed by a colon, followed by a comma delimited list of usage descriptors ('i' for input, 'o' for output, or 'io' for inputoutput) for each parameter. For example, if a service program contains two procedures, PROC1 with two parameters and PROC2 with three parameters, then a possible value would be:

```
-parameterUsage PROC1:i,o:PROC2:i,i,io
```

Note that only those procedures listed will be externalized. This parameter is optional. If not specified the default is to use the parameter usage values specified in the PCML associated with the program object. Note that if the `-propertiesFile` parameter is specified, any parameter usage property values in the file will override what is specified in `-parameterUsage`.

-propertiesFile *property-file*
Specifies an absolute path to file that contains various web service properties. The property file follows the rules of Java properties file. For example, to indicate a line is a comment, you would start the line with a pound sign ('#'). REST properties such as the URI path template, HTTP request method, and the content type of the response may be specified in the file for each entry point in the program object. An example of the contents of a property file for procedure CONVERTTEMP is as follows:

```
uri.path.template=/temperature/{i}
CONVERTTEMP.uri.path.template=
CONVERTTEMP.wrap.input.parameters=false
CONVERTTEMP.wrap.output.parameter=true
CONVERTTEMP.http.request.method=GET
CONVERTTEMP.consumes=*/*
CONVERTTEMP.produces=application/xml, application/json
CONVERTTEMP.response.code.parameter=
CONVERTTEMP.http.headers.parameter=
CONVERTTEMP.TEMPIN.usage=input
CONVERTTEMP.TEMPIN.pathparam=i
CONVERTTEMP.TEMPOUT.usage=output
```

Note: When specifying identifiers in the property file, case matters.

The following properties have defaults and do not need to be specified: `uri.path.template` ('/'), `wrap.input.parameters` (false), `wrap.output.parameter` (true), `http.request.method` (GET), `consumes` ('*/*'), and `produces` ('application/xml, application/json'). In addition,

any parameter usage property found in the file overrides any value specified in the -parameterUsage command option.

If you have multiple procedures you would use the same property file, the beginning of the property name would change so that it reflects the procedure name. For example, if you have procedures PROC1 and PROC2, then the property file would contain the following properties:

```
uri.path.template=/temperature/{i}

PROC1.uri.path.template=
PROC1.wrap.input.parameters=false
PROC1.wrap.output.parameter=true
.
.
.
PROC2.uri.path.template=
PROC2.wrap.input.parameters=false
PROC2.wrap.output.parameter=true
```

The following table shows how to specify a property that will inject input data from the various input sources into parameter TEMPIN:

Table 9: Parameter injection via property file.

Input source	**Example**
path parameter	CONVERTTEMP.TEMPIN.pathparam=i
query parameter	CONVERTTEMP.TEMPIN.queryparam=i
form parameter	CONVERTTEMP.TEMPIN.formparam=i
matrix parameter	CONVERTTEMP.TEMPIN.matrixparam=i
HTTP headers	CONVERTTEMP.TEMPIN.headerparam=i
cookie	CONVERTTEMP.TEMPIN.cookieparam=i
Note: For all the types of parameters except path parameters, you are able to specify a default value. For example, to indicate TEMPIN should have a default value of 32, you would specify the following in the property file: CONVERTTEMP.TEMPIN.default=32.	

-libraryList *library-list*
Specifies a list of libraries that will be added to the library list prior to invoking the web service. Each library in the list must be delimited by a colon.

-libraryListPosition *FIRST | *LAST
Specifies the position in the user portion of the job library list where the list of libraries specified in -libraryList will be placed. A value of *FIRST inserts the libraries at the beginning of the user portion of the library list. A value of *LAST inserts the libraries at the end of the user portion of the library list. If not specified, a value of *LAST will be used.

-transportMetadata *NONE | *metadata-list*
Specifies what transport metadata to pass to the web service implementation code. Transport metadata is passed as environment variables. The default value of *NONE indicates that no metadata is set. A colon delimited string of metadata can also be specified. Supported value(s): REMOTE_ADDR, REMOTE_USER, REQUEST_METHOD, REQUEST_URL, REQUEST_URI, QUERY_STRING, SERVER_NAME and SERVER_PORT.

Note: REMOTE_ADDR is the only supported value for servers running version 1.3 or 1.5 and any other values that are set will be ignored.

-transportHeaders *NONE | *header-list*
Specifies what transport headers (e.g. HTTP headers) to pass to the web service implementation code. Transport headers are passed as environment variables. The environment variable name for HTTP headers is made up of the specified HTTP header prefixed with 'HTTP_', all uppercased. For

example, if 'Content-type' is specified, then the environment variable name would be 'HTTP_CONTENT-TYPE'. The default value of *NONE indicates that no transport headers should be set. A colon delimited string of transport headers can also be specified.

-soapOptions *option-list*
Specifies a list of SOAP options. Each option must be delimited by a colon. The following options may be specified:

- nooptional - Indicates that elements in generated WSDL should not be designated as optional. If not specified, elements are defined as optional. This parameter is ignored for version 1.3 of web services engine.
- nonillable - Indicates that elements in generated WSDL should not be designated as nillable. If not specified, elements are defined as nillable. This parameter is ignored for version 1.3 of web services engine.
- qualified - Indicates that elements should be namespace qualified. If not specified, element names will not be namespace qualified.
- soapaction - Indicates that the SOAPAction HTTP header must be set on SOAP requests to the service. If not specified, the SOAPAction HTTP header must not be set.
- addunderscore - Indicates that element names will be prefixed with an underscore character. If not specified, element names in WSDL will not start with the underscore character.
- noxmlops - Indicates whether operations that return an XML document as a string for SOAP web services should not be generated. The default value is for operations to be generated.

This parameter is optional.

-useParamNameAsElementName
Indicates whether the element name should be matched to the parameter name for the wrapper element in XML payloads. If specified, the wrapper element name will match the parameter name. If not specified, the wrapper element name will match the structure name. This parameter is ignored for SOAP services. This parameter is optional.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.
2. In order to run the web service implementation code under an authenticated user ID, you may need to redeploy the web service since the support to run under an authenticated user ID was introduced in 2018.
3. If the web service implementation code is hosted on a remote system and the connection between the web services server and the web service is secure, the default key store for the web services server must be set.
4. The various options are discussed in more detail in “The deploy new service wizard” on page 66.

Example

The following command installs the ILE service program QIWSSAMPLE as a web service named ConvertTemp2 in the web services server named MyServer:

```
installWebService.sh -server MyServer
    -programObject /qsys.lib/qsysdir.lib/qiwssample.srvpgm
    -service ConvertTemp2
```

listWebServices.sh command

The **listWebServices.sh** command displays the web services that are deployed in a specified web services server.

Synopsis

```
listWebServices.sh
    -server server-name [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server for which deployed web services will be listed.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command lists the web services that are deployed in the web services server named MyServer:

```
listWebServices.sh -server MyServer
```

The result of the command is as follows:

```
ConvertTemp (Stopped)
ConvertTemp2 (Stopped)
```

listWebServicesServers.sh command

The **listWebServicesServers.sh** command displays a list of integrated web services servers.

Synopsis

```
listWebServicesServers.sh [ -printErrorDetails ] [ -help ]
```

Arguments

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority or must have permission to use the command. To learn more about permissions, see “Permissions” on page 58.

Example

The following command lists all the integrated web services servers on the system:

```
listWebServicesServers.sh
```

The result of the command is as follows:

```
MyServer (Stopped)
ryaniws (Stopped)
WSDEV (Stopped)
WSERVICE (Stopped)
WSERVICE1 (Stopped)
WSERVICE2 (Running)
WSERVICE7 (Stopped)
```

restoreWebServices.sh command

The **restoreWebServices.sh** command restores a copy of one or more web services into an integrated web services server.

Synopsis

```
restoreWebServices.sh
    -server server-name  -saveFile save-file
    -fromServerDirectory server-directory
    [ -serviceList *ALL|service-list ]
    [ -migrationOptions 'option-list' ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server in which the web services will be restored.

-saveFile *service-file*
Specifies the absolute path name of the save file from which the web services will be restored.

-fromServerDirectory *server-directory*
Specifies the absolute path name of the server that hosted the web services saved in the save file.

Optional arguments

-serviceList *service-list*
Specifies a colon-delimited list of web services to be restored. A value of *ALL means all web services will be restored. The default value is *ALL.

-migrationOptions *option-list*
Specifies a colon-delimited string listing various options when migrating services from version 1.3 or 1.5 to a more recent version of the server. Possible values:

- `addunderscore` - add underscores to WSDL element names (default is to not prefix element names with underscores)
- `soap12` - use SOAP 1.2 (default is SOAP 1.1)
- `addxmlops` - add _XML operations (default is to not generate _XML operations)
- `version13` - indicates services are being migrated from version 1.3 of the server (default is 1.5)

  This parameter will be ignored if the services to be restored did not reside in a 1.3 or 1.5 server or the version of the server in which the services will be restored is not a more recent version of the server.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have `*ALLOBJ` authority to use the command.
2. Previously, if services from a version 1.3 or 1.5 of the server were restored to a more current version of the server, the WSDL file had minor changes. In addition, the path to the web service was different. Thus, the WSDL needed to be sent out to all users of the web service. A recent update to the **restoreWebServices.sh** now eliminates WSDL changes, resulting in no disruption to client applications. The path to the service is still different, but that may be mitigated by the HTTP server's URL rewriting support. For more information, see “Simplifying web service URIs” on page 131.

Example

The following command restores web services `WS1` and `WS2` to a server named `MyServer2` from a save file named `MYSAVF` in library `MYLIB`. The web services were saved from a server with a absolute path name of `/www/MyServer`:

```
restoreWebServices.sh -server MyServer2 -serviceList WS2:WS1
   -fromServerDirectory /www/MyServer
   -saveFile /qsys.lib/mylib.lib/mysavf.file
```

restoreWebServicesServer.sh command

The **restoreWebServicesServer.sh** command restores a copy of an integrated web services server.

Synopsis

```
restoreWebServicesServer.sh
    -fromServerDirectory server-directory
    -saveFile save-file
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-fromServerDirectory *server-directory*
Specifies the absolute path name of the server that hosted the web services saved in the save file.

-saveFile *service-file*
Specifies the absolute path name of the save file from which the web services will be restored.

Optional arguments

-locationDirectory *location-directory*
Specifies the absolute path to the directory in which the web services server will be restored. If not specified, the server will be restored in the /www directory. The command will fail if the user profile of the caller of the command does not have authority to the specified directory.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority to use the command.

Example

The following command restores a server named MyServer2 from a save file named MYSAVF in library MYLIB. The web services server was saved from a server with a absolute path name of /www/MyServer2:

```
restoreWebServices.sh
   -fromServerDirectory /www/MyServer2
   -saveFile /qsys.lib/mylib.lib/mysavf.file
```

saveWebServices.sh command

The **saveWebServices.sh** command saves a copy of one or more web services.

Synopsis

```
saveWebServices.sh
    -server server-name  -saveFile save-file
    [ -serviceList *ALL|service-list ]
    [ -targetRelease *CURRENT|*PREVIOUS ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server containing the services to be saved.

