Web Services Technologies: State of the Art

Definitions, Standards, Case Study

WORKING PAPER

ABDALDHEM ALBRESHNE, PATRIK FUHRER, JACQUES PASQUIER

September 2009

Abstract

The concept of web services (WS) has gained great importance during the last few years. It has become essential to make applications broadly available in the World Wide Web. WS could be especially useful for the creation of dynamic e-business applications and to allow Java EE and .NET technologies to interoperate. New WS standards have been developed through the cooperation of several corporations. Numerous existing concepts such as business process management, security, directory services, routing and transactions are being adapted for WS. The aim of this paper is to provide an overview and an analysis of recent developments and standards in the field of web services as well as to discuss the benefits they offer. In addition, the paper will look at the creation of WS applications in general and examine alternative technologies for deploying web services, e.g. RESTful web services.

Keywords: Web Services, REST, SOAP, HTTP, SOA, WSDL, Services Composition, Workflow, Services Oriented.

Table of Contents

1 Introduction

1.1 Motivation

Forty years ago, computers began to be connected to the Internet and data transfer among computers was already common. Since then, Internet has evolved to form a huge information space, in which users can move transparently from one machine to another. In the field of application programs, a similar development is ongoing. Distributed computing has been used as long as there have been computer networks. But at present, distributed applications are increasingly viewed and constructed as one vast computing medium. Applications which interact between different machines to provide orchestrated services have now been deployed on a large scale. This evolution is allowed by new protocols built upon HTTP that are designed to enable interaction between programs [Erl06].

The existing distributed computing solutions (CORBA [*OMG09*], Java RMI [*Sun09*]) imply tight coupling between various components in a system. The high level of coordination and shared context among business systems from different organizations needed by those solutions makes them unreliable for open, low-overhead ubiquitous B2B e-business. Systems composed by loosely coupled, dynamically bound elements are much more flexible and have therefore better chances to dominate the next generation of information systems [*Got09*]. These distributed pieces of software-called services in the current jargon are spread across many different machines and new web service frameworks that exploit the new infrastructure are deployed by companies.

The object of this paper is to explain the principles and best practices of web services computing. It presents the concepts, architectures, theories, techniques, standards, and infrastructure necessary for web services and related disciplines.

1.2 Organization of the document

The paper is organized as follows:

- Chapter 1 introduces our motivation, the key trends and architectures in modern computing that motivate how and why services are emerging.
- Chapter 2 provides a good background about the techniques and the methodologies for describing web services as well as a discussion of several WS-* specifications (security, addressing, ...).
- Chapter 3 gives an overview about important topics such as SOAP, WSDL and service discovery.
- Chapter 4 is about Restful web services and compares them with the web services described in Chapter 2.

- Chapter 5 is the conclusion of this paper.
- Chapter 6 presents a concrete case study for web services.
- Chapter 7 presents a concrete case study for RESTful web services.

1.3 Enterprise Architecture

Application architecture is an essential instrument for an application development team. The degree of abstraction used in the documentation of application architecture varies from one organisation to another. While some provide only highly abstract physical and logical representations of the technical patterns, others include more detail, such as common data models, communication flow diagrams, application-wide security requirements, and aspects of infrastructure. An organization often uses distinct application architectures for different solution environments. For example, an organization that implements .NET as well as Java EE solutions would probably define separate application architecture specifications for each [Erl06].

In larger IT infrastructures, we need to define a high level architecture. These specifications will help to control and manage IT infrastructure when numerous, disparate application architectures co-exist and sometimes even integrate. In such a heterogeneous context, the underlying hosting platforms must be able to meet complex demands. Further, enterprise architectures often contain a long-term vision of how the organization plans to evolve its technology and environments [Erl06].

1.3.1 Client-Server Architecture

In its general meaning, the term "client-server" designs an environment in which pieces of software request or receive information from another. This condition is met by virtually every variation of application architecture that ever existed. Yet the meaning of the industry term "client-server architecture" is more specific: it refers to a particular generation of environments in which the client and server had each particular functions as well as different implementation characteristics. Generally, this architecture was composed by multiple fat clients where each of them needed to connect to a database on a central server. Client-side software hosted the essential part of the processing, including all presentation-related and most data access logic (see Figure 1.1). In addition, these clients were supported by one or several servers which hosted RDBMSs [Erl06].

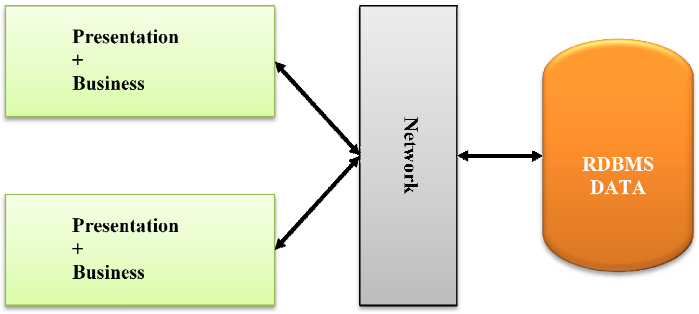


Figure 1.1 *A typical two-tier client-server architecture (inspired from* **[Erl06]***)*

1.3.2 **Distributed Internet Architecture**

Component-based applications became popular because they provided a better alternative to the costly and inflexible two-tier client server architecture. The new multi-tier client-server applications as shown on Figure 1.2 divide the monolithic client executable into components designed to different degrees of compliance with object orientation. Applications can be deployed more easily when the application logic is located in numerous components (some situated on the client, others on the server) because the logic is essentially centralised on servers. Sharing and managing pools of database connections by server-side components located on special applications servers reduces concurrent usage on the database server. These improvements brought also disadvantages with them: higher complexity and more costly development and administration processes. Creating applications able to treat concurrent and multiple requests is even more complex [Erl06].

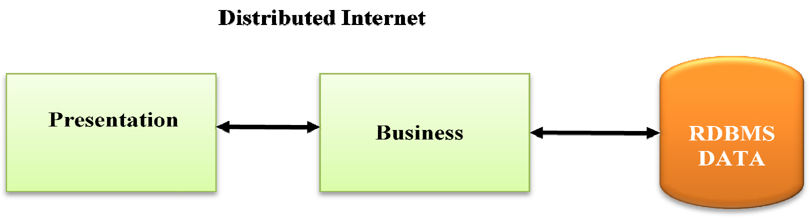


Figure 1.2 *A typical multi-tier client-server architecture (inspired from* [Erl06]*)*

Moreover, the client-server remote procedure call (RPC) [*NWG88*] has partially replaced the client-server database connections. RPC technologies like CORBA [*OMG09*] and DCOM [*Mic09*] enabled remote communications between components distributed between clients and servers. Problems appeared which were similar to those implied by client-server architectures, such as resources and persistent connections management. Additionally, the maintenance effort had to be increased due to the middleware layer. Servers and transaction monitors needed much attention in large environments.

1.3.3 **Web Services Architecture**

"*Web Services, at a basic level, can be considered a universal client/server architecture that allows disparate systems to communicate with each other without using proprietary client libraries*" [Mye09].

The Web Services architecture is a new approach for e-business architectures. A progressing transformation from object-oriented systems toward systems of services can be observed. Web Services systems enable a high level of decoupling as well as dynamic binding of services. Such systems are composed by services which contain behaviour and messages. Services are found by applications using service discovery [*Got09*].

1.3.4 **Service-Oriented Architecture (SOA)**

SOA presents a new method to create distributed applications where basic services can be published, discovered and bound together so as to build more complex composed services representing greater added value. Applications interact with services through an interface endpoint and not at the implementation level. Thus, applications become more flexible due to their ability to interact with any implementation of a contract [Pap08].

Only the implementation through multiple solution platforms allows retrieving maximum advantage of SOA. In this way, it is possible to obtain a maximal return on the investment by providing reusable and interoperable services based on a vendor-neutral communications environment without implying that the whole enterprise has to become service-oriented. SOA causes a great positive impact by the features and characteristics it offers [Erl06].

2 Web Services

2.1 Web Services definition

Different books and organizations give several complementary definitions of Web Services. Some of them are given here:

"*A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. Because all communication is in XML, web services are not tied to any one operating system or programming language--Java can talk with Perl; Windows applications can talk with Unix applications*" [*Cer02*].

"*Web Services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or Web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML*" [*IBM09*].

"*A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards*" [*Wha09*].

Thus, web services are platform-independent, based on XML messages. The idea is to distribute services over the Internet and to make them available for clients. These services can be implemented with any language and can be invoked as well as composed.

2.2 Web Services Architecture

There are three major roles within the web services architecture:

- Service provider
  This is the provider of the web service. The service provider builds the service and makes it available on the Internet for consumers.

- Service requestor
  This is any consumer of the web service. The requestor invokes an existing web service by opening a network connection and sending an XML-SOAP (see Subsection 3.1) request.

- Service registry
  It is a centralized directory of services. The registry is used as a central place where providers or developers can publish new services or find existing ones. It therefore serves as a centralized clearinghouse for companies and their services.

Figure 2.1 gives a logical view of web services by illustrating the relationship between the web services roles and operations. First, the web service provider publishes its web services with the discovery agency. Next, the web service consumer looks for desired web services using the registry of the discovery agency. Finally, the web services client invokes the web services by using the information obtained from the discovery agency.

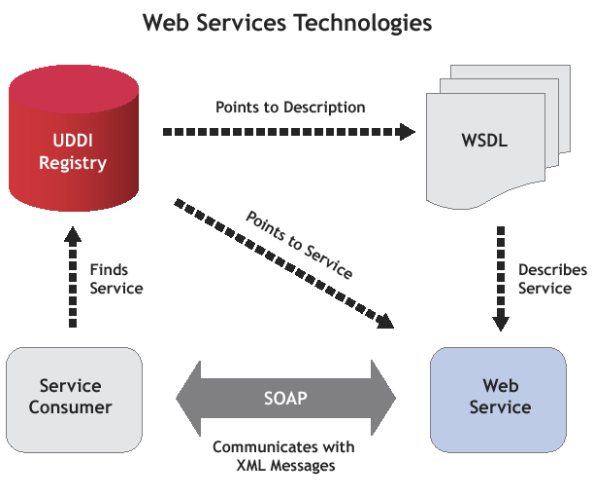


Figure 2.1 *(taken from* **[***Dmr02***]***)*

2.2.1 **The Web Service Protocol Stack**

The architecture of a Web Services stack varies from one organization to another. The number and complexity of layers for the stack depend on the organization. Web services are built by using various related technologies. Figure 2.2 illustrates the stack of specific, complementary standards on which web services are generally based on.

