Web services payment systems

Master Thesis
Technical University of Denmark

Submitted by Mike Andreasen
31.12.2003

Contents

Preface

This rapport is made as M. SC. project in computer science at the Technical University of Denmark, Informatics and Mathematical Modeling department. The rapport describes the work done in developing a payment platform to use with Java web services. The rapport describes some of the technologies within the web service domain, but the main effort is used in the design and implementation of the system developed.

The project is made in cooperation with IBM Denmark crypto department, and the source material is IBM property.

Special thanks to my supervisor at DTU Dr. Christian Damsgaard Jensen, and IBM Denmark Crypto Team.

31.12.2003

Mike Andreasen

S961171

Introduction

During the last decade of years the internet has evolved from being a media to share relative static data among scientific institutions, to a media used to transport all sorts of data among many different applications and devices. Many things are able to communicate via the Internet, but there is still a huge challenge in integrating all these things via one common language. During the last years XML has proven to be a serious answer to such language. Build on XML, Web Services defines a protocol that enables programs to share functionality via a network. The potential of programs sharing functionality independent of device, operating system and implementation language is huge: businesses can communicate independent of vendor of back end systems, which is reducing cost by eliminate the human factor.

In longer terms Web services makes it possible for the user to be independent of one system to do a specific task, because the common language makes it possible to select another system that can do the same task. This is the first step in direction of a computer grid that always can find computer power to serve the user. Autonomic computing where computers maintain themselves is also one step closer, if the systems know how to communicate with each other. Web services offers a way to register available services, searching in such registries is also done in a standardized language, which enables computer systems to automatic find and use applications on other systems.

One huge challenge in implementing web services, is that the majority of computer systems should be web services enabled, before the full potential of web services is obvious: Speaking an internationally language not enough if everybody else do not understand it. It is therefore important to have as many existing system converting as possible, and make it easy to implement web services in new systems.

An issue when communicating between businesses (and when communicating on the Internet in general), is security. If web services cannot guarantee security, it will be very difficult to persuade companies to convert their existing systems, and choose web services in new applications.

Another issue when implementing web services is the cost of the implementation. Typically companies have good programmers understanding their existing systems, if they must be re-educated to understand web services, it can be relative costly.

Introducing an abstraction layer between the application and the web service technology, makes it possible to have the application programmers concerning about the applications which they have proven to be good at. The details of web service technology can be maintained by other specialized programmers, possible outside the company.

This rapport will investigate the possibility of introducing an abstract layer that can separate the application development process from the security and transport related tasks. The focus will be on how to obtain different levels of security and access control in payment systems. From the start the project is limited to focus on Java as implementing language.

The result of the investigations will be used to develop a system that allows the user to easily deploy applications as web services using different security models.

State of the art

The purpose of the section is to describe the foundation on which a web service payment platform can be build.

The history of distributed computing will be shortly described, and the need for web services will be explained. XML, which is the foundation for web services will be explained, and the different tools that can help developing web services in Java will be discussed. Standards that are used in the domain of web services and computer security in general, will also be briefly covered.

Distributed computing evolution

The need for connecting computers in network evolved because the process of carrying data between systems was a time consuming process. The reason for sharing data among different systems was mainly to get calculations done on specialized systems. Later, the networks were also used to transport data for central storage. The ratio between the cost of network bandwidth, storage and computing power, have had an impact on the way computer systems have been designed since the first computers got connected in networks.

In the early days when computing power was expensive it was a very good idea to have a central server to run CPU intensive jobs on. This server was in general very expensive, and to get as much for the money as possible, the server should constrainedly run with maximum workload. The role of distributed computing was to transfer CPU intensive jobs from the client to one specific server. The technologies used to

In the 1990s advanced technologies for distributed computing were developed by different companies; Object Management Group (OMG) developed CORBA, Microsoft DCOM, IBM DSOM and SUN developed RMI/IIOP. Each of the technologies allows programs written in various languages, to run on remote computers and act as if they were on the same computer. The problem with the technologies is that they cannot communicate with other technologies, which is a problem if e.g. two companies wishes to communicate, but have not bought the same technology.

By the time the distributed technologies were developed, the need for interoperability was very little, because in general companies were not connected to each other via network. The Internet, which formed a connection, was not mature to handle reliable business-to-business communication.

During the 1990s, computing power and storage also became less expensive, and more processing power and storage were added to the workstations. This mend that many of the jobs that required to run on a large server, could run on the client, but also that the client had a lot of spare CPU power when not running CPU intensive tasks. This made the model of one server with many clients less common. Further many computers got connected to the Internet via reliable high-speed connections, so the possibility of selecting between more servers emerged.

In the new marked that focused more on the users needs, than on optimizing the use of centralized servers, the lack of interoperability in the existing distributed computing technologies became an issue.

Web services are expected to be the one common standard that all distributed systems will follow, to overcome the interoperability problem. Making a standard instead of trying to develop a new distributed technology that is better than previous technologies, and therefore should be adapted by all, is an approach that have been tried in other areas of computer science, without great success. An example is OSF/1, which many thought would be the standard for Unix systems in the late 80'ties. The difference in web services is that the standard is accepted by most or the major players in the distributed computing area. The standard will therefore not have to compete with other technologies. The challenge in web services is more to define a standard that everyone can agree on, so the standards take relative long time develop. The long development time for standards, can result in the customers of web services make systems that do not follow the specifications completely.