-saveFile *service-file*
Specifies the absolute path name of the save file used to save the web services. If the save file doesn't exist it will be created automatically.

Optional arguments

-serviceList *service-list*
Specifies a colon-delimited list of web services to be saved. A value of *ALL means all web services will be saved. The default value is *ALL.

-targetRelease *CURRENT | *PREVIOUS
Specifies the release level of the operating system on which you intend to use the object(s) being saved. A value of *CURRENT indicates that the object is to be restored to, and used on, the release of the operating system currently running on your system. The object can also be restored to a system with any subsequent release of the operating system installed. A value of *PREVIOUS indicates that the object is to be restored to the previous release with modification level 0 of the operating system.

The object can also be restored to a system with any subsequent release of the operating system installed. The default value is *CURRENT.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority to use the command.

Example

The following command saves web services WS1 and WS2 deployed in a server named MyServer to a save file named MYSAVF in library MYLIB:

```
saveWebServices.sh -server MyServer -serviceList WS2:WS1
   -saveFile /qsys.lib/mylib.lib/mysavf.file
```

saveWebServicesServer.sh command

The **saveWebServicesServer.sh** command saves a copy of a web services server to a save file.

Synopsis

```
saveWebServicesServer.sh
    -server server-name  -saveFile save-file
    [ -targetRelease *CURRENT|*PREVIOUS ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server to be saved.

-saveFile *service-file*
Specifies the absolute path name of the save file used to save the web service server. If the save file doesn't exist it will be created automatically.

Optional arguments

-targetRelease *CURRENT | *PREVIOUS
Specifies the release level of the operating system on which you intend to use the object(s) being saved. A value of *CURRENT indicates that the object is to be restored to, and used on, the release of the operating system currently running on your system. The object can also be restored to a system with any subsequent release of the operating system installed. A value of *PREVIOUS indicates that the object is to be restored to the previous release with modification level 0 of the operating system. The object can also be restored to a system with any subsequent release of the operating system installed. The default value is *CURRENT.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. A user must have *ALLOBJ authority to use the command.

Example

The following command saves a server named MyServer to a save file named MYSAVF in library MYLIB:

```
saveWebServicesServer.sh -server MyServer
   -saveFile /qsys.lib/mylib.lib/mysavf.file
```

setWebServiceProperties.sh command

The **setWebServiceProperties.sh** command sets various properties for a specified web service.

Synopsis

```
setWebServiceProperties.sh
    -server server-name -service service-name
    [ -programObject program-object ]
    [ -userid '*SERVER|*AUTHENTICATED|userid' ]
    [ -host host-server ] [ -resetWSDL ]
    [ -disableNillableWSDLElements] [ -disableOptionalWSDLElements ]
    [ -addUnderscoreToWSDLElementNames ]
    [ -libraryList library-list ] [ -libraryListPosition  *FIRST|*LAST ]
    [ -autoStartup true|false ] [ -connPoolCCSID *USERID|ccsid ]
    [ -connPoolCleanupInterval cleanup-interval ]
    [ -connPoolMaxConnections *NOMAX|max-connections ]
    [ -connPoolMaxInactivity *NOMAX|max-inactivity ]
    [ -connPoolMaxLifetime *NOMAX|max-lifetime ]
    [ -connPoolMaxUseCount *NOMAX|max-use-count ]
    [ -connPoolMaxUseTime *NOMAX|max-use-time ]
    [ -connPoolFillCount fill-count ]
    [ -connPoolUseThreads true|false ]
    [ -transportMetadata *NONE|metadata-list ]
    [ -transportHeaders *NONE|header-list ]
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server containing the service to be modified.

-service *service-name*
Specifies the web service for which the properties is to be retrieved.

Optional arguments

-programObject *program-object*
Specifies the integrated file system path to the ILE program or service program that implements the web service.

Note: This is a new feature and if you are on IBM i 7.3 or previous releases you will need to ensure the integrated web services PTF is applied (7.1-SI68785, 7.2-SI68745, 7.3-SI68746) and you may need to redeploy the web service for this option to take affect.

-userid *SERVER | *AUTHENTICATED | *userid*
Specifies the user profile the web service will run under. A value of *SERVER will ensure the web service runs under the same user profile as the web services server. A value of *AUTHENTICATED will ensure the web service runs under an authenticated user profile.

Note: The web service server's user profile needs to have *USE authority to the user profile(s) specified.

-host *host-server*
Specifies the host name or internet protocol (IP) address of the server hosting the web service implementation code (i.e. ILE program object). Specify `localhost` if the web service implementation code resides on the same server as the integrated web services server. A secure channel between the server and the server hosting the web service implementation code may be requested by appending `:SECURE` to the host value. For example, `-host iserver.com:SECURE`.

Note: This option is not supported on version 1.3 and 1.5 of the server.

-resetWSDL
Generates a new WSDL file for the specified web service. The new WSDL file will be the same as the one generated when the web service was first deployed. Any changes that have been made to the existing WSDL file will be lost.

-disableNillableWSDLElements
Indicates that the WSDL file will not include nillable elements. If not specified, elements are defined as nillable. This parameter is ignored if `-resetWSDL` is not specified or if the server is running version 1.3 of web services engine.

-disableOptionalWSDLElements
Indicates that the WSDL file will not include optional elements. If not specified, elements are defined as optional. This parameter is ignored if `-resetWSDL` is not specified or if the server is running version 1.3 of web services engine.

-addUnderscoreToWSDLElementNames
Indicates that element names in the WSDL file will start with underscore, as was always done in version 1.3 of the web services server. If not specified, element names in WSDL will not start with the underscore character. This parameter is ignored if `-resetWSDL` is not specified or if the server is running version 1.3 of web services engine.

-libraryList *library-list*
Specifies a list of libraries that will be added to the library list prior to invoking the web service. Each library in the list must be delimited by a colon. A value of `*NONE` means no libraries will be added to the library list.

-libraryListPosition *FIRST | *LAST
Specifies the position in the user portion of the job library list where the list of libraries specified in `-libraryList` will be placed. A value of `*FIRST` inserts the libraries at the beginning of the user portion of the library list. A value of `*LAST` inserts the libraries at the end of the user portion of the library list.

-autoStartup true | false
Specifies whether or not the web service should automatically be started at server startup time. A value of `true` means the web service should be set to autostart. A value of `false` means the web service should not be set to autostart.

-connPoolCCSID *USERID | *ccsid*
Specifies the coded character set identifier (CCSID) used when converting character data that is sent to the web service implementation code. Data received from the web service implementation code is assumed to be in the CCSID that is specified. The default value of `*USERID` indicates that the CCSID of user profile used to run the web service is used.

-connPoolCleanupInterval *cleanup-interval*
Specifies the time interval, in seconds, for how often the connection pool maintenance daemon is run. The default value is 300 seconds (5 minutes).

-connPoolMaxConnections *NOMAX | *max-connections*
Specifies the maximum number of connections for the connection pool. The default value of `*NOMAX` indicates no limit to the number of connections.

-connPoolMaxInactivity *NOMAX | *max-inactivity*
Specifies the maximum amount of time, in seconds, an inactive connection is available before the connection is closed. The default value is 3600 seconds (60 minutes). A value of `*NOMAX` indicates that there is no limit.

-connPoolMaxLifetime *NOMAX | *max-lifetime*
Specifies the maximum life, in seconds, for an available connection before the connection is closed. The default value is 86400 seconds (24 hours). A value of *NOMAX indicates that there is no limit.

-connPoolMaxUseCount *NOMAX | *max-use-count*
Specifies the maximum number of times a connection can be used before it is replaced in the pool. The default value of *NOMAX indicates that there is no limit.

-connPoolFillCount *fill-count*
Specifies the number connections the connection pool is filled with when the web service is started. The default value is 1.

-connPoolUseThreads true | false
Specifies whether additional threads are used to perform maintenance on the connection pool. If true, an extra thread is created to perform maintenance. If false, maintenance will be performed on thread handling the client request. The default value is true.

-transportMetadata *NONE | *metadata-list*
Specifies what transport metadata to pass to the web service implementation code. Transport metadata is passed as environment variables. The default value of *NONE indicates that no metadata is set. A colon delimited string of metadata can also be specified. Supported value(s): REMOTE_ADDR, REMOTE_USER, REQUEST_METHOD, REQUEST_URL, REQUEST_URI, QUERY_STRING, SERVER_NAME and SERVER_PORT.

Note: REMOTE_ADDR is the only supported value for servers running version 1.3 or 1.5 and any other values that are set will be ignored.

-transportHeaders *NONE | *header-list*
Specifies what transport headers (e.g. HTTP headers) to pass to the web service implementation code. Transport headers are passed as environment variables. The environment variable name for HTTP headers is made up of the specified HTTP header prefixed with 'HTTP_', all uppercased. For example, if 'Content-type' is specified, then the environment variable name would be 'HTTP_CONTENT-TYPE'. The default value of *NONE indicates that no transport headers should be set. A colon delimited string of transport headers can also be specified.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.
2. In order to run the web service implementation code under an authenticated user ID or set connection pool fill count, you may need to redeploy the web service since the enhancements were introduced in 2018.
3. If the web service implementation code is hosted on a remote system and the connection between the web services server and the web service is secure, the default key store for the web services server must be set.
4. If the web service implementation code is hosted on a remote system and the service is being run under a user profile other the server user profile, the server user profile must have *USE authority to the user profile the web service is running under on the remote system.

Example

The following command sets the properties for web service `ConvertTemp` that is deployed in the web services server named `MyServer`:

```
setWebServiceProperties.sh -server MyServer -service ConvertTemp
   -transportMetadata REMOTE_ADDR
```

setWebServicesServerProperties.sh command

The **setWebServicesServerProperties.sh** command sets various properties of the integrated web services server.