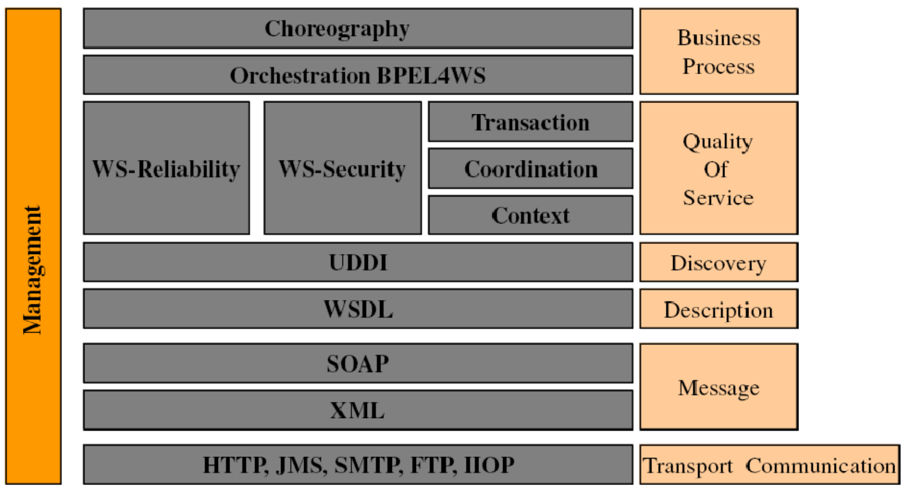


Figure 2.2 *The Web Services technology stack (inspired from* **[Pap08]**)

- Service transport
  The service transport layer delivers messages between applications. This layer usually implements hypertext transfer protocol (HTTP) [*NWG99*], Simple Mail Transfer Protocol (SMTP) [*Wik092*], file transfer protocol (FTP), and newer protocols, such as Blocks Extensible Exchange Protocol (BEEP) [*Wha09*].

- XML messaging
  This layer is responsible for encoding messages in a common XML [*W3C08*] format so that messages can be understood at either end. Currently, this layer includes XML-RPC and SOAP [*Wha09*].

- Simple object access protocol (SOAP)
  SOAP is a simple XML-based messaging protocol responsible for transferring data between different web services. It is built using XML and relies on common Internet transport protocol like HTTP to transport its messages. SOAP allows communication among interacting web services by implementing a request/response model and using HTTP to access networks protected by firewalls, which do not currently prevent HTTP and FTP service requests [Pap08]. See Section 3.1 for further information.

- Service description WSDL
  The purpose of this layer is to define the public interface of a specific web service. Currently, service description is realised through the Web Service Description Language (WSDL) which is based on XML [*Wha09*]. See Section 3.2 for further details about WSDL.

- Service discovery
  The service discovery layer registers services into a common repository and provides an easy publish/find mechanism. This layer is often implemented via Universal Description, Discovery, and Integration (UDDI) [*Wha09*]. But, the problem of service discovery is much discussed and the UDDI standard seems not to be used in large scale deployments. See Section 3.3 for more details.

- Service orchestration
  The topmost service orchestration layer is in charge of the execution logic of web services based applications by determining their control flows (e.g. conditional, sequential, parallel and exceptional execution). This layer enables enterprises to define and realise complex business processes [Pap08].

With the ongoing evolution of web services, it is possible that more layers will be added to these different layers, for example specifying quality of services (QoS) aspects, may be added to the technology stack described above as well.

2.2.2 **Additional specifications, WS***

Several Web services specifications have recently been developed or are still being developed in order to enlarge Web Services functionalities. These specifications are generally referred to as WS-*. Below, some of these are briefly described.

- WS-Security
  The WS-Security specification determines the use of XML Encryption and XML

Signature in SOAP so as to secure communication. It is used either as an alternative or an extension to using HTTPS to secure the message exchanges. As mentioned in [Erl06], the principal aspects of security addressed by these specifications are identification, authentication, authorization, integrity, confidentiality and non-repudiation.

- WS-Reliability
  The purpose of the WS-Reliability specification is to provide an environment that insures the delivery of a SOAP message or the reporting of a failure condition [Erl06].

- WS-Transaction
  The Web Services Transactions specifications enable Web services domains to interoperate and allow composing transactional qualities of service into Web services applications. These specifications describe an extensible coordination framework and specific coordination types for: Short duration, ACID transactions (WS-AtomicTransaction) and Longer running business transactions (WS-BusinessActivity). The interested reader is referred to [*IBM04*] for further details.

- WS-addressing
  The WS-addressing specification implements addressing extensions which are based on endpoint references and message information headers. Endpoint references allow the identification in a standardized way of a specific instance of a web service. Message information headers provide message exchange properties to a specific message, conveying interaction semantics to recipient services [Erl06].

- WS-policies
  The WS-Policies specification serves to attach features/characteristics (e. g. rules, behaviours, requirements and preferences) to Web resources, principally web services [Erl06].

- WS-Metadata exchange
  WS-Metadata exchange specification enables service requestors to issue requests to obtain metadata (such as WSDL, Schema, Policy Meta information) from services providers. This specification helps to improve the service description discovery process [Erl06].

- WS-Notification and eventing
  WS-Notification is a group of specifications in the context of the WS-Resource environment. These specifications allow event driven programming between web services by providing a protocol for web services to subscribe to another web service, or to accept a subscription from another web service [*Wik09*].

2.3 How do web services work?

The programmer builds a web service using a specific programming language. This service is published using a WSDL interface. This service can be invoked by a consumer “client” using this interface. Web services are presented to clients as a set of operations that provide business logic on behalf of the provider. Web services must be deployed on a server container to be available for consumers. On the client side, a remote object that represents the remote service must be generated. This allows clients to invoke the operations defined on the server side. The developer has not to care about creating or parsing SOAP messages. That task is performed by the web service's APIs runtime system. .Net web services for example can be invoked by any web service client and vice versa.

A Java-based Web Service built and deployed on Solaris operating system can be accessed from Visual Basic program which runs on Windows. Any language can also be used to realise new web services. These web services are invoked from any web application which is implemented using any other programming language and runs on any operating system.

An Example

Let us consider a simple application to sell cars. The buyer uses a client application built with Visual Basic or JSP to get information about a specific vehicle.

The processing logic for this application is written in Java and resides on a Linux machine which also interacts with a database to store the information.

The steps illustrated above are:

- The client calls the web service method through a remote object which represents that web service.
- The client program bundles the car information request into a SOAP message.
- This SOAP message is sent to the Web service as the body of an HTTP POST request.
- The Web service unpacks the SOAP request and converts it into a command that the application can understand. The application processes the information as required and responds with details concerning the selected car.
- Next, the Web Service packages up the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
- The client program unpacks the SOAP message to obtain the results of the car details process (the return result of the called method).

2.4 Web services characteristics

From [Pap08] and [*Wha09*] one can derive the following characteristics of web services:

- XML-based
  Web Services rely on XML for data representation and transportation. The use of XML avoids any network, operating system or platform binding.
- Loose coupling
  There is no direct tie between a web service and its user. Alterations of the WS

interface do not deteriorate the user's capability of interacting with the service. Whereas in a tightly coupled system, the client and server logic are closely bound to each other, implementing a loosely coupled architecture facilitates software system management and helps the integration of different systems.

- Ability to be synchronous or asynchronous
  Ties between the client and the execution of the service are referred to as synchronicity. In synchronous invocations, the client sends his request and then waits for the response without being able to execute other operations during this period. In contrast, asynchronous invocations allow clients to request a service and then immediately execute other operations without waiting for the result ("fire and forget" model). Asynchronous capability is a crucial factor to make loosely coupled systems possible.

- Supports Remote Procedure Calls (RPCs)
  Web services enable clients to invoke methods and operations on remote objects using an XML-based protocol (SOAP). Input and output parameters which a web service must support are made available through remote procedures. A web service supports/implements RPC either by providing services of its own, equivalent to those of a traditional component, or by translating incoming invocations into an invocation of an EJB or a .NET component.

- Supports document exchange
  A central aspect of XML is that it is capable to represent data, simple and even complex documents in a generic way. This transparent exchange of documents supported by Web services contributes to make business integration easier.

2.5 Benefits of using Web Services

As also mentioned in [Pap08] and [*Wha09*], using web services offers many interesting benefits such as:

- Reusability
  A Web service is a component which can be remotely accessed using HTTP. Web Services provide a means to make a pre-existing code available through Internet. As a result, the program's functionalities can be invoked by other applications.

- Interoperability
  Web Services enable the share of data and the communication between different applications. For example, .NET applications can interact with Java web services and vice versa. Thus, applications become platform and technology independent.

- Standardized Protocol
  Web Services uses industry standard protocol for the communication. All the four layers (Service Transport, XML Messaging, Service Description and Service Discovery layers) use well defined protocol in the Web Services protocol stack. This standardization of protocol stack gives the business many advantages like wide range of choices, reduction in the cost due to competition and increase in the quality.

- Automatic Discovery
  Web Services automatic discovery mechanism allows businesses to easily find the

Service Providers and retrieve web service description that have been previously published. The first step to access a web service is through the discovering process. Client queries the service registry for web service matching his needs. The query contains search criteria like service type or preferred price. After the discovery process is completed, the client can access the web service.

3 Web Services Technologies

This chapter describes the concepts related to SOAP, WSDL and Service Discovery. We will supply introductory descriptions of the primary elements provided by these technologies.

3.1 Messaging Protocol

Traditional distributed object communication protocols, such as CORBA [*OMG09*], DCOM [*Mic09*], Java/RMI [*Sun09*], and others, present severe weaknesses for client-to-server communication. These limitations appear most evidently when clients' machines are distributed over the Internet. The conventional distributed communication link must be implemented under a distributed object model and would need the deployment of libraries. To solve such problems, the Simple Object Access Protocol (SOAP) was created. SOAP allows interoperability between a wide range of programs and platforms. In this way, existing applications can be accessed by a broader range of users [Pap08].

3.1.1 SOAP: Simple Object Access Protocol

The messaging protocol currently used by web services is SOAP. SOAP is designed to enable separate distributed computing platforms to interoperate. This aim is accomplished by following the same principles as other successful web protocols: simplicity, flexibility, firewall friendliness, platform neutrality as well as XML based messaging. Instead of representing a new technological innovation, SOAP merely suggests a manner to codify the usage of existing internet technologies in order to standardize distributed communications over the Web. SOAP is usually exchanged through HTTP which is used by Web browsers to access Web resources. HTTP constitutes an efficient way of sending and receiving SOAP messages [Pap08].

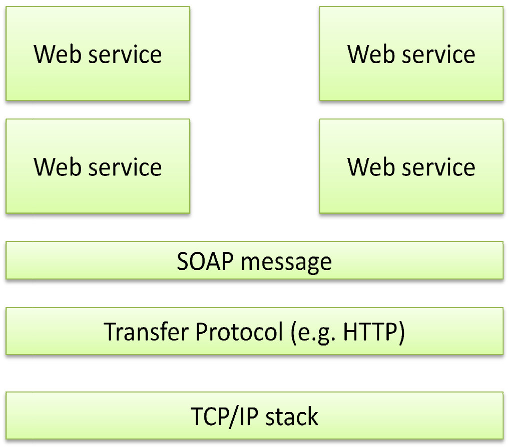


Figure 3.1 ***The Web services communication and messaging network (inspired from*** **[Pap08]*)***

Figure 3.1 shows that SOAP messages can be built on different protocols such as HTTP to transport messages. SOAP's role is to define how a message is formatted but not how the message is delivered. HTTP is the most commonly used transport protocol. However, also other protocols, such as SMTP or FTP may be used [Pap08].

3.1.2 Structure of SOAP Messages

The structure of SOAP messages is relatively simple. It consists of a header and a body with some fault sections, all defined in an envelope as shown in Listing 3.1.

```
<Envelope xmlns=http://www.w3.org/2001/12/soap-envelope>
<Header>
  ...
</Header>
<Body>
  ...
  <Fault>
    ...
  </Fault>
</Body>
</Envelope>
```

Listing 3.1 ***SOAP message structure***

- **Envelope:** a mandatory root element which defines the beginning and the end of the message. It contains an optional <Header> section and a mandatory <Body> section. All elements of a SOAP envelope are defined using a W3C XML schema.

- **Header** is an optional element which defines any optional attributes of the message. The role of the Header is to host extensions separately from payload without changing the fundamental structure of SOAP. Due to this separation, additional features and functionalities can be added e. g. security, transactions, QoS attributes without modifying the specification [Pap08].

- **Body** is a mandatory element which envelops the message to send in XML format. This element holds either the requested data (response) or an error message (fault). Requested data represent the application-specific data exchanged with a Web service. This information can be XML data or parameters to a method call. Inside the SOAP <Body>, the method call information as well as its related arguments are defined, the response to a method call is placed and error information can be saved [Pap08].

- **Fault** is an optional element that provides information about errors that occurred while processing the message.

3.1.3 The SOAP communication model

The web service communication model describes how to invoke web services and relies on SOAP. The SOAP communication model is defined by its communication style and its encoding style. SOAP supports two possible communication styles: RPC and document (message).

- RPC- style Web services
  RPC-style web services are used as remote objects on the client application side. Clients send their request as a method call. The method returns a response message [Pap08]. This information is formatted as sets of XML elements loaded into a SOAP message as shown in Listing 3.2.

```
<Envelope xmlns="http://www.w3.org/2001/12/soap-envelope"
<Header>
  ...
</Header>
<Body>
  <GetProductPrice>
    <product-id>4562</product-id>
  </GetProductPrice>
</Body>
</Envelope>
```

Listing 3.2 ***RPC-style web services***

- Document (message)-style Web services
  SOAP supports documents exchange for any kind of XML data. The client sends the whole document to the provider instead of sending a set of arguments [Pap08]. See Listing 3.3.

```
<soap:Envelope xmlns:SOAP=http://www.w3.org/2001/12/soap-envelope>
<soap:Body>
  <pourchaseOrder orderDate="2009-05-20" xmlnso=http://www.amzon.com/POs>
    <po:accountName>Ricard</po:accountName>
    <po:accountNumber>1234</po:accountNumber>
    <po:book>
     <po:title>J2EE web services</po:title>
     <po:quantity>300</po:quantity>
     <po:price>24.5</po:price>
    </p:book>
  </pourchaseOrder>
</soap:Body>
</soap:Envelope>
```

Listing 3.3 ***Document-style web services***

3.1.4 Advantages and disadvantages

As presented in [Pap08], there are several advantages of using SOAP:

- Simplicity: SOAP is simple as it uses XML that is well structured and easy to parse.
- Portability: SOAP is platform-independent and thus portable.
- Firewall-friendliness: SOAP is capable of getting past firewalls which are totally blocking for other protocols. This is possible due to using the HTTP protocol.
- Use of open standard: SOAP is based on the open Standard XML to format data. As a consequence, SOAP becomes easily extendable and well supported.
- Interoperability: SOAP relies on open instead of vendor-specific technologies and thus enables distributed interoperability and loosely coupled applications.
- Resilience to changes: It is unlikely that future modifications of SOAP infrastructure will have any impact on applications using the method, as long as no significant serialization changes are made to the SOAP specification.

There are, however, several aspects of SOAP which can be viewed as disadvantageous. These include the following:

- SOAP is stateless which implies that the requesting application has to reintroduce itself to other applications if additional connections are needed as if it was connected for the first time.

- SOAP serializes by value and does not serialize by reference.
- SOAP was at the beginning mainly based on HTTP. This imposed a request/response architecture which did not suit all situations. As HTTP is relatively slow, the performance of SOAP was affected.

3.2 Describing Web Services

In [*W3C09*] the following definition for the Web Services Description Language (WSDL) is provided:

"*WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. 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 (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate*".

Thus a WSDL document defines the point of contact (endpoint) for a service provider. It contains a formal definition of the service interface so that requestors who intend to invoke the service provider know how to build the messages. Additionally, it provides the physical location of the service [Erl06].

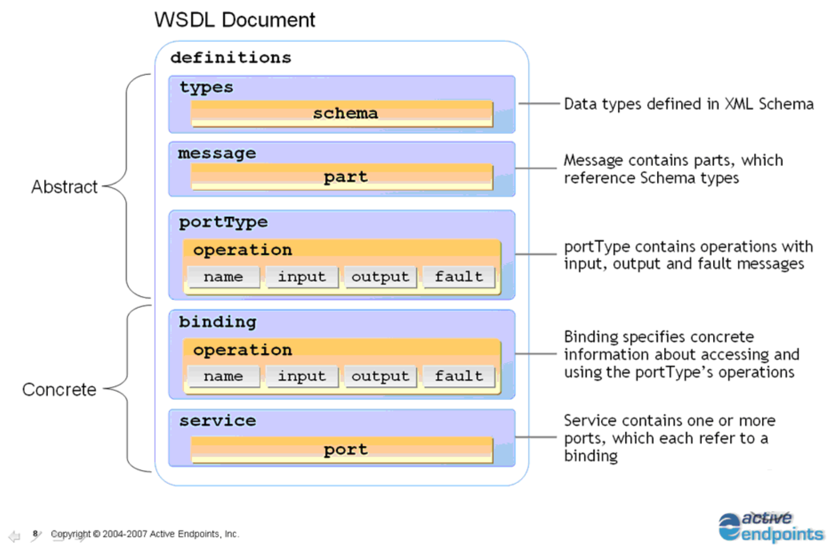
Let us dig deeper into how the service description document itself is organized. According to [Erl06] a WSDL service description (see Figure 3.2) can be separated into two parts:

1. Abstract description
   An abstract description portion provides information about the interface characteristics of the Web service without any description or details concerning the technology used to implement the web services or the one used to send messages. As a result, the integrity of the web service is maintained notwithstanding a change of the underlying technology platform. It consists of several parts including types, message and port type [Erl06].
2. Concrete description
   The concrete description portion of the WSDL file defines the connection to the real implementation of the web services where is its logic. This description contains three parts: binding, port and service [Erl06].

A WSDL document uses the following elements in the definition of network services as defined by W3C [*W3C09*]:

- **Types**: "*a container for data type definitions using some type system (such as XSD)*".
- **Message**: "an *abstract, typed definition of the data being communicated*".
- **Port Type**: "*an abstract set of operations supported by one or more endpoints*".

- **Operation**: "*an abstract description of an action supported by the service*".
- **Binding**: "*a concrete protocol and data format specification for a particular port type*".
- **Service**: a collection of related endpoints.
- **Port**: "*a single endpoint defined as a combination of a binding and a network address*".



WSDL Document
definitions
types
schema
message
part
portType
operation
name
input
output
fault
binding
operation
name
input
output
fault
service
port
Abstract
Concrete
Data types defined in XML Schema
Message contains parts, which reference Schema types
portType contains operations with input, output and fault messages
Binding specifies concrete information about accessing and using the portType's operations
Service contains one or more ports, which each refer to a binding
Copyright © 2004-2007 Active Endpoints, Inc.
active endpoints

Figure 3.2 *WSDL file structure (taken from* [Act09] *)*

```
<definitions.... >
   <types>
       <xsd:schema .... />
   </types>
   <import namespace="http://www.xml.com/tls/schema"
        Location=http://www.xml.com/tls/schema/car.xsd/>
   <message name="getID">
        <part  type="xsd:intger"/>
   </message>
   <portType name="CarInterface">
       <documentation>
          Get Car Details operation.
       </documentation>
       <operation name="getCarDetails">
          <input message="tns:rentCar"/>
          <output message="tns:rentCarResponse"/>
       </operation>
       <operation name="UpdateCarDetails">
          ....................
       </operation>
   </portType>
   <binding name="CarBinding" type="tns:CarInterface">
         <soap:binding style="document"
             Transport=http://schemas:xmlsoap.org/soap/http/>
         <operation name="GetCarDetails">
          ...................................
         </operation>
   </binding>
   <service name="CarService">
         <port binding="tns:CarBinding" name="CarPort">
           <soap:address location=http://www.localhost:8080/car/>
         </port>
   </service>
</definitions>
```

Listing 3.4 *WSDL elements*

Each of these parts relates to corresponding elements that are defined in the WSDL specification [*W3C09*]. Let us describe in brief the syntactical implementation of these elements (shown in Listing 3.4) as presented in [Erl06].

- The `definitions` element
  Definitions element encapsulates the entire WSDL document and it usually contains several namespaces definitions.

- The `types` element
  Types serve as a container which contains all abstract data types. It can contain XML schema that define other data type definition [*W3C09*].

- The `import` element
  This element allows importing parts of the WSDL definition and XSD schemas.

- The `message` and `part` elements
  Message definition determines the payloads of messages which are sent or received by a web service. Messages are composed of `<part>` elements. Each part stands for an instance of a particular type.

- The `portType`, interface and `operations` elements
  `portType` describes several abstract operations with the outgoing and incoming messages.

- The `documentation` element
  With the documentation element, it is possible to add descriptive, human readable annotations within a WSDL definition. This information facilitates the discovery of the service within a service registry.