Synopsis

```
setWebServicesServerProperties.sh
    -server server-name  [ -portList port-list ]
    [ -httpPort http-port|old-port:new-port ]
    [ -httpsPort https-port|old-port:new-port ]
    [ -adminPort admin-port ]
    [ -contextRoot context-root ]
    [ -defaultKeystore keystore|*NONE ]
    [ -defaultKeystorePassword password ]
    [ -disableMustUnderstandCheck true|false ]
    [ -trace traceOptions ] [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server whose properties will be modified.

Optional arguments

-portList *port-list*
Specifies a colon delimited string containing which internal port numbers should be associated with the specified server.

Note: This argument should only be used for server versions 1.3 and 1.5. For all other servers, use the `-httpPort`, or `-httpsPort`, or `-adminPort` arguments.

-httpPort *http-port* | *old-port:new-port*
Specifies the HTTP port associated with the server. If only a port is specified, the default HTTP port for the server will be set to the specified value. If a colon delimited value is specified, the first value specified must exist in the server configuration and will be replaced by the second value. Valid values range from 1 to 65535. A value of -1 disables the port.

Note: This argument is ignored for server versions 1.3 and 1.5.

-httpsPort *https-port* | *old-port:new-port*
Specifies the HTTP SSL port associated with the server. If only a port is specified, the default HTTP SSL port for the server will be set to the specified value. If a colon delimited value is specified, the first value specified must exist in the server configuration and will be replaced by the second value. Valid values range from 1 to 65535. A value of -1 disables the port.

Note: This argument is ignored for server versions 1.3 and 1.5.

-adminPort *admin-port*
Specifies the internal administration port associated with the server. Valid values range from 0 to 65535. A value of 0 will result in an ephemeral port to be chosen at run time.

Note: This argument is ignored for server versions 1.3 and 1.5.

-contextRoot *context-root*
Specifies the name of the context root for the specified server. The context root name makes up part of the URL used to access all web services running on this server. This parameter can only be modified when the server is in a stopped state.

-defaultKeystore *keystore* | *NONE
Specifies the path to the default keystore for the server. The value must be the path to the SYSTEM keystore, `/QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB`. If `*NONE` is specified, the default keystore for the server is removed. If a keystore other than `*NONE` is specified, the `-defaultKeystorePassword` must be specified.

-defaultKeystorePassword *password*
Specifies the password to the keystore. If this parameter is specified, the parameter `-defaultKeystore` must also be specified.

-disableMustUnderstandCheck true | false
Specifies whether the server issues a SOAP fault if a SOAP header with `mustUnderstand` attribute set to `true` can not be processed by the server. If `true`, a SOAP fault is not issued if the SOAP header is not handled by the server. If `false`, a SOAP fault is issued if the SOAP header is not handled by the server.

Note: This parameter is only supported for versions 1.3 and 1.5 of the server.

-trace *traceOptions*
Specifies a colon delimited string containing at least one of the following trace option values:

- `none` : disables all tracing
- `all` : enables all tracing
- `toolbox` : enables toolbox tracing
- `message` : enables message tracing
- `runtime` : enables runtime tracing

Note: The `message` and `runtime` trace options is only supported on versions 1.3 and 1.5 of the server.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have `*ALLOBJ` authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.
2. If the default keystore is set, the user profile used to run the web services server must have `*RX` (read, execute) authority to all parts of the key store path. In addition, the server that is hosting the web service implementation code has to have the Remote Command Server and the Signon Server applications configures to use SSL as documented in the support page Configuring the SSL Telnet and Host Servers for Server Authentication for the First Time.

Example

The following command sets the context root and enables toolbox tracing for the web services server named `MyServer`:

```
setWebServicesServerProperties.sh -server MyServer
     -contextRoot /web/service -trace toolbox
```

startWebService.sh command

The **startWebService.sh** command starts a web service that is in a stopped state.

Synopsis

```
startWebService.sh
    -server server-name  -service service-name
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server which contains the service to be started.

-service *service-name*
Specifies the name of web service to be started.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see “Permissions” on page 58.

Example

The following command starts the web service `ConvertTemp` that is deployed in the web services server named `MyServer`:

```
startWebService.sh -server MyServer -service ConvertTemp
```

startWebServicesServer.sh command

The **startWebServicesServer.sh** command starts an integrated web services server.

Synopsis

```
startWebServicesServer.sh
    -server server-name  [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server to be started.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command starts the integrated web services server named `MyServer`:

```
startWebServicesServer.sh -server MyServer
```

stopWebService.sh command

The **stopWebService.sh** command stops a web service that is in a started state.

Synopsis

```
stopWebService.sh
    -server server-name  -service service-name
    [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server which contains the service to be stopped.

-service *service-name*
Specifies the name of web service to be stopped.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command stops the web service `ConvertTemp` that is deployed in the web services server named `MyServer`:

```
stopWebService.sh -server MyServer -service ConvertTemp
```

stopWebServicesServer.sh command

The **stopWebServicesServer.sh** command stops an integrated web services server.

Synopsis

```
stopWebServicesServer.sh
    -server server-name  [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server to be stopped.

Optional arguments

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command stops the integrated web services server named `MyServer`:

```
stopWebServicesServer.sh -server MyServer
```

uninstallWebService.sh command

The **uninstallWebService.sh** command uninstalls a web service from an integrated web services server..

Synopsis

```
uninstallWebService.sh
    -server server-name -service service-name
    [ -stopService ] [ -printErrorDetails ] [ -help ]
```

Arguments

Required arguments

-server *server-name*
Specifies the name of the web services server in which the web service is deployed.

-service *service-name*
Specifies the name of web service to be uninstalled.

Optional arguments

-stopService
Specifies whether the service should be stopped before an uninstall. If not specified, an error will be returned if the service is active.

-help
Displays a help message and exits.

-printErrorDetails
Indicates that additional error information, such as stack traces and error codes, should be shown if the command fails.

Usage notes

1. To use the command a user must have *ALLOBJ authority or must have permission to the server. To learn more about permissions, see "Permissions" on page 58.

Example

The following command uninstalls web service ConvertTemp that is deployed in the web services server named MyServer:

```
uninstallWebService.sh -server MyServer -service ConvertTemp
```

Part 3. Web service programming considerations

This part of the document describes general programming considerations and techniques.

Chapter 8. General programming considerations and techniques

This chapter contains information of general interest that you may need to consider or be aware of when deploying web services.

Simplifying web service URIs

If we deploy a web service using a service name of `ConvertTemp` to an integrated web services server, the resultant URLs for SOAP and REST would look similar to what is shown in Figure 65 on page 131.

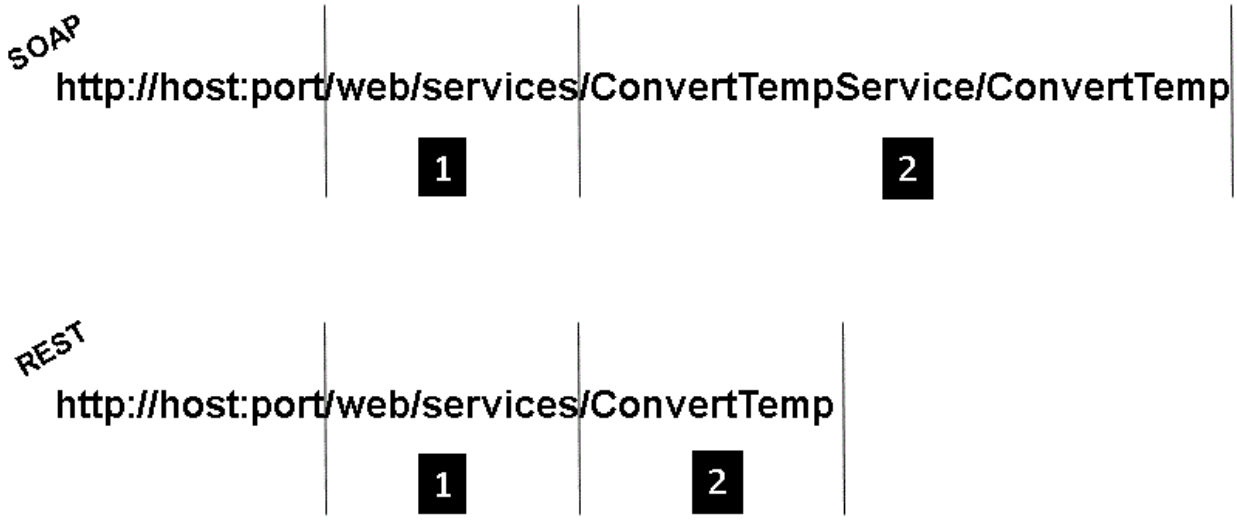


Figure 65: Example URLs for SOAP and REST web service

The first part of the URI path (1) is obtained from the context root that is specified in the web services tab of the server properties as described in Figure 47 on page 90. The second part (2) of the path is generated from the web service name and is not configurable.

You may decide that you want to change the URL to meet some company standard. Or maybe you simply want to simplify the SOAP URL. There is a way to do this if you use the associated HTTP server and you ensure that web service requests go to the HTTP server. This can be done by using the HTTP server's URL rewriting support.

Let us look at an example. Suppose we wanted users to issue SOAP requests using the following URL format:

```
http://host:port/ws/<web-service>
```

where `<web-service>` is the web service name. So if a user wanted to invoke the `ConvertTemp` web service, then the following URL would be used:

```
http://host:port/ws/ConvertTemp
```

And if a user wanted to retrieve the WSDL, the following URL would be used:

```
http://host:port/ws/ConvertTemp?wsdl
```

So how do we achieve this? By adding the following directives to the HTTP server configuration file, we can meet the goal:

```
RewriteEngine On
RewriteRule ^/web/services/(.*)Service/(.*)$ /web/services/$1Service/$2 [PT]
RewriteRule ^/ws/(.*)\?wsdl$ /web/services/$1Service/$1?wsdl [PT]
RewriteRule ^/ws/(.*)$ /web/services/$1Service/$1 [PT]
```

The `RewriteEngine` directive enables or disables the runtime rewriting engine. In this case we are enabling it. The `RewriteRule` directives do the mapping of the URI. That is it. We have achieved the goal of simplifying the URL.

There are a lot of things you can do with the HTTP server's URL rewriting support. To learn more about these directives, search on the HTTP server directives in the IBM Knowledge Center.

Web services and independent ASPs

An integrated web services server located in an independent auxiliary storage pool (ASP) is not supported. However, you can deploy a program or service program located in an independent ASP.

In order to deploy a program object that is located in an independent ASP, you must specify an absolute path to the program object. An available independent ASP has a directory in the root directory. The directory has the same name as the independent ASP. When the independent ASP is available, the contents of the independent ASP are mounted to the independent ASP directory. For example, if an independent ASP has a name of XSM, and program `MYPGM` to be deployed as a web service is in library `MYLIB`, then the path to the program would be `/XSM/QSYS.LIB/MYLIB.LIB/MYPGM.PGM`.

Once deployed, prior to calling the program to handle an incoming client request, the set auxiliary storage pool group (SETASPGRP) command is run in the host server job in order to set the ASP group. Once the specified ASP group has been set, all libraries in the independent ASPs in the ASP group are accessible and objects in those libraries can be referenced using regular library-qualified object name syntax. The libraries in the independent ASPs in the specified ASP group plus the libraries in the system ASP (ASP number 1) and basic user ASPs (ASP numbers 2-32) form the library name space for the job. All libraries in the library list need to be in the new library name space or the library list is not changed and the new ASP group is not set.

To learn more about independent ASPs, search on independent ASPs in the IBM Knowledge Center.

PCML considerations

An ILE program or service program may be composed of one or more module objects. A module object is a nonrunnable object that is the output of an ILE compiler. A module object is represented to the system by the symbol `*MODULE`. A module object is the basic building block for creating runnable ILE objects.

When a program object is submitted to be deployed as a web service, the integrated web services support goes through the process of obtaining all the PCML's from each module object. If the program object is a service program, then procedures that are not in the export list are removed from the PCML. Then all the PCML's are merged into one PCML.

The generated PCML can be found in the web service root directory. For example, if `ConvertTemp2` is an installed web service in server `WSERVICE1` located in directory `/www`, then the PCML file path would be `/www/WSERVICE1/webservices/services/ConvertTemp2/ConvertTemp2.pcml`.

Ensure that you are not using unsupported constructs for your programming language. You can find out about restrictions and limitations of PCML for your programming language in IBM Knowledge Center.

In the context of integrated web services server, you should be aware of the following restrictions and limitations:

- The following PCML data types are not supported: Time, Pointer, Procedure Pointer, 1-Byte Integer, and 8-byte Unsigned Integer.
- The following PCML constructs are not supported: Offsets and relative names.
- A procedure in an ILE service program (*SRVPGM) that is to be externalized as a web service can have a maximum of 7 parameters. An ILE program (*PGM) can have a maximum of 32 parameters on IBM i 6.1 and 255 parameters on IBM i 7.1 and subsequent releases.
- The size of a character parameter should not exceed 16700000 bytes.

PCML and the RPG programming language

It is a good idea that you ensure you specify the PGMINFO keyword in the RPG source code. The PGMINFO keyword specifies how program-interface information is to be generated for the module or program. Figure 66 on page 133 shows how to do this in fixed format and free format RPG.

```
Fixed format
H PGMINFO(*PCML:*MODULE)

Free format
Ctl-Opt PGMINFO(*PCML:*MODULE);
```

Figure 66: Specifying PGMINFO in RPG source code

By default, the compiler capitalizes all identifiers when generating the PCML. This means the XML and JSON documents that are exchanged between a client and server uses identifiers that are all capitalized. For example, the ConvertTemp sample web service that is deployed on integrated web services servers has SOAP request and response messages that look like the following:

```
Request message
<q0:converttemp>
  <arg0><TEMPIN>34</TEMPIN></arg0>
</q0:converttemp>

Response message
<a:converttempResponse>
  <return><TEMPOUT>1.11</TEMPOUT></return>
</a:converttempResponse>
```

You can control the case of the element identifiers by either editing the PCML file manually and referencing the PCML during the deployment of the program object, or by indicating that the declaration case of the identifier be used by adding the *DCLCASE keyword to the PGMINFO specification as follows:

```
Fixed format
H PGMINFO(*PCML:*MODULE:*DCLCASE)

Free format
Ctl-Opt PGMINFO(*PCML:*MODULE:*DCLCASE);
```

After adding *DCLCASE to the sample code, compiling it, and deploying the service program, the message flows now look like the following:

```
Request message
<q0:converttemp>
  <arg0><tempIn>34</tempIn></arg0>
</q0:converttemp>

Response message
<a:converttempResponse>
  <return><tempOut>1.11</tempOut></return>
</a:converttempResponse>
```

PCML and the COBOL programming language

You may specify the PGMINFO keyword in the COBOL source code in order to automatically generate PCML. The PGMINFO keyword specifies how program-interface information is to be generated for the module or program, as shown in Figure 67 on page 134.

```
PROCESS OPTIONS PGMINFO(PCML MODULE)
```

Figure 67: Specifying PGMINFO in COBOL source code

PCML and the C programming language

Although the C compiler cannot generate PCML for the C programming language, you can still use C as your web service implementation code. This may be done by manually generating the PCML file and then referencing the PCML file during the deployment of the C-based program or service program.

The one thing to note as a C programmer is that character fields must be padded with blanks and the fields are not null-terminated. So if you have an input field that is 10 bytes in length, the C program will receive the value "ABC" as "ABC" padded with seven blanks. For output fields, the field will need to be padded with blanks unless you have a field that specifies the length of the output field as documented in “Automatic length detection” on page 148.

Data type considerations

The following sections discusses how integrated web services server handles the supported data types.

Date and time types

The integrated web services server supports the date, time and dateTime data types.

The format of the date, time, and dateTime values is governed by the ISO (International Organization for Standardization) for representation of dates and times, and may be referenced by RFC 3339 (https://tools.ietf.org/html/rfc3339).

If the web service is REST and the incoming data media type is not XML, a UNIX timestamp may also be specified. A UNIX timestamp is just the number of milliseconds since 1 Jan 1970 00:00:00 UTC (Coordinated Universal Time). However, the recommendation is to use human readable forms since it is more precise with less ambiguity.

By default, time and dateTime values are passed to the web service implementation code in the timezone of the server. You can control the timezone of the server by adding the `user.timezone` Java property to the JVM properties of the integrated web services server by using the Web Administration GUI and updating the JVM properties by selecting **Server**->**Properties** link and selecting the **JVM options** tab. For example, if you wanted the timezone to be in Greenwich Mean Time (GMT), you would add the following JVM property:

```
-Duser.timezone=GMT
```

Setting the `user.timezone` Java property sets the timezone of the server (local time). This means that any time and dateTime values passed to the server will be normalized to the timezone specified by the `user.timezone` property before being passed to the web service implementation code. Similarly, any time or dateTime values returned by the web service implementation code will be returned in the timezone of the server.

By default, time or dateTime values that do not contain timezone information is assumed to be in the timezone of the server (local time). If you add the `-Dcom.ibm.iws.datetime.tz.utc=true` Java property to the JVM properties of the integrated web services server, then time or dateTime values that do not contain timezone information is assumed to be in UTC.

The following sections gives further details on how the integrated web services server handles date, time, and dateTime types.

The date type

The date value is specified in the form

```
YYYY-MM-DD[Z|(+|-)hh:mm]
```

where

- YYYY indicates the year
- MM indicates the month
- DD indicates the day
- Z indicates UTC
- hh indicates the hour
- mm indicates the minute

Examples of valid date values include:

- 1971-01-01
- 1971-01-01Z
- 1971-01-01+05:00

Notes:

1. If a date field is omitted or set to a null value on an incoming request, the value that will be passed to the web service implementation code is `0001-01-01`.
2. Timezones or time offsets are ignored, The web service implementation code will receive the date value as it was sent.
3. An error will result if the date is not specified in the form defined by the standard.

The time type

The time value is specified in the form

```
hh:mm:ss[Z|(+|-)hh:mm]
```

where

- hh indicates the hour
- mm indicates the minute
- ss indicates the second
- Z indicates UTC

Examples of valid time values include:

- 15:35:30
- 15:35:30Z
- 15:35:30-05:00

Notes:

1. If a time field is omitted or set to a null value on an incoming request, the value that will be passed to the web service implementation code is `00:00:00`.
2. If a timezone or time offset is not specified, the timezone of the server (local time) is used, unless the JVM property `-Dcom.ibm.iws.datetime.tz.utc=true` is specified, in which case the time value is considered to be UTC.

3. The time is normalized to the timezone of the server (local time).
4. Fractional seconds will be discarded.
5. An error will result if the time is not specified in the form defined by the standard.

The dateTime type

The dateTime value is specified in the form