- The `input` and `output` elements
  For every operation, there are input and optionally output child elements. An operation uses inputs and outputs to request and return messages.

- The `binding` element
  The binding element determines which communications protocol can be used to access and interact with WSDL. It defines where the service is located or to which network address the message has to be sent.

- The `service` and `port` elements
  The service element provides a physical address of the service. It contains the port element that defines this location information.

3.3 Registering and Discovering Web Services

A directory service serves as information point for components and participants (e.g. applications, agents, web services, people, objects, requestors and procedures) which enables them to locate each other. Directories organize and provide web services' location and description details and publish them to any requestor. There are two kinds of directories available: name servers referred to as "white pages" in which entries are defined and found by their name as well as so called "yellow pages", where entries are defined and found by their characteristics and functions [Sin05].

Two main standards for directories have already been defined: ebXML (Electronic Business using XML) [*ebX08*] registries and UDDI (Universal Description, Discovery and Integration) registries. Inconveniently, both of them lack semantic descriptions and semantic searching on functionality. The searches can be done only by keywords, like a service's name, provider, location or business category. In contrast with UDDI registries, ebXML registries permit at least SQL-based queries on keywords [Sin05].

3.3.1 Electronic Business Extensible Markup Language (ebXML)

ebXML is used as a global electronic market place where enterprises of any size, anywhere can determine each other electronically and clear semantics by exchanging XML messages. This exchange is based on mutual trading protocol agreements. In other words, ebXML Registry's goal is to provide generic, extensible, secure and federated information management [*Tut09*].

ebXML architecture enables to define business processes and their related messages. It gives as well a way to register and discover business process sequences. Moreover, ebXML allows defining company profiles and a uniform message transport layer [*Tut09*].

3.3.2 Universal Description, Discovery and Integration (UDDI)

[Sin05] defines UDDI as "*a platform-independent, Extensible Markup Language (XML)-based registry for businesses worldwide to list themselves on the Internet. UDDI is an open industry initiative, sponsored by the Organization for the Advancement of Structured Information Standards (OASIS), enabling businesses to publish service listings and discover each other and define how the services or software applications interact over the Internet*".

UDDI supports three kinds of services descriptions: white-pages, yellow-pages, and green-pages services.

- *White pages* contains the following information fields:
    - Business name.
    - Text description: a list of Multilanguage text strings.
    - Contact information: names, phone numbers, and web sites.
- *Yellow pages* provide the business categories organized as the following major taxonomies:
    - Industry, a six-digit code for classifying companies.
    - Products and services.
    - Geographical location for countries and region code.

- *Green pages* contain the information business used to describe how other businesses can conduct electronic commerce with them. It is a nested model comprising business processes, service descriptions, and binding information

Data structures describe details about organizations (for example their services, implementation technologies, relationships to other businesses). UDDI defines five principal data structures [Sin05]:

- A businessEntity represents the business or organization that provides the web services.
- A businessService represents a web service.
- A BindingTemplate represents the technical binding of a web services to its access point, its URL and to tModels.
- A tModel represents a specific kind of technology, such as SOAP or WSDL.
- A publisher Assertion represents a relationship between two business entities.

There are *two kinds of APIs* that allow programmatic access to a UDDI registry: firstly the *inquiry API* which is used to retrieve information from a registry and, secondly, the *publish API* which is used to store information in a registry. In contrast with the inquiry API, the publish API needs authenticated access.

3.3.3 **ebXML vs UDDI**

Below is a high level summary comparing the design centers of the two standards UDDI and ebXML Registry [*SUN05*].

UDDI Registry	**ebXML Registry and Repository**
Contains no repository. Unable to store content. Capable only of storing metadata about (or pointers to) content.	Has an integrated Registry and Repository. Able to store content as well as metadata.
Design center lists businesses and services, similar to yellow/white pages.	Design center provides secure, federated information management of any type of artifact.
Protocols and information model is focused and specific.	Protocols and information model is generic and extensible.
Supports multi-registry topologies using replication of every transaction to all participating registries.	Supports multi-registry topologies using loosely coupled federations with optional selective replication.

Table 3.1 ***ebXML vs UDDI***

4 RESTful Web Services

4.1 What is a RESTful web service?

REST the abbreviation of “Representational State Transfer” designates an architecture style used to create networked applications. The terms "representational state transfer" and "REST" were introduced in 2000 in the doctoral dissertation of Roy Fielding, [Fie00] one of the principal authors of the Hypertext Transfer Protocol (HTTP) [*NWG99*] specification. REST uses a stateless, client-server, cacheable communications protocol which is almost always the HTTP protocol. Its original feature is to work by using mere HTTP to make calls between machines instead of choosing complex mechanisms such as CORBA, RPC [*NWG88*] or SOAP [Ric07] [*Elk09*].

In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture. Many aspects of the Internet correspond to the characteristics of a REST-based architecture. Restful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations [*Elk09*].

REST does not offer security features, encryption, session management, QoS guarantees, etc. But these can be added by building on top of HTTP, for example username/password tokens are often used and for encryption, REST can be used on top of HTTPS (secure sockets).

4.2 RESTful & Resources

Characteristic elements of RESTful technology are sources of specific information which are referred to as resources. Each of them is linked to a global identifier, for example a URI in HTTP. These resources are accessed by components of the network (user agents and servers) which communicate through a standardized protocol (e.g., HTTP) and exchange content (representations) of these resources. In order to interoperate with a resource, an application must possess both the resource’s identifier and the required method. On the opposite, there is no need to know the services implementation and system configuration, i.e. whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between the application and the server which hosts the resources. However, the application must be capable of interpreting the data format (representation) returned from the resource, which is often an HTML or XML document, though it may also be an image, plain text, or any other content [*Wik091*].

4.3 RESTful & Messages types

Restful allows the realisation of applications without requiring any new format. Resources can be represented by any format (i.e. HTML, GIF and PDF files). XML can be used to transmit structured data.

4.4 RESTful Principles

The REST architectural style is based on four principles as presented in [Pau08]:

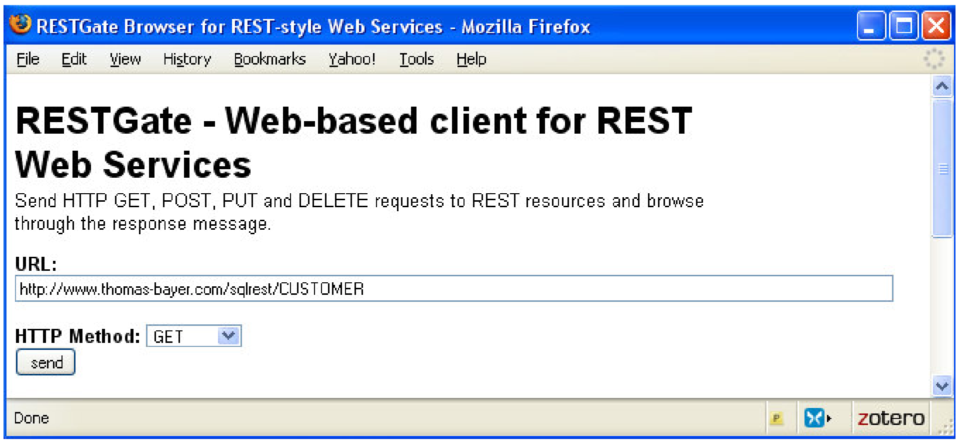
- *Resource identification through URI*. Resources are identified by URIs which provide a service discovery mechanism.
- *Uniform interface*. A fixed set of four create, read, update, delete operations is responsible for the manipulation of resources.
- *Self-descriptive messages*. Resources' content can be presented in several formats (e.g. HTML, XML, plain text, PDF or JPEG). Metadata about the resource can be used to detect transmission errors and perform authentication or access control.
- *Stateful interactions through hyperlinks*. Stateful interaction can be achieved using several techniques, for example rewriting URL, cookies or hidden form fields.

4.5 RESTful Web Service HTTP methods

The examples below provided by [*Bay07*] show how the HTTP verbs are typically used to implement a web service:

- GET method: listing the members of the collection. It lists for example all the customers for a company.

  GET Request: Figure 4.1 shows a GET request to display the customers list



RESTGate Browser for REST-style Web Services - Mozilla Firefox

RESTGate - Web-based client for REST Web Services

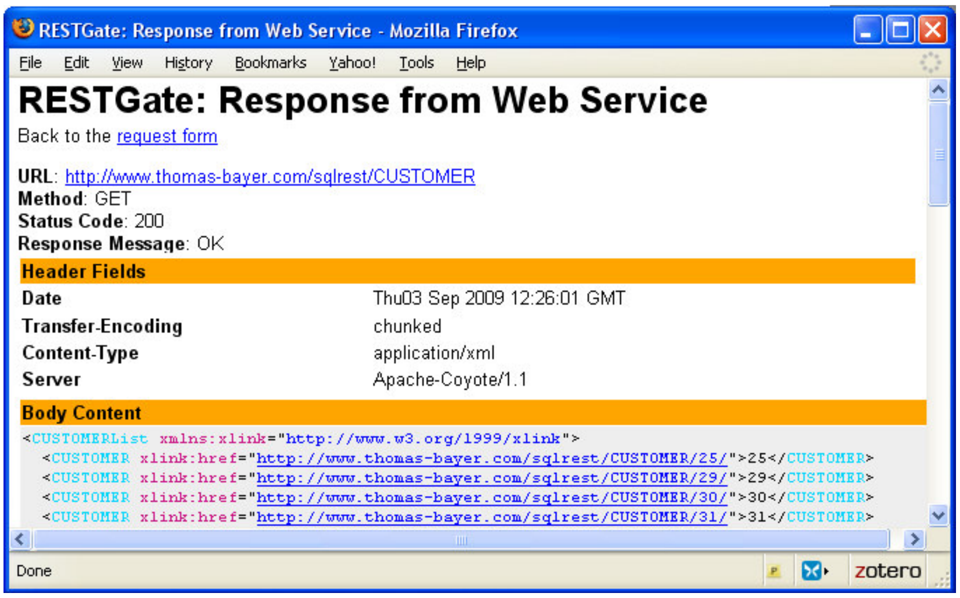
Send HTTP GET, POST, PUT and DELETE requests to REST resources and browse through the response message.

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER

HTTP Method: GET

Figure 4.1 ***Request a list of resources***

GET Response*:* Figure 4.2 shows the response to the request according to Figure 4.1.