```
YYYY-MM-DDThh:mm:ss.sss[Z|(+|-)hh:mm]
```

where

- YYYY indicates the year
- MM indicates the month
- DD indicates the day
- T indicates the start of the time section
- hh indicates the hour
- mm indicates the minute
- ss indicates the second
- Z indicates UTC

Examples of valid dateTime values include:

- 1971-01-01T15:35:30
- 1971-01-01T15:35:30.123
- 1971-01-01T15:35:30Z
- 1971-01-01T15:35:30-05:00

Notes:

1. If a dateTime field is omitted or set to a null value on an incoming request, the value that will be passed to the web service implementation code is `0001-01-01T00:00:00`.
2. If a timezone or time offset is not specified, the timezone of the server (local time) is used, unless the JVM property `-Dcom.ibm.iws.datetime.tz.utc=true` is specified, in which case the dateTime value is considered to be UTC.
3. The date and time are normalized to the timezone of the server (local time).
4. An error will result if the dateTime is not specified in the form defined by the standard.

Numeric types

The integrated web services server currently supports integer, packed, zoned, and float numeric types.

If a numeric field is omitted on an incoming request, the value that will be passed to the web service implementation code is `0`. Depending on the web service processor that is being used, invalid numeric data may result in an exception (such as invalid numeric JSON data) or simply a zero being passed to the web service implementation code (such as numeric data in an XML document).

National language considerations

The CCSID of character data that is sent by the integrated web services server to the web service implementation program object is dependent on the CCSID of the host server job. The server expects the data received from the web service implementation program object to be in the CCSID of the host server job as well.

A user can control what CCSID the data exchanged between the server and the web service implementation program object is in by modifying a connection pool property of the web service, **Connection CCSID**, as shown in Figure 68 on page 137.

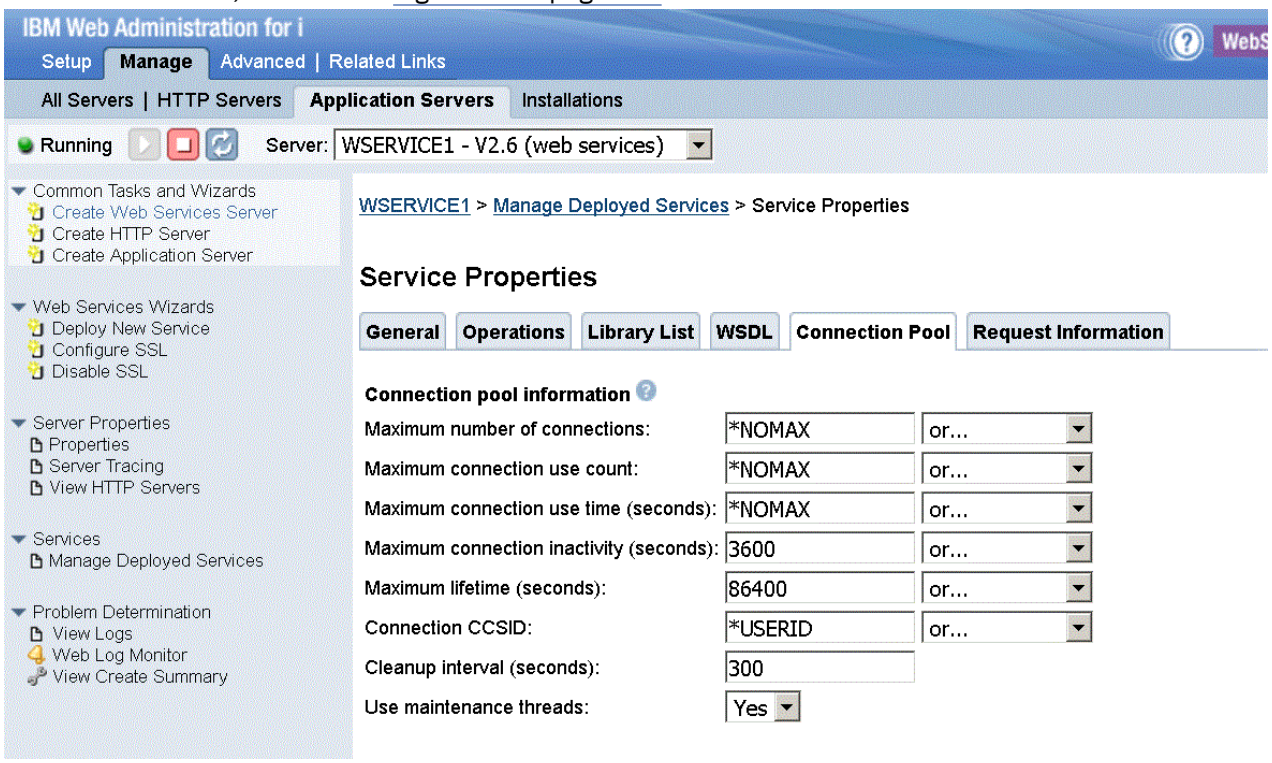


Figure 68: Properties panel - Connection Pool

REST-based web service considerations

This section gives information on web services based on REST principles.

How HTTP status codes set by web service implementation code is handled

As indicated previously in “Panel (REST-ONLY): Specify resource method information” on page 72, you can designate an output parameter as the parameter that will contain an HTTP status code to be returned to the client. The response that is returned to the client is dependent on the HTTP status codes:

- If the status code does not map to an existing status code as defined in the HTTP 1.1 protocol[12], then an HTTP status code of 500 (Internal Server Error) will be returned with no additional error content in the response body.
- If the status code is informational (1xx status codes), or indicates success (2xx status codes), or indicates redirection (3xx status codes), then any output parameters are processed and returned in the response body.
- If the status code indicates failure, then output parameters are not processed and the status code is returned with no additional content in the response body.

Returning user-defined content

A RESTful web service has the capability to return user defined content. That is, something other than XML or JSON. For example, a content type of `text/html`. In order for the web service implementation code to be able to do this, it must have one output parameter that is to be returned as a response, and the parameter must have a type of character. Note that you can even return binary images, assuming the web

[12] For a list of HTTP status codes, see https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.

service implementation code encodes the image in the Base64 MIME content transfer encoding and the `Content-Transfer-Encoding` HTTP header is set to `base64`.

SOAP-based web service considerations

This section gives information on web services based on the SOAP protocol.

Optional and nillable elements

As indicated previously in “Panel (SOAP-ONLY): Specify WSDL options” on page 78, you can indicate whether the WSDL should be generated with optional and/or nillable elements. Elements that are omitted and elements that have been designated to have a nil value will result in the web service implementation code receiving the field with all blanks if the field is a character field, or a value of zero if the field is a numeric field. If the field is a structure, then the sub-fields will contain blanks or zeros depending on the field type.

Mandatory elements not enforced by server

The server does not ensure that mandatory elements are sent. Field values for omitted fields will follow the rules discussed in “Optional and nillable elements” on page 138.

Chapter 9. Serviceability and troubleshooting

There are several tools to help identify problems with the server and the web services deployed to it. In this chapter we cover trace, server dump, and web service debugging.

Note: It is best to ensure the latest HTTP Group PTF for your release is loaded and applied. All fixes and enhancements relating to the integrated web servers server are packaged as part of the HTTP Group PTF.

Tracing

The integrated web services server has the ability to produce a variety of traces that enable you to debug issues with the server and your applications. In this section we describe where to find the output and how to configure what trace data is collected.

The integrated web services server records limited information by default. This basic information is useful for debugging common configuration issues. You can view the output logs by opening the `messages.log` file by clicking on the **View Logs** link in the navigation bar (see Figure 69 on page 139).

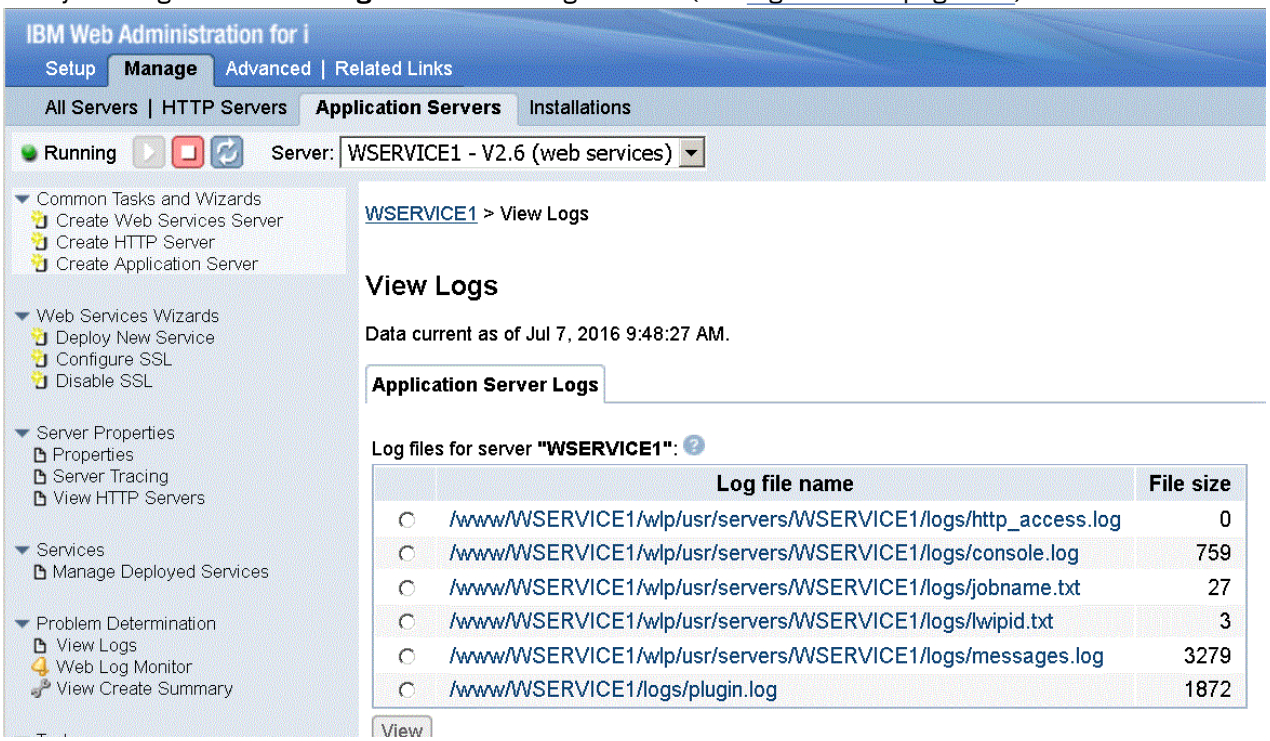


Figure 69: View logs panel

Some files are associated with internal server startup processing. For example, `jobname.txt` and `lwipid.txt`. From the perspective of a programmer or administrator, there are four primary log files for an integrated web services server and any associated HTTP server:

- `console.log` - containing the redirected standard output and standard error from the JVM process. This console output is intended for direct human consumption. The console output contains major events and errors. The console output also contains any messages that are written to the standard output and standard error streams. The console output always contains messages that are written directly by the JVM process, such as the output generated by the JVM property `-verbose:gc`.
- `messages.log` - containing all messages except server trace messages that are written or captured by the logging component. All messages that are written to this file contain additional information such as the message time stamp and the ID of the thread that wrote the message. This file does not contain

messages that are written directly by the JVM process. This file will contain Java toolbox trace records, discussed in “Configuration of Java toolbox trace” on page 140.

- `trace.log` - containing all messages that are written or captured by the product. This file is created only if you enable the advanced server tracing discussed in “Configuration additional server trace” on page 141. This file does not contain messages that are written directly by the JVM process.
- `plugin.log` - contains all messages written by the HTTP server plug-in. The plug-in enables the IBM HTTP server to communicate with the integrated web services server.

Figure 70 on page 140 shows sample output written to `messages.log` from a server starting.