RESTGate: Response from Web Service - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
Done
zotero

RESTGate: Response from Web Service

Back to the request form

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER
Method: GET
Status Code: 200
Response Message: OK

Header Fields	
Date	Thu03 Sep 2009 12:26:01 GMT
Transfer-Encoding	chunked
Content-Type	application/xml
Server	Apache-Coyote/1.1

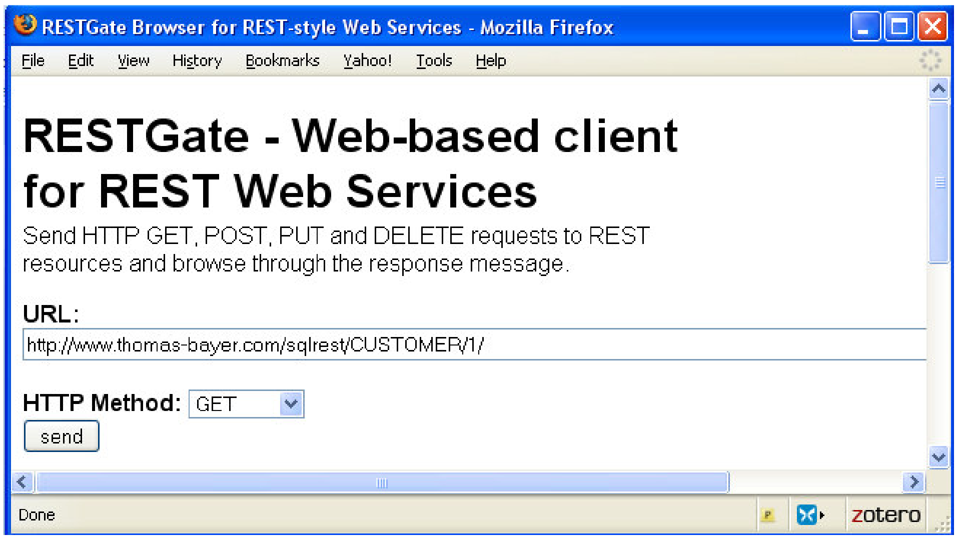
Body Content

```
<CUSTOMERList xmlns:xlink="http://www.w3.org/1999/xlink">
  <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest/CUSTOMER/25/">25</CUSTOMER>
  <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest/CUSTOMER/29/">29</CUSTOMER>
  <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest/CUSTOMER/30/">30</CUSTOMER>
  <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest/CUSTOMER/31/">31</CUSTOMER>
```

Figure 4.2 *Response after requesting a list of resources*

- GET: retrieving the addressed member of the collection. It allows for example to get details about the customer whose customer ID is 1.

  GET Request: Figure 4.3 shows a GET request to display customer 1 details.


RESTGate Browser for REST-style Web Services - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
send
Done
zotero

RESTGate - Web-based client for REST Web Services

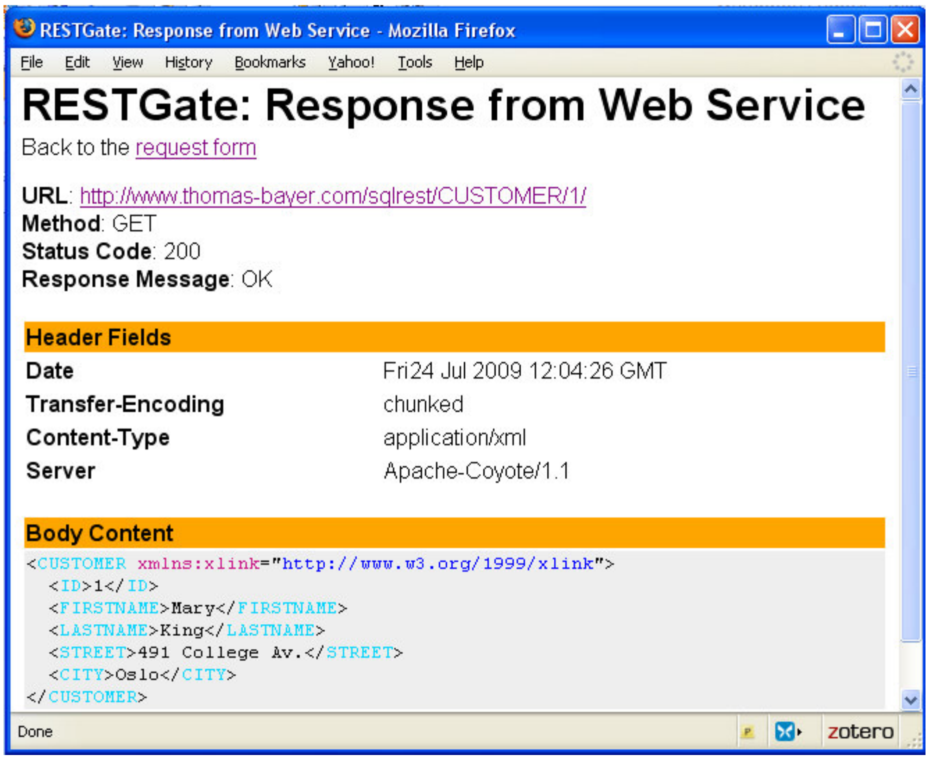
Send HTTP GET, POST, PUT and DELETE requests to REST resources and browse through the response message.

URL:
http://www.thomas-bayer.com/sqlrest/CUSTOMER/1/

HTTP Method: GET

Figure 4.3 *Requesting resource's details*

GET Response: Figure 4.4 shows the response to the request showed in Figure 4.3.


RESTGate: Response from Web Service - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
Done
zotero

RESTGate: Response from Web Service

Back to the request form

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/1/
Method: GET
Status Code: 200
Response Message: OK

Header Fields	
Date	Fri24 Jul 2009 12:04:26 GMT
Transfer-Encoding	chunked
Content-Type	application/xml
Server	Apache-Coyote/1.1

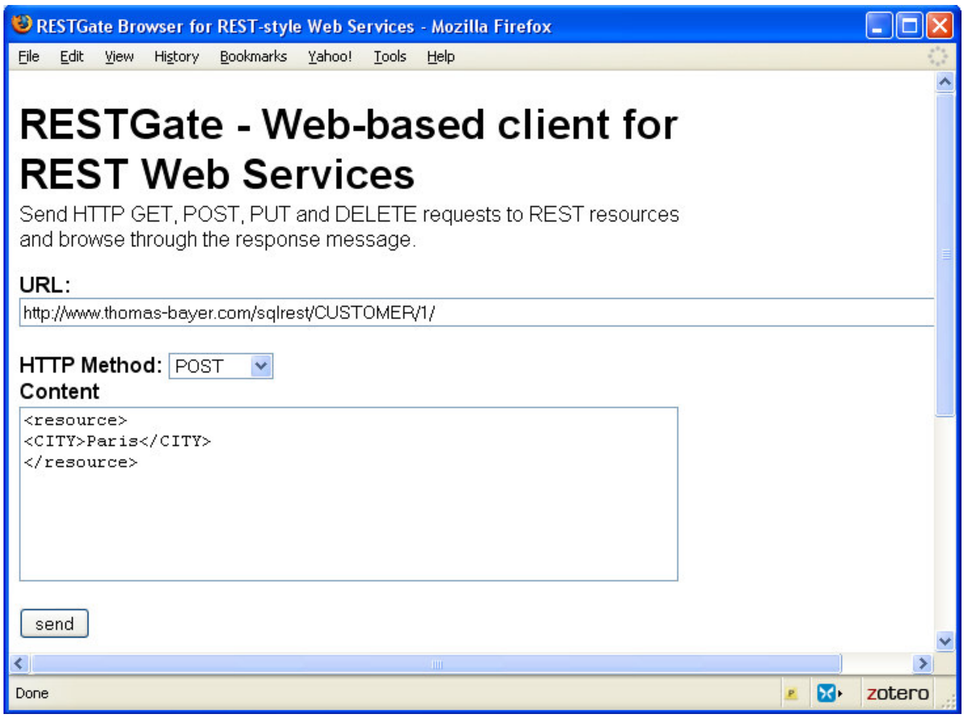
Body Content

```
<CUSTOMER xmlns:xlink="http://www.w3.org/1999/xlink">
  <ID>1</ID>
  <FIRSTNAME>Mary</FIRSTNAME>
  <LASTNAME>King</LASTNAME>
  <STREET>491 College Av.</STREET>
  <CITY>Oslo</CITY>
</CUSTOMER>
```

Figure 4.4 *Response after requesting resource's details*

- POST method: This method updates the addressed member of the collection with a defined ID.

  POST Request: Figure 4.5 shows a POST method to update the customer city attributed to customer ID 1.


RESTGate Browser for REST-style Web Services - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
send
Done
zotero

RESTGate - Web-based client for REST Web Services

Send HTTP GET, POST, PUT and DELETE requests to REST resources and browse through the response message.

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/1/

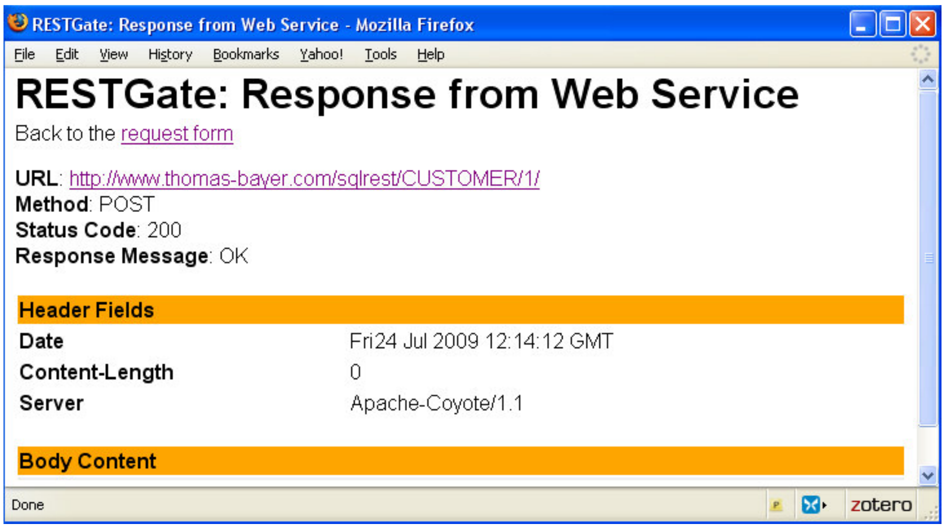
HTTP Method: POST

Content

```
<resource>
<CITY>Paris</CITY>
</resource>
```

Figure 4.5 *Updating a resource*

POST Response*:* Figure 4.6 shows the response to the request in Figure 4.5**.**


RESTGate: Response from Web Service - Mozilla Firefox

RESTGate: Response from Web Service

Back to the request form

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/1/
Method: POST
Status Code: 200
Response Message: OK