```
****************************************************************************
product = WebSphere Application Server 8.5.5.9 (wlp-1.0.12.cl50920160227-1523)
wlp.install.dir = /QIBM/ProdData/OS/ApplicationServer/runtime/wlp/
server.config.dir = /www/WSERVICE1/wlp/usr/servers/WSERVICE1/
java.home = /QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/jre
java.version = 1.7.0
java.runtime = Java(TM) SE Runtime Environment (jvmap3270sr9fp40-20160422_012.6)
os = OS/400 (V7R3M0; ppc) (en_US)
process = 3489@UT30P44.RCH.STGLABS.IBM.COM
****************************************************************************
[7/10/16 21:01:35:539 CDT] 00000001 com.ibm.ws.kernel.launch.internal.FrameworkManager
A CWWKE0001I: The server WSERVICE1 has been launched.
[7/10/16 21:01:37:547 CDT] 00000020 com.ibm.ws.logging.internal.TraceSpecification
I TRAS0018I: The trace state has been changed. The new trace state
is *=info:org.apache.cxf.*=warning.
[7/10/16 21:01:37:935 CDT] 00000001 com.ibm.ws.kernel.launch.internal.FrameworkManager
I CWWKE0002I: The kernel started after 3.052 seconds
[7/10/16 21:01:38:161 CDT] 00000025 com.ibm.ws.kernel.feature.internal.FeatureManager
I CWWKF0007I: Feature update started.
[7/10/16 21:01:42:020 CDT] 0000002a com.ibm.ws.tcpchannel.internal.TCPChannel
I CWWKO0219I: TCP Channel defaultHttpEndpoint has been started and is now
listening for requests on host *  (IPv4) port 10022.
[7/10/16 21:01:42:416 CDT] 00000033 com.ibm.ws.app.manager.AppMessageHelper
I CWWKZ0018I: Starting application ConvertTemp.
[7/10/16 21:01:43:256 CDT] 00000033 com.ibm.ws.webcontainer.osgi.webapp.WebGroup
I SRVE0169I: Loading Web Module: ConvertTemp.
[7/10/16 21:01:43:259 CDT] 00000033 com.ibm.ws.webcontainer
I SRVE0250I: Web Module ConvertTemp has been bound to default_host.
[7/10/16 21:01:43:260 CDT] 00000033 com.ibm.ws.http.internal.VirtualHostImpl
A CWWKT0016I: Web application available (default_host):
http://ut30p44:10022/web/services/ConvertTempService/
[7/10/16 21:01:43:263 CDT] 00000033 com.ibm.ws.app.manager.AppMessageHelper
A CWWKZ0001I: Application ConvertTemp started in 0.848 seconds.
[7/10/16 21:01:43:346 CDT] 00000025 com.ibm.ws.kernel.feature.internal.FeatureManager
A CWWKF0015I: The server has the following interim fixes installed:
PI58918,PI57228,PI58457.
[7/10/16 21:01:43:347 CDT] 00000025 com.ibm.ws.kernel.feature.internal.FeatureManager
A CWWKF0012I: The server installed the following features: [jaxws-2.2, ssl-1.0,
localConnector-1.0, json-1.0, servlet-3.0, jaxrs-1.1, jaxb-2.2].
[7/10/16 21:01:43:347 CDT] 00000025 com.ibm.ws.kernel.feature.internal.FeatureManager
I CWWKF0008I: Feature update completed in 5.418 seconds.
[7/10/16 21:01:43:347 CDT] 00000025 com.ibm.ws.kernel.feature.internal.FeatureManager
A CWWKF0011I: The server WSERVICE1 is ready to run a smarter planet.
```

Figure 70: Sample output of messages.log

The log shows the server startup procedure. First, the kernel starts. Then the feature manager initializes and reads the configuration files. The server is configured to listen on a given port. Then the `ConvertTemp` web service is started and finally the server is ready to serve the web services.

Enabling trace will produce lots of information and will affect the performance of the web service, so you do not want to enable traces in a production environment unless you really need to.

Configuration of Java toolbox trace

As has been previously discussed in “Server programming model” on page 51, there is Java proxy code that invokes the ILE program object web service implementation code by using Java toolbox classes. The communication flow between the proxy code and the web service implementation code may be traced by enabling Java toolbox trace.

To configure Java toolbox trace, follow these simple steps:

1. Click on the **Server Tracing** link to bring up the server tracing panel as shown in Figure 71 on page 141.

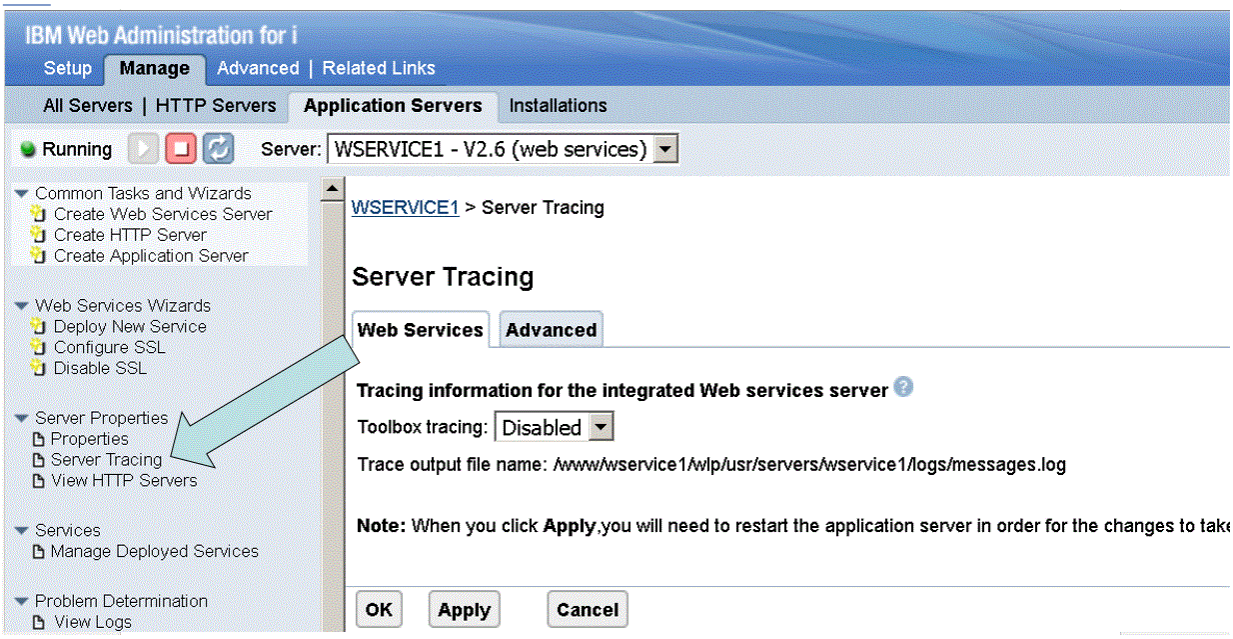


Figure 71: Server tracing properties

2. Enable Toolbox tracing.
3. Press **OK** or **Apply**.
4. Restart the server.

Configuration additional server trace

The integrated web services server can be configured to gather debug information for the server runtime. You can modify the server runtime tracing level by clicking on the server tracing **Advanced** tab as shown in Figure 72 on page 141.

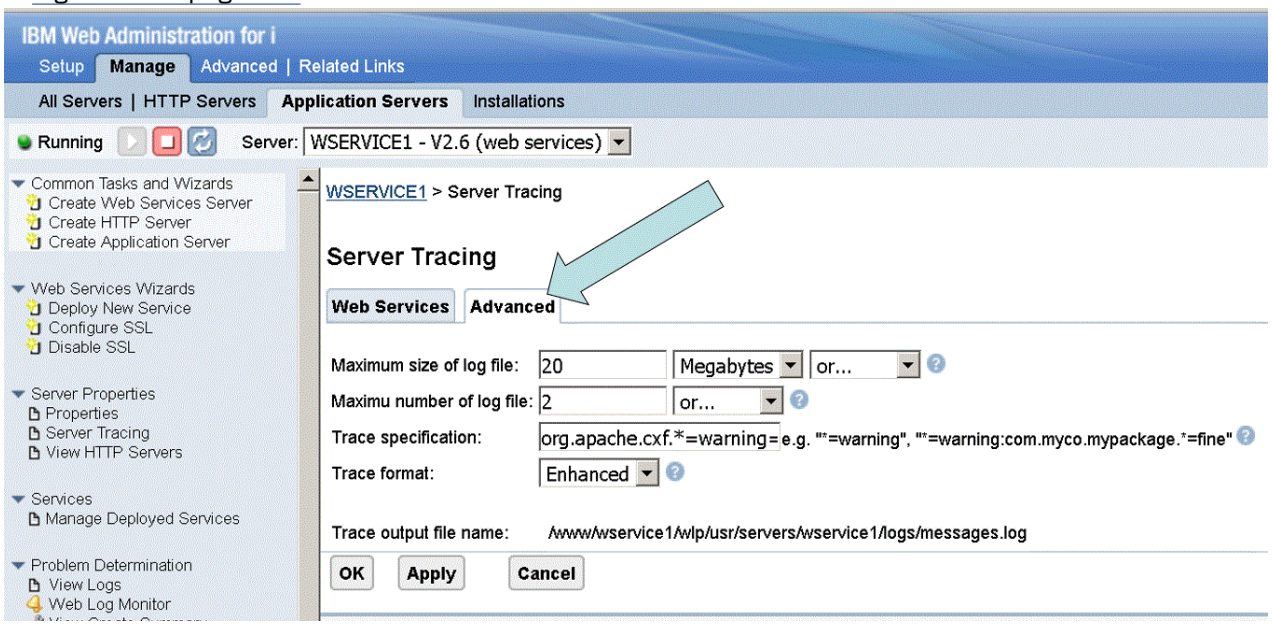


Figure 72: Server tracing properties - advanced tab

The format of the trace specification is:

```
<component> = <level>
```

where <component> is the component for which to set a log detail level, and <level> is one of the valid logger levels shown in Table 10 on page 142. Separate multiple trace specifications with colons (:).

Components correspond to Java packages and classes, or to collections of Java packages. Use an asterisk (*) as a wildcard to indicate components that include all the classes in all the packages that are contained by the specified component. For example:

*****
Specifies all traceable code that is running in the server, including the product system code and customer code.

com.ibm.ws.*
Specifies all classes with the package name beginning with com.ibm.ws.

com.ibm.ws.classloader.JarClassLoader
Specifies the JarClassLoader class only.

The following table shows valid logging levels:

Table 10: Valid logging levels.

Logging level	Description
off	Logging is turned off.
fatal	Task cannot continue and component, application, and server cannot function.
severe	Task cannot continue but component, application, and server can still function. This level can also indicate an impending unrecoverable error.
warning	Potential error or impending error. This level can also indicate a progressive failure (for example, the potential leaking of resources).
audit	Significant event that affects server state or resources.
info	General information that outlines overall task progress.
config	Configuration change or status.
detail	General information that details subtask progress.
fine	General trace information plus method entry, exit, and return values.
finer	Detailed trace information.
finest	A more detailed trace that includes all the detail that is needed to debug problems.
all	All events are logged, includes custom levels, and can provide a more detailed trace than finest.

Server runtime tracing is typically asked for by IBM service to debug a problem in the server.

Server dump

The integrated web services server is an application server running in a Java Virtual Machine (JVM). As such, you have the ability, with the proper authority, to diagnose problems at the JVM level, such as hung threads, deadlocks, excessive processing, excessive memory consumption, memory leaks, and defects in the virtual machine.

The most straightforward way to look at various aspects of the server is by using the Work with Java Virtual Machine (WRKJVMJOB) CL command. The following information or functionality is available for IBM Technology for Java JVM jobs:

- The arguments and options with which the JVM was started.
- Environment variables for both ILE and PASE.
- Java locks being blocked, held and waiting on.
- Garbage collection information.
- The properties with which the JVM was started.
- The properties with which the JVM is currently running.
- The list of threads associated with the JVM.
- The partially completed job log for the JVM job.
- The ability to generate JVM (System, Heap, and Java) dumps.
- The ability to enable and disable verbose garbage collection.

To learn more about the WRKJVMJOB command, search on the command in the IBM Knowledge Center.

Web service debugging

Sometimes invoking the web service may result in an exception that looks like the following in the `messages.log`:

```
[7/11/16 9:44:22:263 CDT] 00005fbb SystemErr
R java.lang.RuntimeException: java.lang.RuntimeException:
Invocation of program failed.
AS400Message (ID: RNX0105 text: A character representation of a numeric
value is in error.):com.ibm.as400.access.AS400Message@30e030e0
```

Error messages that include `com.ibm.as400.access.AS400Message` in the error message usually means that you ILE program or service program took some sort of exception. In these cases you may want to start the debugger and debug the code step by step. You can either use the traditional system debugger or the graphical debugger. The following steps assumes you are using the traditional system debugger:

1. Issue a web service request to cause the error.
2. Find the host server job that handled the request. Web services are run in the QUSRWRK subsystem. There may be lots of jobs in the subsystem, so it is best to debug the code where you can control number of requests to the web service. In this example it is easy to find the job (see Figure 73 on page 144) since it is running under the default web service user profile `QWSERVICE`.

```
                          Work with Active Jobs                    LP103U27
                                                                07/11/16  09:55:58
CPU %:      .5     Elapsed time:   55:15:43     Active jobs:   241

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...
                    Current
Opt  Subsystem/Job  User        Type  CPU %  Function        Status
__     QZRCSRVS     AMRA        PJ       .0                 TIMW
__     QZRCSRVS     QWSERVICE   PJ                          TIMW
__   QALERT         QSYS        SYS      .0                 DEQW
__   QCMNARB01      QSYS        SYS      .0                 EVTW
__   QCMNARB02      QSYS        SYS      .0                 EVTW
__   QCMNARB03      QSYS        SYS      .0                 EVTW
     QCMNARB04      QSYS        SYS      .0                 EVTW
```

Figure 73: Finding host server job for web service

3. Look at the job log of the web service by choosing option 5 from the WRKACTJOB panel and then option 10 - Display job log, if active, on job queue, or pending. You should find the error in the job log (see Figure 74 on page 144) and may be able to diagnose the problem simply from the error message.