Header Fields

Date	Fri24 Jul 2009 12:14:12 GMT
Content-Length	0
Server	Apache-Coyote/1.1

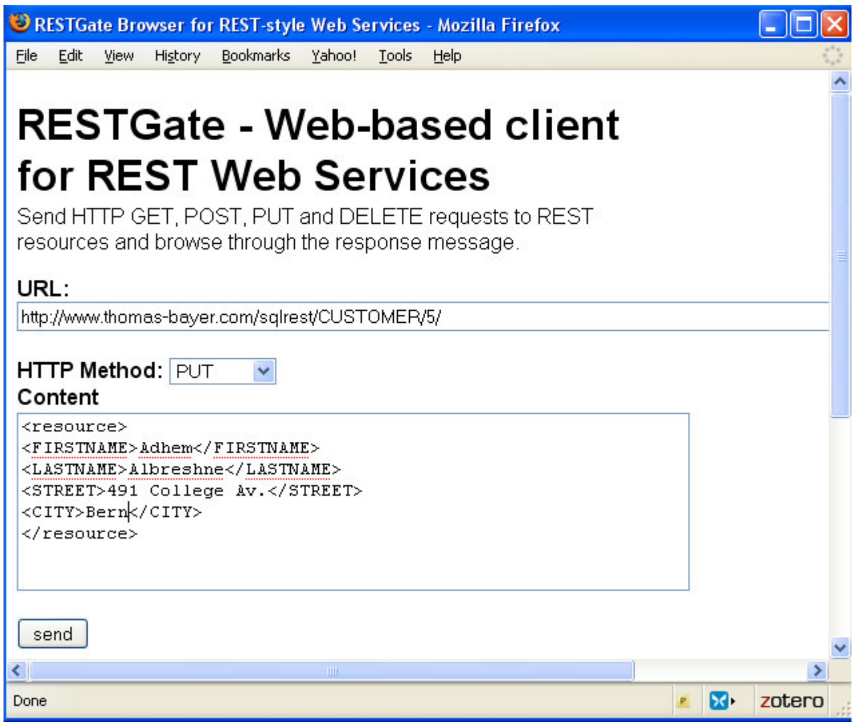
Body Content

Done

Figure 4.6 ***Response after updating a resource***

- PUT method: This method creates a new entry in the collection.

  PUT Request: Figure 4.7 shows the PUT method to create a new customer with customer ID 5.


RESTGate Browser for REST-style Web Services - Mozilla Firefox

RESTGate - Web-based client for REST Web Services

Send HTTP GET, POST, PUT and DELETE requests to REST resources and browse through the response message.

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/5/

HTTP Method: PUT

Content

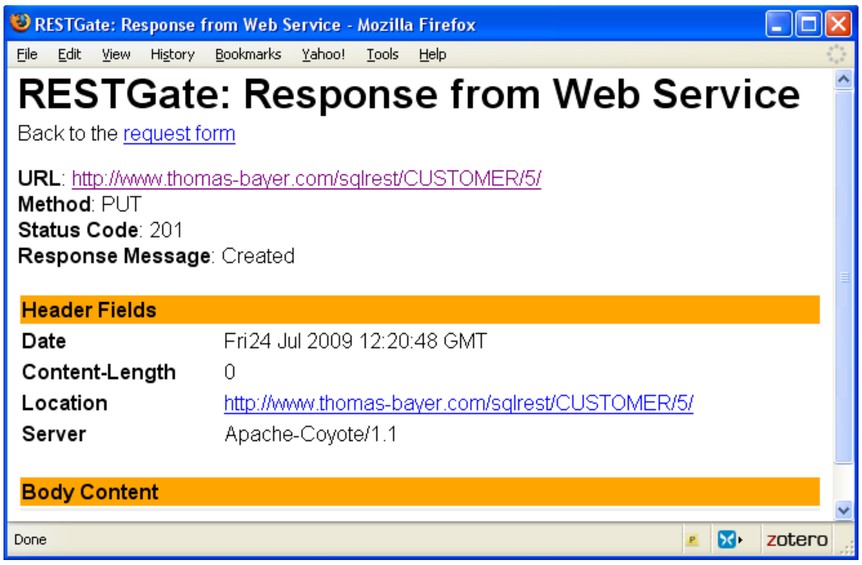
```
<resource>
<FIRSTNAME>Adhem</FIRSTNAME>
<LASTNAME>Albreshne</LASTNAME>
<STREET>491 College Av.</STREET>
<CITY>Bern</CITY>
</resource>
```

send

Done

Figure 4.7 ***Creating a new resource***

  PUT Response: Figure 4.8 shows the response to the request showed in Figure 4.7.


RESTGate: Response from Web Service - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help

RESTGate: Response from Web Service

Back to the request form

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/5/
Method: PUT
Status Code: 201
Response Message: Created

Header Fields

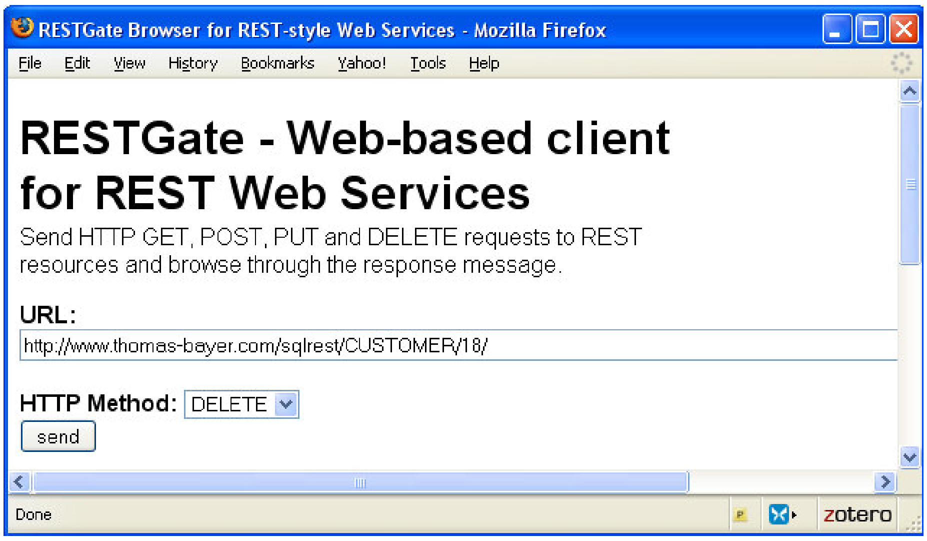
Date	Fri24 Jul 2009 12:20:48 GMT
Content-Length	0
Location	http://www.thomas-bayer.com/sqlrest/CUSTOMER/5/
Server	Apache-Coyote/1.1

Body Content

Figure 4.8 *Response after creating a new resource*

- DELETE Method: This method deletes the addressed member of the collection.

DELETE Request: Figure 4.9 shows the delete method to delete the customer whose customer ID is 18.


RESTGate Browser for REST-style Web Services - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help

RESTGate - Web-based client for REST Web Services

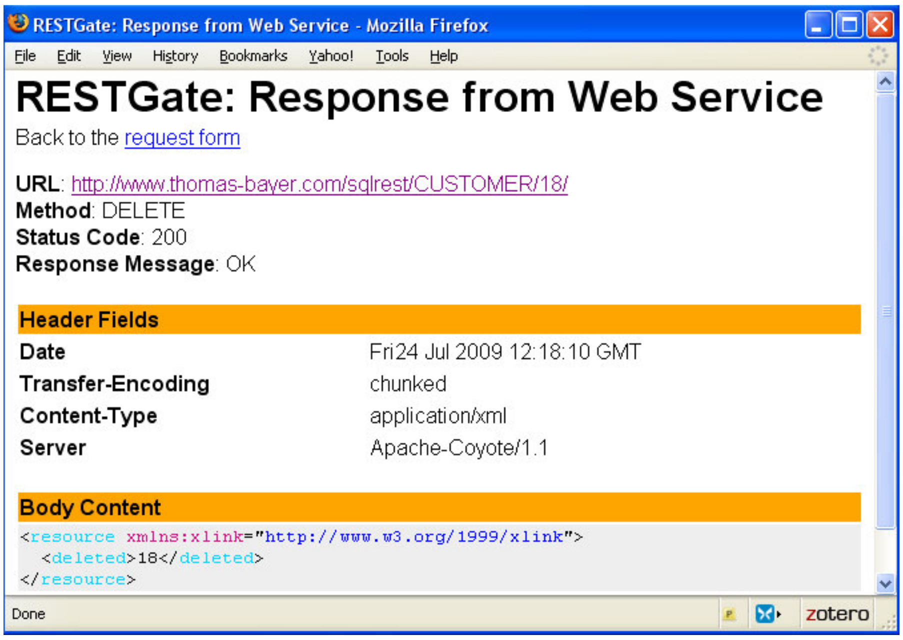
Send HTTP GET, POST, PUT and DELETE requests to REST resources and browse through the response message.

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/18/

HTTP Method: DELETE

Figure 4.9 *Removing a resource with HTTP Delete*

DELETE Response: Figure 4.10 shows the response to the request according to Figure 4.9.


RESTGate: Response from Web Service - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
Done
zotero

RESTGate: Response from Web Service

Back to the request form

URL: http://www.thomas-bayer.com/sqlrest/CUSTOMER/18/
Method: DELETE
Status Code: 200
Response Message: OK

Header Fields	
Date	Fri24 Jul 2009 12:18:10 GMT
Transfer-Encoding	chunked
Content-Type	application/xml
Server	Apache-Coyote/1.1

Body Content

```
<resource xmlns:xlink="http://www.w3.org/1999/xlink">
  <deleted>18</deleted>
</resource>
```

Figure 4.10 *Response after removing a resource*

4.6 Advantages

RESTful has some aspects which can be viewed as positive, including the following [Pau08]:

- RESTful Web services appear to be simple because REST applies many existing well-known standards (HTTP, XML, URI, and MIME) and need only infrastructure that has already become ordinary.
- HTTP clients and servers are compatible with all programming languages and operating system/hardware platforms, and the default HTTP port 80 is usually left open by default in most firewall configurations.
- Only a small effort is needed to build a client of a Restful service. Services can be tested using simply a mere web browser and the development of client software becomes superfluous.
- REST allows discovering Web resources without any discovery or registry repository.

4.7 Disadvantages

RESTful has also some aspects which can be viewed as negative, including the following [Pau08]:

- Encoding a large amount of input data in the resource URI is impossible because the server either refuses such requests or crashes
- It may also be challenging to encode complex data structures into URI as there is no commonly accepted marshalling mechanism. Inherently, the POST method does not suffer from such limitations.

- Unlike SOAP-based web services, which have a standard vocabulary to describe the web service interface through WSDL, Restful web services currently have no such grammar. Both the service consumer and service producer must have an out-of-band agreement. Services can be described using Web Application Description Language (WADL). It is an XML-based file format that provides a machine-readable description of REST web services. WADL is not yet widely supported.

- While SOAP-based web services support a standard vocabulary to define the web service interface by using WSDL, Restful web services at present do not define such grammar. An agreement has to be established between the service consumer and service producer.

4.8 RESTful Web Services Vs WS-*

[Pau08] proposes a good comparison of RESTful and WS-* which complies with the architectural, conceptual and technology principles comparison.

- **Protocol Layering**
  RESTful uses the Web to publish accessible information whereas WS-* uses the web as transport medium for exchanging messages between web services.

- **Dealing with Heterogeneity**
  Both RESTful and WS-* allow to build applications over heterogeneous systems using two different mechanisms. RESTful is based on HTTP protocol and WS-* based on SOAP while both of them are platform independent.

- **Defining Loose Coupling**
  RESTful and WS-* have loose coupling characteristics. Modifications can be made to a Web service without affecting its clients.

- **Contract Design**
  With RESTful web services, no decisions must be taken to define the available operations (contract less) whereas with WS-*, there are Contract-First (begin the development of the web services from the specification of its interface) and Contract-last which implies the bottom-up approach (existing service implementation is published with automatically generated contract).

- **Message Exchange Patterns**
  RESTful supports request-response message exchange (synchronous) whereas WS-* supports both request-response (synchronous) and on-way (asynchronous) methods. RESTful allows Resources to be represented by any format (i.e. HTML, GIF and PDF files) while WS-* is XML-based data type.

- **Transport Protocol**
  WS-* is transport-independent. SOAP messages can be exchanged using different transport protocols. From the perspective of RESTful web services, there is no choice but to use HTTP protocol.

- **Service Identification**
  RESTful web services use the URL standard as the naming mechanism to address resources. Recently, WS-addressing which represents addressing information in WS-* has been introduced.

- **Service Description**
  WS-* are based on the interface description language (WSDL) to describe their services. For RESTful web services, developers must define their resources using URLs. Web application description language (WADL) has recently been proposed for RESTful web services.

- **Services Composition**
  Several languages and tools have been developed to enable WS-* composition (for example BPEL4WS). RESTful Web services composition can be achieved by using Mashups.

- **Reliability, Security, Transactions**
  The WS-* stack consists of several optional specifications related to the Quality of Service of messages exchanged. The basic guarantees of protocols such as HTTP (best effort) and HTTPS (point-to-point SSL security) are used by both RESTful and WS-* Web services.

- **Service Discovery**
  WS-* has Universal Description, Discovery and integration (UDDI) registries which are used to register and discover services. RESTful has no such option. In order to perform those tasks, there is no alternative to do-it-yourself for RESTful web services.

- **Implementation Technology**
  Both WS-* and RESTful Web services can be implemented in any programming language and the client-side library for consuming web services is available to both.

5 Conclusion

In this paper we have described the nature and characteristics of web services and have concentrated on their advantages. We have seen that web services constitute a distinct group of automated services which communicate via Internet and rely on open Internet-based standards. Web services are invoked using a network and perform tasks, solve problems or conduct transactions for an application or user. Web services allow composing pre-existing applications by invoking services via a network which helps to use them more efficiently so as to reduce the need to create new applications. A fundamental characteristic of web services is loose coupling. This implies that the service requestor ignores any implementation or technical detail of the service provider (e.g. programming language, platform). Services are invoked by using messages instead of APIs or file formats. This works due to the independence of the service interface part (WSDL) from the implementation part. We have seen that web services hold great potential for improving efficiency and broadening applications portfolios. Nevertheless, web services are at present subject to limitations which include low performance, weak transaction management facilities as well as insufficient support for business semantics. In addition, we note a lack of homogeneity and coordination regarding the wide range of existing and emerging standards.

RESTful web services have appeared as a lightweight alternative approach to design web services. A central advantage of a RESTful API is its flexibility. Various applications can be provided with system’s resources through data formatted in a standard manner. This technology allows to abiding by integration requirements which are crucial for the conception of systems enabling easy combination of data (Mashups). However, RESTful also present several limitations. No common standard exists for the formal REST service description and RESTful is not applicable for all web services functions, like Transactions, Security, Addressing, Trust and Coordination.

Many open questions remain in the field of web services and RESTful web services. Firstly, there is the unanswered question of defining suitable abstractions for representing web services and their behaviours. Another important challenge is to find modeling techniques and tools which enable the semi-automatic composition and analysis of web services taking into account there semantic and behaviours properties. Moreover, an appropriate way to compose different web services has still to be developed.

In Chapter 6 and 7, two case studies are introduced in order to illustrate various concepts around web services and RESTful web services respectively. The scenario of these case studies shows how to build a web service by using JAX-WS and JAX-RS APIs.

6 Web Services Case Study

This example demonstrates the basics of using Java technology to develop a JAX-WS [*Sun091*] web service (Java API for creating web services. It is part of the Java EE platform from Sun Microsystems). After we create the web service, we develop a client that uses the web service over a network, which is called "consuming" a web service. The client could be either a Java class in a Java Standalone application, or a Servlet, or a JSP (Java Server Page) in a web application. For our example we will use a standalone client type. The code resources for this example are available for download from [*Adh09*].

To realize this example, we need the software and resources enumerated in Table 6.1.

Software or Resource	**Version Required**
Java Development Kit (JDK) [*Sun092*]	Version 5
Web or application server [*Net09*]	Glassfish application (it is bundled with the NetBeans IDE)
Ant [*Apa09*]	Version 1.7.0
JAX-WS APIS [*Sun091*]	

Table 6.1 ***Software Resources***

Preparing the environment

- Install Ant.
- Install Java Development Kit 1.5.
- Install Glassfish Server.
- Set the path to java bin directory.
- Add glassfish/bin to the CLASSPATH.

Creating a web service

Building a JAX-WS style web service using the Java EE 5 platform involves the following steps:

1. Create the application package
   To get started, we create a directory of our choice. For this example, we created a `wsapp` directory and three subdirectories of `wsapp` called `build`, `lib` and `src`. The `lib` directory contains the JAX-WS APIs package. The `src` directory contains the `com` directory which contains the java source files, and the `build`

directory contains the compiled files .class, as well as other files that will be automatically generated.

2. Design and code the web service endpoint
   The web service which we will develop provides a summation service for adding two numbers. In the next steps we demonstrate how to develop, deploy, and use web services. The code of the web service implementation (see Listing 6.1) shows the service endpoint interface which is a regular Java class.
   Note: JAX-RPC 2.0 no longer requires that a Web service bean implements an interface. Thus, we do not need to declare a service interface, but we can start coding a Web service directly as a simple Java class.
   The CalculatorSum class declares one method, add, which takes two integer values and returns an integer value representing the sum of the two integer parameters.

```java
package com;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
@WebService()
public class CalculateSum {
  @WebMethod(operationName = "add")
  public Integer add(@WebParam(name = "param1")
                     int param1, @WebParam(name = "param2")int param2) {
      return param1+param2;}
}
```

Listing 6.1 ***Web service implementation***

3. The endpoint as a web application
   The web service endpoint is deployed as a web application. web.xml and sun-web.xml deployment descriptors (describe how a web application should be deployed) shown in Listing 6.2 and Listing 6.3 must be created.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
  <context-root>/sumwsexample</context-root>
  <class-loader delegate="true"/><jsp-config>
    <property name="keepgenerated" value="true">
      <description>Keep a copy of the generated servlet class' java code.</description>
    </property>
  </jsp-config>
</sun-web-app>
```

Listing 6.2 ***sun-web.xml descriptor file***

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list><welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Listing 6.3 ***web.xml descriptor file***

4. Compile
   Now, we compile the `.java` files by using `build.xml` (download from [*Adh09*]). The java class generated by the compiler will be written to the `build` directory created above. We used the ant task to compile:

   ```
   Prompt> ant build
   ```

5. Packaging the Service with ant
   The web service is implemented as a Servlet. Because a Servlet is a web component, we need to build the application in a `war` archive and then deploy it on the glassfish server. During this process the ant creates a `war` archive and add the deployment descriptors (`web.xml, sun-web.xml`) and java classes to the `war` archive:

   ```
   Prompt> ant buildWar
   ```

6. Deploy the application
   To deploy the web application at first we have to define the `glassfish.deploy` property in `build.xml`. `glassfish.deploy` property points to the `auto-deploy` directory of the domain of glassfish server. We copy the `war` archive to the auto-deploy directory of the server using the following command:

   ```
   Prompt> ant deploy
   ```

   The Glassfish server can be started using the following command line:

   ```
   Prompt> asadmin start-domain domain1
   ```

Consuming the Web Service

The web service client programming model for Java EE is about accessing a remote web service from a Java EE component. Remember that with web services the client and the server are fundamentally disconnected. There are server side issues and client side issues. In other words, if you setup a web service endpoint any client that adheres to the abstract contract of its WSDL can talk to that endpoint.

- Create the Service Endpoint Interface
  We create a client that accesses the Calculator service that we have just deployed. A client invokes a web service in the same way it invokes a method locally. To get started with client application, we create a directory under `wsapp` directory. The same `lib` directory on which contains the JAX-WS APIs library will be used. We use the `wsimport` command provided by the glassfish server library to generate the required artifacts from the web service WSDL file:

  ```
  Prompt> wsimport -d dest
  http://localhost:8080/wsexample/CalculateSumService?wsdl
  ```

  This command will generate a group of java classes under the `dest` directory which will be used by the client to access web service methods (see Listing 6.4).

```
Com/CalculateSum.class
Com/CalculateSumService.class
Com/ObjectFactory.class
Com/Operation.class
Com/OperationResponse.class
Com/package-info.class
```

Listing 6.4 ***Generated java classes***

We create a `JAR` archive which contains all the java classes that we have generated in the previous step:

```
Prompt> ant buildclientlib
```

- Create the standalone client
  Let us develop a client main class that calls the `add` method of `CalculateSumService` (see Listing 6.5). It makes the call through a local object that acts as a client proxy to the remote service. It is called a static stub because the stub is generated before runtime by the `wsimport` tool.