```
                          Display Job Log
                                                          System:   LP103U27
Job . . :   QZRCSRVS      User . . :   QUSER        Number . . . :   047041

_    Client request - run program QSYS/QUSRJOBI.
     Client request - run command QSYS/ADDLIBLE.
     Library QSYSDIR added to library list.
     Client request - run program QSYS/QZRUCLSP.
     Client request - run program QSYS/QZRUCLSP.
     A character representation of a numeric value is in error.
```

Figure 74: Jog log of host server job for web service

4. Record the job information and go to the command line and perform the Start Service Job (STRSRVJOB) command. The STRSRVJOB command starts the remote service operation for a specified job (other than the job issuing the command) so that other service commands can be entered to service the specified job. Any dump, debug, and trace commands can be run in that job until service operation ends. Service operation continues until the End Service Job (ENDSRVJOB) command is run. Here is an example:

```
STRSRVJOB JOB(047041/QUSER/QZRCSRVS)
```

5. Put the job in debug mode by issuing the Start Debug (STRDBG) command. Note that the program or service program must have modules that have been compiled with source debug views. The following is an example of the STRDBG command:

```
STRDBG DFTPGM(*NONE) UPDPROD(*YES) SRVPGM(AMRA/QIWSSAMPLE)
```

   When the command is run, the command will take you to the display module source panel. You can add additional modules.

6. Add breakpoints to the source and then exit.
7. Run the web service request. The breakpoint should pop up and you can step through the code.

Part 4. Advanced topics

This part of the document provides information on the topics of security and performance.

Chapter 10. Performance tuning

When looking at improving performance, one needs to look at the entire *web environment*. In this context, a web environment is a grouping of related web server, application server, and operating system settings that form a web solution. This includes any web services running in the application servers (i.e. integrated web services servers). This chapter attempts to highlight various approaches to improving the performance of a web service. The information in this section share common approaches to solving common problems based on real environments. They do not provide a “one size fits all” solution. As technology evolves, new recommendations and information might be added to the information in this document.

For a definitive discussion on performance, you should read the topics under the performance management subtopic in the IBM i system management web page. Of particular importance are the following documents:

- The *IBM i on Power - Performance FAQ* document. This document is intended to address most frequently asked questions concerning IBM i performance on Power Systems, and provide best practice guidelines for most commonly seen performance issues. Here is a link to the IBM i on Power - Performance FAQ.
- The *Performance Capabilities Reference* documents. The purpose of this document is to help provide guidance in terms of IBM i operating system performance, capacity planning information, and tips to obtain optimal performance on IBM i operating system. Here is a link to the IBM i 7.2 Performance Capabilities Reference (January 2016).

A key reference from which a lot of the information in this chapter is derived from is the *WebSphere Application Server Performance Cookbook*. Although the cookbook covers performance tuning for WebSphere Application Server, there is a very strong focus on Java, Operating Systems, and theory which can be applied to other products and environments. Here is the link to the cookbook.

Performance tuning the web service

The following sections discuss some basic considerations for achieving high-performance, which you should know once you start designing a web services application.

Parsing and payload size

The resources, such as CPU and storage, that are consumed by the integrated web services server in processing web service requests are principally affected by the efficiency of the XML/JSON parsing and the amount of data transmitted. The most critical performance consideration is the translation between the XML/JSON messages and the Java object. Application design, deployment, and tuning can be applied in order to improve such performance.

For a given message, the process of generating the outbound message consumes less CPU than the process of parsing an inbound message of the same size. Thus the inbound message size has a more significant impact on performance than the outbound message size. The XML/JSON associated with inbound messages is parsed to extract the data elements that are to be passed to the application. Structuring the XML/JSON to reduce the complexity of the elements is likely to be one of the best methods to improve the scalability of web service applications. Note that as with payload size, the complexity of the inbound message is of prime importance because XML/JSON parsing is only performed for the inbound message, and not on the outbound message.

The key factors affecting the XML/JSON parsing costs are:

1. The number of XML/JSON elements
2. The size of each element
3. The size of the element tags

The obvious rule for optimizing your web service performance is to keep your payload small and simple. However, in the real world where you're trying to solve real business problems, you do not always have the luxury of adhering to this rule. Long running business processes may require that XML/JSON documents be exchanged that capture not only the relevant business information, but also the state of the process. Larger messages result in longer parsing times while complex XML/JSON structures with nested elements result in longer times for the marshalling and un-marshalling of objects and XML/JSON elements. The goals should be an awareness of these impacts and spending time architecting your programming objects to minimize the size and complexity of the message structures. However, you should choose to support a single invocation that includes a somewhat larger and more complex message versus supporting separate individual message transactions.

Automatic length detection

When deploying web services, ensure that **Detect length fields** (see Figure 75 on page 148) is selected and that length fields are used to describe number of elements in an output array and the actual lengths of very large character fields.

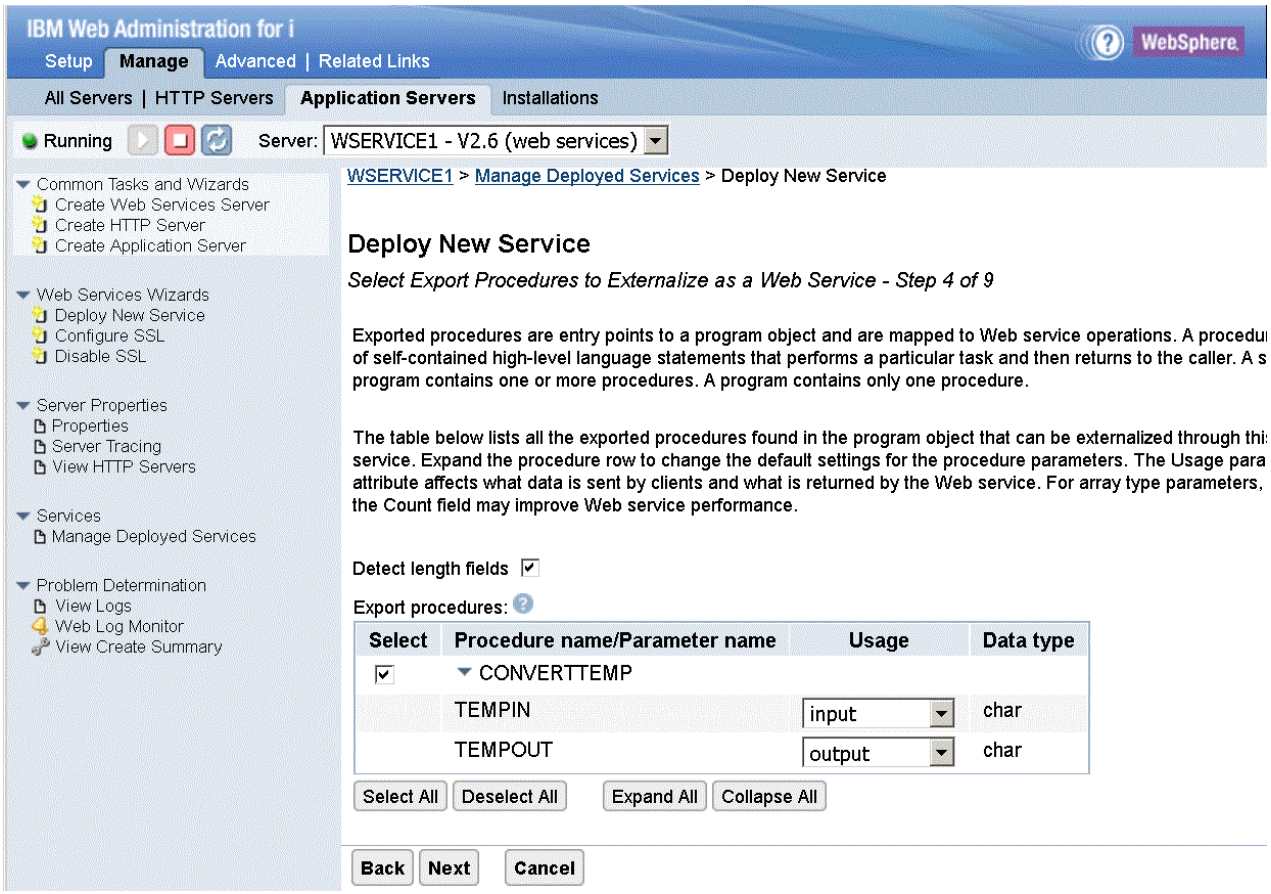


Figure 75: Select export procedures to externalize as a web service

The benefits of enabling **Detect length fields** as it pertains to performance include:

- Support of arrays, including nested output arrays, in an efficient manner. This is done by assuming that any numeric field that immediately precedes an array field with the same name as the array field appended with _LENGTH is a length field that will be used to indicate the actual number of elements in the array. Without this support, empty elements would be returned in the response.
- Improves the processing of very large output character fields. This is done by assuming that any numeric field that immediately precedes a character field with the same name as the character field appended with _LENGTH is a length field that will be used to indicate the actual number of characters in

the field. Without this support, the length of the string is determined by traversing the field a byte at a time, from right to left, looking for the first non-blank character.

It should be noted that length fields are hidden from the client and is not returned in the client response.

Connection pools

Each web service has a connection pool (see Figure 76 on page 149). The connection pool contains connections to a pool of host server jobs that are used to handle web service requests. Modifying these properties may improve the performance of the web service.

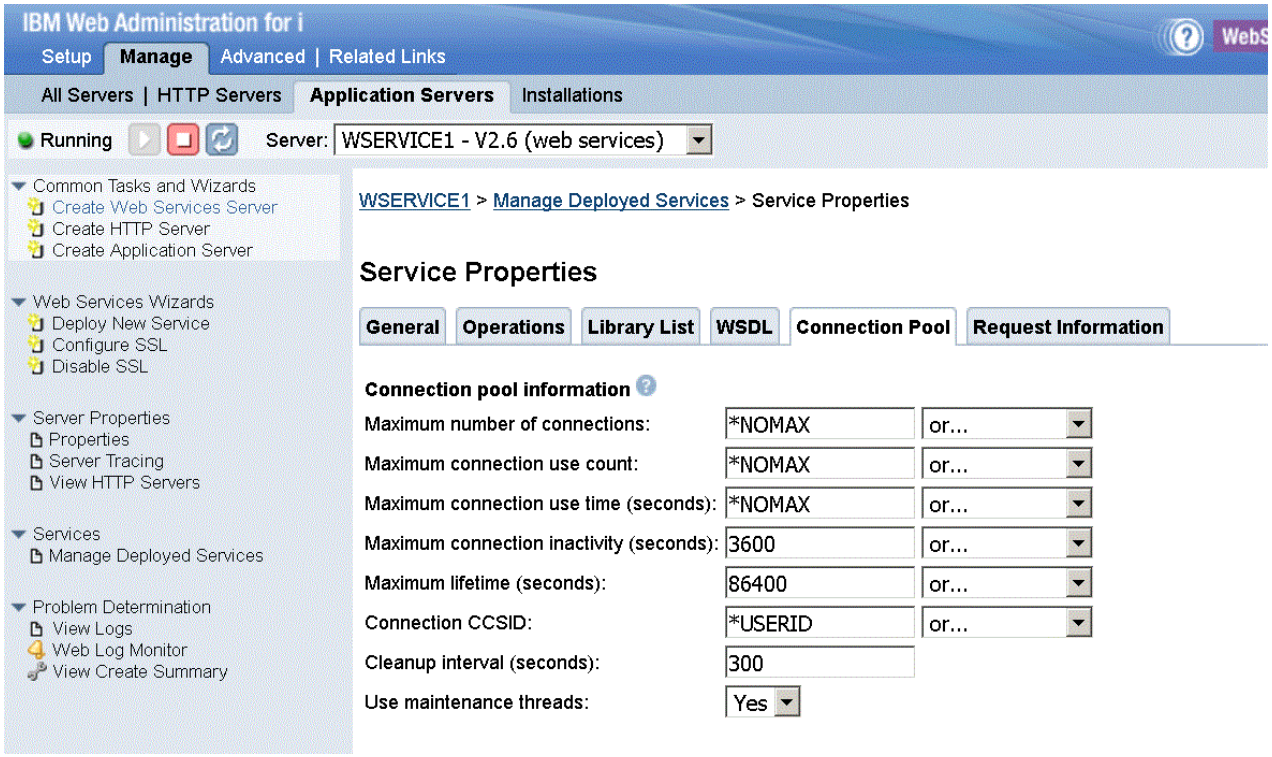


Figure 76: Properties panel - Connection Pool

A description of the connection pool attributes follows:

Maximum number of connections
Specifies the maximum number of connections. The default value of *NOMAX indicates no limit to the number of connections.

Maximum connection use count
Specifies the maximum number of times a connection can be used before it is replaced in the pool. The default value of *NOMAX indicates that there is no limit.

Maximum connection use time
Specifies the maximum time, in seconds, a connection can be in use before it is closed and returned to the pool. The default value of *NOMAX indicates that there is no limit.

Maximum connection inactivity
Specifies the maximum amount of time, in seconds, an inactive connection is available before the connection is closed. The default value is 3600 seconds (60 minutes). A value of *NOMAX indicates that there is no limit.

Maximum lifetime
Specifies the maximum life, in seconds, for an available connection before the connection is closed. The default value is 86400 seconds (24 hours). A value of *NOMAX indicates that there is no limit.

Connection CCSID
Specifies the coded character set identifier (CCSID) used when converting character data that is sent to the web service implementation code. Data received from the web service implementation code is assumed to be in the CCSID that is specified. The default value of `*USERID` indicates that the CCSID of user profile used to run the web service is used.

Cleanup interval
Specifies the time interval, in seconds, for how often the connection pool maintenance daemon is run. The default value is 300 seconds (5 minutes).

Use maintenance threads
Specifies whether additional threads are used to perform maintenance on the connection pool. Using additional threads will be beneficial to performance. If set to YES, an extra thread is created to perform maintenance. If set to `NO`, maintenance will be performed on thread handling the client request. The default value is YES.

Host server jobs

The web service implementation code is run in remote command and program call host server jobs. By default, there is only one host server job that is pre-started. You may want to increase the number of pre-started jobs by using the Change Prestart Job Entry (CHGPJE) command. To learn more about prestart jobs, search on "Use of prestart jobs" in the IBM Knowledge Center.

Performance tuning the HTTP server

If you are front-ending the integrated web services server with an associated HTTP server, you may want to adjust the `ThreadsPerChild` directive. This directive is used to specify the maximum number of threads per server child process.

Performance tuning the integrated web services server

For the most part, the integrated web services server is created with optimal performance attributes. However, there is one area that may be optimized, and that is the JVM. For best performance use the most current JDK level you can and apply the latest PTFs.

The server is created with the following memory options:

```
-Xmx1024m
-Xms64m
```

For a development environment, you might be interested in faster server startup, so consider setting the minimum heap size to a small value, and the maximum heap size to whatever value is needed for your web services. For a production environment, setting the minimum heap size and maximum heap size to the same value can provide the best performance by avoiding heap expansion and contraction.

The updating of JVM options is discussed in “Properties relating to the server” on page 86.

Performance tuning the network

The network adaptors being used will determine the maximum speed which could be reached, e.g. 1Gb, 10Gb, etc (at least in theory). However the protocol being used, the networking parameters set and the quality of the connections as well as other systems in the same subnet in the network will determine the actual performance in terms of network throughput and speed.

- Consider increasing the TCP/IP buffer for send and receive operations via the CHGTCPA command to a value greater than the default of 64KB. This can significantly reduce the network traffic.

- Ensure the parameters for current line speed is reflecting what the adapter is being capable of, e.g. 1Gb, 10Gb, etc. A single device with a lower line speed capability will force the whole subnet in the network to run at the lower speed and can severely degrade network performance.
- Ensure the current DUPLEX parameter is set to *FULL so the connection can be used for send and receive at the same time.
- Consider increasing the maximum frame size from 1496 bytes to 8996 bytes as this can significantly reduce the traffic between the system and the next network router and speed up the connections especially when virtual Ethernet connections are used.

Load balancing

The integrated web services server does not support clusters. However, you can create a *conceptual* cluster of servers and have a load balancer spray the requests among the servers. The servers are a conceptual cluster in that the servers do not know about each other and there is no central management interface that allows you to manage the servers as one.

There are different topologies you may choose to use, the most common include:

- Having multiple integrated web services servers in a partition. In this topology the TCP/IP ports used by the servers must be different.
- Having an integrated web services server on multiple partitions or systems. In this topology the server may be duplicated across all the partitions or systems. The duplication of a server can easily be done by creating an integrated web services server, deploying the web services to the server, and then saving the server by using the `saveWebServicesServer.sh` script and restoring the server on the various partitions and systems using the `restoreWebServicesServer.sh` script.
- Having a combination of multiple integrated web services servers in a partition and multiple servers in multiple partitions or systems.

Consider using the `saveWebServices.sh` and `restoreWebServices.sh` scripts to help with the task of deploying web services to servers that are not identical (that is, if you are not able to save and restore entire servers using the `saveWebServicesServer.sh` and `restoreWebServicesServer.sh` scripts).

Whichever topology you use, you will need to ensure that the system is able to handle the workload.

Chapter 11. Security

Security is an essential component of any enterprise-level application. In this chapter we provide you with a basic introduction to various security options available to you when using the integrated web services server.

Configuring SSL

SSL client authentication occurs during the connection handshake by using SSL certificates. The SSL handshake is a series of messages that are exchanged over the SSL protocol to negotiate for connection-specific protection. During the handshake, the secure server requests that the client send back a certificate or certificate chain for the authentication.

You have the ability to configure SSL for the integrated web services server or the associated HTTP server. This can be done by going through the **Configure SSL** wizard discussed in "The configure SSL wizard" on page 85.

Enabling basic authentication

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint using HTTP basic authentication. HTTP basic authentication[13] is a simple challenge and response mechanism with which a server can request authentication information (a user ID and password) from a client. The client passes the authentication information to the server in an `Authorization` header. The authentication information is in base-64 encoding. Although the basic authentication data is base64-encoded, sending data over HTTPS (SSL) is recommended. The integrity and confidentiality of the data can be protected by the SSL protocol.

To enable HTTP basic authentication, the integrated web services server must be associated with an HTTP server. By default, all requests sent to the HTTP server are forwarded to the integrated web services server. Basic authentication is then configured in the HTTP server. Clients wanting to invoke the web service must go through the HTTP server. To enable basic authentication in the HTTP server, perform the following steps:

1. From the Web Administration for i interface, select the HTTP server associated with the integrated web services server.
2. From within the navigation bar, click on **Edit Configuration File**.
3. Replace the following directives:

   ```
   <Location />
     Require all granted
   </Location>
   ```

   with the following:

   ```
   <Location />
   AuthType Basic
   AuthName "IBM Server"
   PasswdFile %%SYSTEM%%
   require valid-user
   </Location>
   ```

If you do enable basic authentication in the HTTP server associated with the integrated web services server, you will have the ability to retrieve the authenticated user from the REMOTE_USER environment

[13] For details, see RFC 2617, *HTTP Authentication: Basic and Digest Access Authentication*, at http://www.ietf.org/rfc/rfc2617.txt..

variable, assuming you indicate that the REMOTE_USER transport metadata is to be passed to the web service implementation code (see Figure 60 on page 99).

You will find more detailed information about HTTP server basic authentication directives in the IBM Knowledge Center.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Glossary

Ajax
Asynchronous JavaScript And XML. Ajax provides the ability for client-side code to send data to and retrieve from a server in the background without interfering with the display behavior of the existing page.

ANSI
American National Standard for Information Systems

API
Application Programming Interface

attachment
Data that is attached to a message on the wire, separately from the SOAP envelope. Attachments are often used for sending large files or images.

certificate
A credential used as an identity of proof between the server and client. It consists of a public key and some identifying information that a certificate authority (CA), an entity to sign certificates, has digitally signed. Each public key has an associated private key and the server must prove that it has access to the private key associated with the public key contained within the digital certificate. A self-signed certificate means it is signed by the server itself. If a self-signed certificate is specified to a server, clients might not trust the connection. To obtain a signed certificate from a public CA, you need to generate a Certificate Signing Request (CSR) and send it to the CA. After a certificate is returned, it is imported to your keystore.

DLL
Dynamic Link Library

global handler
A handler that is called regardless of the web service or message name.

GSKit
Global Security Kit, IBM's SSL component

handler
A library component that has the ability to manipulate a SOAP message, thus allowing the user to customize or extend any message components. Handlers are invoked either just before a request message is transmitted or just after a response message has been received.

HTTP
HyperText Transfer Protocol.

IBM Toolbox for Java
A library of Java classes supporting client/server and Internet programming model to an IBM i system.

IEEE
Institute of Electrical and Electronic Engineers.

JSON
JavaScript Object Notation. Lightweight data-interchange format that is built on a collection of name/value pairs alongside ordered lists of values.

keystore
A storage facility for cryptographic keys and certificates. A private key entry in a keystore file holds a cryptographic private key and a certificate chain for the corresponding public key. A private key entry can be specified to a server when configuring SSL. A trusted certificate entry contains a public key for a trusted party, normally a CA. A trusted certificate is used to authenticate the signer of certificates provided by a server or client. The keystore types that the Web Administrator for i GUI supports are: JKS, JCEKS, PKCS12, and CMS. Additionally, the Digital Certificate Manager (DCM) *SYSTEM is also supported.

RPC
Remote Procedure Call

secure endpoint URL
Endpoint beginning with https

service handler
A handler that is specific to the web service with which it is associated.

SOAP
Simple Object Access Protocol

SSL
Secure Sockets Layer

SSL tunneling
In SSL tunneling, the client establishes an unsecure connection to the proxy server, and then attempts to tunnel through the proxy server to the content server over a secure connection where encrypted data is passed through the proxy server unaltered.

TCPIP
Transmission Control Protocol/Internet Protocol

WAR file
A file used to distribute a collection of JavaServer Pages, Java Servlets, Java classes, XML files, static web pages, and other resources that together constitute a web application.

wire
All the underlying components that are responsible for physically sending or receiving a message on the web.

WSDL
Web Service Description Language. WSDLs are XML files containing all the information relating to services that are available at a particular location on the internet.

XML
eXtensible Mark-up Language

XSD
XML Schema Definition

Index

Special Characters

B

C

D

G

H

I

J

L

M

N

P

R

S

T

U

W

X