```
/** * @author Albreshne Adhem*/
public class Main {
    public static void main(String[] args) {
          try { // Call Web Service Operation
            com.CalculateSumService service = new com.CalculateSumService();
            com.CalculateSum port = service.getCalculateSumPort();
            int param1 = 400;
            int param2 = 200;
            java.lang.Integer result = port.add(param1, param2);
            System.out.println("Result = "+result);
        } catch (Exception ex) {
            // TODO handle custom exceptions here
        }
}}
```

Listing 6.5 ***Client implementation***

- Compile the client Main java class:

```
Prompt> ant buildclient
```

- Run the client application:

```
Prompt> ant run
```

The result should be like this:

```
[java] Result=600
```

7 RESTful Web Services Case Study

This example demonstrates the basics of using Java technology to develop a Restful web service using JAX-RS (Java API for RESTful Web Services) [*JAX09*], This JAX-RS specification defines a set of Java APIs for the development of Web services built according to the Representational State Transfer (REST) architectural style. After we create the RESTful web service, we consume the RESTful web service using a simple browser. The code resources for this example are available for download from [*Adh09*].

To realize this example, we need the software and resources enumerated in Table 7.1.

Software or Resource	Version Required
Java Development Kit (JDK) [*Sun092*]	Version 5 or later
Web or application server [*Net09*]	Glassfish application (it is included with NetBeans IDE)
Ant [*Apa09*]	Version 1.7.0
JAX-RS APIs [*JAX09*]	

Table 7.1 ***Software Resources***

Preparing the environment

- Install Ant.
- Install Java development Kit 1.5.
- Set the path to java bin directory.

Creating a RESTful web service

Building a JAX-RS style web service using the JAVA EE 5 platform involves the following steps:

1. Create the application package
   To get started, we create a directory of our choice. For this example, we created an `rsapp` directory and three subdirectories of `rsapp` called `build`, `lib` and `src`. The `src` directory contains the `helloWorld` directory which contains the `HelloWorldResource` java class. The `lib` directory contains the JAX-RS APIs. The `build` directory contains the java compiled classes, as well as other descriptor files.

2. Design and code the RESTful web service endpoint
   The RESTful web service, we will develop, provides a "`Hello World`" service. In the next steps we demonstrate how to develop, deploy, and use a RESTful web service. The code in Listing 7.1 shows the service endpoint interface,

which is a regular Java class. The “`HelloWorld`” class declares the `Get` method which returns “`Hello World !`”.

```java
package helloworld;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.POST;
import javax.ws.rs.DELETE;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
/**
* REST Web Service
*/
@Path("/helloWorld")
public class HelloWorldResource {
    @Context
    private UriInfo context;
   /** Creates a new instance of HelloWorldResource */
    public HelloWorldResource() {
    }

    /**
     * Retrieves representation of an instance of helloworld.HelloWorldResource
     * @return an instance of java.lang.String
     */
    @GET
    @Produces("text/html")
    public String getXml() {
        return "<html><body><h1>Hello World !</body></h1></html>";
    }
}
```

Listing 7.1 ***Web service implementation***

3. The endpoint as a web application
   The web service endpoint is deployed as a web application. We create the `War` archive which is constructed as following:

```
-class
    -helloWorld
       -HelloWorldResource.class
-sun-web.xml
-web.xml
-index.jsp
```

Listing 7.2 ***War archive components***

`web.xml` and `sun-web.xml` deployment descriptors shown in Listing 7.3 and Listing 7.4 must be created.

```xml
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
<context-root>/HelloWorld2</context-root>
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>
          Keep a copy of the generated servlet class' java code.
      </description>
    </property>
  </jsp-config>
</sun-web-app>
```

Listing 7.3 ***sun-web.xml descriptor file***

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
 <servlet><servlet-name>ServletAdaptor</servlet-name>
   <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
   <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
   <servlet-name>ServletAdaptor</servlet-name>
   <url-pattern>/resources/*</url-pattern>
 </servlet-mapping>
 <session-config><session-timeout>30</session-timeout></session-config>
 <welcome-file-list><welcome-file>index.jsp</welcome-file></welcome-file-list>
</web-app>
```

Listing 7.4 ***web.xml descriptor file***

4. Compile
   Now, we compile the `HelloWorldResource` java class. The java class generated by the compiler will be written to the `build` directory created above. We used Ant task to compile:

   ```
   Prompt> ant build
   ```

5. Packaging the Service with ant
   The web service is implemented as a Servlet. Because a servlet is a web component, we need to build the application in a `War` archive and then deploy it on the glassfish server (it could indifferently be any other server).

   During this process the ant creates a `War` archive and adds the deployment descriptors (`web.xml`, `sun-web.xml`) and java classes to the `War` archive:

   ```
   Prompt>ant buildWar
   ```

6. Deploy the application
   To deploy the web application at first we have to define the `glassfish.deploy` property in `build.xml`. `glassfish.deploy` property points to the `autodeploy` directory of the domain of glassfish server. We copy the `war` archive to auto-deploy directory of the server using the following command:

   ```
   Prompt> ant deploy
   ```

   Glassfish server can be started using the following command line:

   ```
   Prompt>asadmin start-domain domain1
   ```

Consuming the Web Service

Because RESTful web services are using the standard HTTP protocol and methods, they can be easily accessed from browsers as following:

URL Request:
`http://localhost:8080/rsexample/resources/helloWorld`

Response:
```
Hello World !
```

References

[Erl06] **Erl, Thomas.** *Service-Oriented Architecture, Concepts, Technology, and Design.* s.l. : Prentice Hall Indiana, 2006. 0-13-185858-0.

[Mye09] **Myerson, Judith.** *Web Services Architectures.*

[Pap08] **Papazoglou, Michael.** *Web Services: Principles and Technology.* s.l. : Prentic Hall, 2008.

[Sin05] **Singh, Munindar et Huhns, Michael.** *Service-Oriented Computing.* s.l. : Wiley, 2005.

[Ric07] **Richardson, Leonard et Ruby, Sam.** *RESTful Web Services.* s.l. : O'Reilly, 2007.

[Pau08] **Pautasso, Cesare, Zimmermann, Olaf et Leymann, Frank.** *RESTful Web Services vs. "Big" Web Services.* Beijing : IW3C2, 2008.

[Fie00] **Fielding, Roy Thomas.** *Architectural Styles and the Design of Network-based Software Architectures.* s.l. : University of California, 2000.

Referenced Web Resources

[*OMG09*] **OMG.** Corba. http://www.corba.org/ [Accessed 07 20, 2009].

[*Sun09*] **Sun.** Remote Method Invocation Home. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp [Accessed 07 20, 2009].

[*Got09*] **Gottschalk,Karl.** Web Services Architecture Overview. 2000. http://www.ibm.com/developerworks/webservices/library/w-ovr/ [Accessed 07 20, 2009].

[*NWG88*] **Network Working Group, Sun.** RPC: Remote Procedure Call. 1988. http://tools.ietf.org/html/rfc1057 [Accessed 07 23, 2009].

[*Mic09*] **Microsoft.** Distributed Component Object Model (DCOM) Remote Protocol Specification. 2009. http://msdn.microsoft.com/en-us/library/cc201989.aspx [Accessed 07 20, 2009].

[*Cer02*] **Cerami, Ethan.** Top Ten FAQs for Web Services. 2002. http://webservices.xml.com/lpt/a/1130 [Accessed 07 21, 2009].

[*IBM09*] **IBM.** WebSphere Business Integration Adapters. http://publib.boulder.ibm.com/infocenter/wbihelp/v6rxmx/index.jsp?topic=/com.ibm.wbia_adapters.doc/doc/webservices/webservices17.htm [Accessed 05 15, 2009].

[*Wha09*] **Point, Tutorials.** What are Web serivces. http://www.tutorialspoint.com/webservices/what_are_web_services.htm [Accessed 07 22, 2009].

[*Dmr02*] **David, M. Rubin.** Intro to Web Services. 2002. http://www.softstar-inc.com/Methodology/Softstar%20Web%20Services%20Presentation.ppt [Accessed 07 22, 2007].

[*NWG99*] **Group, Network Working.** Hypertext Transfer Protocol -- HTTP/1.1. 06 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html [Accessed 07 18, 2009].

[*Wik092*] **Wikipedia.** Simple Mail Transfer Protocol. http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol [Accessed 07 22, 2009].

[*W3C08*] **W3C.** Extensible Markup Language (XML). 2008. http://www.w3.org/TR/REC-xml/ [Accessed 07 22, 2009].

[*IBM04*] **IBM, Systems, BEA and Microsoft.** Web Services Transactions specifications. 11 2004. http://www.ibm.com/developerworks/library/specification/ws-tx/ [Accessed 06 20, 2009].

[*Wik09*] **Wikipedia.** Web service . http://en.wikipedia.org/wiki/Web_service [Accessed 06 22, 2009].

[*W3C09*] **W3C.** Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl [Accessed 07 10, 2009].

[*Act09*] **active endpoints.** In Depth. http://www.activevos.com/ [Accessed 07 28, 2009].

[*ebX08*] **ebXML.** Enabling A Global Electronic Market. http://www.ebxml.org/ [Accessed 07 17, 2008].

[*Tut09*] **Tutorials Point.** eXML Introduction. http://www.tutorialspoint.com/ebxml/ebxml_introduction.htm [Accessed 08 04, 2009].

[*SUN05*] **SUN.** Effective SOA Deployment Using An SOA Registry Repository. 09 2005. http://www.sun.com/products/soa/registry/soa_registry_wp.pdf [Accessed 07 29, 2009].

[*Elk09*] **Elkstein,M.** Learn REST: A Tutorial. 09. http://learn-rest.blogspot.com/2008/02/what-is-rest.html [Accessed 07 18, 2009].

[*Wik091*] **Wikipedia.** Representational State Transfer. http://en.wikipedia.org/wiki/REST [Accessed 06 25, 2009].

[*Bay07*] **Bayer, Thomas.** sqlrest. http://www.thomas-bayer.com/sqlrest [Accessed 07 23, 2007].

[*Sun091*] **Sun.** JAX-WS Reference Implementation. https://jax-ws.dev.java.net/ [Accessed 07 20, 2009].

[*Adh09*] **Albreshne, Abdaladhem.** Home Page. http://diuf.unifr.ch/people/albreshn/ [Accessed 07 20, 2009].

[*Sun092*] **Sun.** Java JDK 5. http://java.sun.com/javase/downloads/index_jdk5.jsp [Accessed 07 20, 2009].

[*Net09*] **NetBeans.** Home Page. http://www.netbeans.org/ [Accessed 07 15, 2009].

[*Apa09*] **Apache.** The Apache Ant Project. http://ant.apache.org/ [Accessed 07 22, 2009].

[*JAX09*] **JCP.** JAX-RS: The Java API for RESTful Web Services. http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html [Accessed 06 16, 2009].