Introduction to XML

When the Internet became popular among common people, the most used form of communication was by sending HTML documents from a server to a web browser on the users computer. The purpose of the browser is to render the HTML document and thereby make a nice presentation of the information in the HTML document to display on the screen. The problem with HTML is that it is very easy for humans to understand when rendered in a web browser, but it is almost impossible for a computer to separate rendering information from real information in the document.

This is why XML (Extensible markup language) was developed. XML is a meta language which means it is a language to describe other languages, the other languages could be XML security standards which is discussed later in this section. In short XML defines the syntax to use when making new markup languages, this syntax consist of documents elements and attributes.

SGML, which is the parent language of HTML, has the same capabilities as XML and more. The reason for developing XML when SGML existed, was that SGML has so many optional features that do not apply to web publishing activities. XML is therefore a sub set of SGML.

A XML document consist of one root element that can have attributes, a value, and sub elements. Because XML documents contain one root element with sub elements that form a tree structure, it is easy to represent semi-structured data. All data are represented in text, which makes it easy to share documents between different types of systems.

XML has proven its usability in many applications during the last years, not only as transport mechanism between network-connected systems, but also for storing data in stand alone systems. Due to its popularity, many different tools exist to work with XML documents. Java, being a platform independent language, has some focus on XML, the following pages describes some of the important tools available in Java for processing XML data.

For complete reference of XML, refer to the W3C XML 1.0 Recommendation.

Java XML processing tools

The standard Java 1.4 platform contains different packages that can be used to process XML data. Earlier versions of Java did not have as many tools available, it is therefore important to know which platform is available where the application should run. This section describes the tools available not only in the latest standard distribution, but also in other free Java distributions.

Simple API for XML parsing (SAX)

Simple API for XML parsing (SAX) is an API for an event based parser. The force of an event based parser is that it is possible to use the parser on stream data, which is powerful when working with large amounts of data, or when receiving data from a slow media. The Parser API works by calling a specific method on an interface when certain events occur. One event could be when the parser reads "/>", in this case the parser will call the method "EndElement()". The API can be used directly on XML data with success, if searching for something relative simple e.g. listing all elements in a document. However when the requirements to the search becomes more complex, it is usual desirable to be able to view the document as a hole, in a tree-structure. This is where the Document Object Model API can be used.

The Document Object Model API (DOM API)

The Document Object model (DOM) is defined by W3C and is not a Java specific component. The model defines the representation of a parsed XLM document as a tree structure. Compared to the SAX parser, the model has some direct advantages:

- Possibility to modify the document by inserting and removing components.
- Random access to data, because the entire document is present in memory.

The disadvantage is that simple queries will usually take longer time, because the entire document must be read before queries can be made. Reading the entire document into memory will of cause also require more memory to be allocated.

By nature XML documents contains semi-structured data, which means that both the structure of the document and the assigned values are important when reading a document. However sometimes it is desirable that a document conforms to a certain structure. One example of this could be a XML document that represent one table row in a relation based database, before trying to insert the row it is useful to know if the document matches the structure of the database, Document Type Definition (DTD) can be used to this. DTD's are documents that describe the structure of XML documents. The Java DOM API gives the possibility to easily validate a document against DTD's. DTD's can only be used to validate the structure of documents, not the content of the document, to validate both the content and structure XML Schema Definition (XSD) can be used. XSD enables the user to validate data using regular expressions written in XML files, but is currently not available in the standard Java platform.

Extensible Stylesheet Language (XSLT)

Like DOM the XSLT API is defined by the W3C, it describes a language for transforming XML documents into other documents or various other formats. One simple (but very useful) transformation is transforming the DOM tree directly to a data stream, which e.g. enables the tree to be written to a file. Other transformations usually require a style sheet that defines the look of the new XML document.

The API can also be used to transform XML documents into other text based forms. One example of this is Extensible HyperText Markup Language (XHTML), which can contain markup tags that can make a useful presentation in an internet browser. In the transformation the API requires information on which markup tags to use.

XPath

XPath is a data model of a XML document like DOM is, the difference between DOM and XPath is that XPath is a conceptual model, which makes it useful when using absolute references in a document. DOM cannot be used for this because it is an actual API, and the parsing of a document may be vendor dependent. XPath can be used to selecting sub sets of an XML document from a query string, and can therefore be used to search for specific elements or attributes in a document.

In Java Standard Edition 1.4 tool for parsing XPart are included. If running an earlier version of Java, some tools are included in IBM XML security API.

Java API for XML processing

Java API for XML processing (JAXP) is a package that contains implementations of the most common used XML tools. The package contains implementations of Simple API for XML Parsing (SAX) and Document Object Model (DOM), which enables the user to handle the XML data as a stream of events, or have the data represented in a tree structure. The API also gives the user the possibility to plug in other parsers implementations that is compliant to the SAX or DOM specification. The package also contains an implementation of the SXLT API, but is also pluggable so other implementations of SXLT can be used.

Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) is a package that allows applications to marshal Java objects into XML files, and un-marshal XML documents into Java objects. Marshalling is the process known as serialize in Java, a marshaller that marshal for Java objects to XML, is therefore capable of serialize the java object and write the serialized object as text in an XML file.

The package can also validate an object against a XML schema file, to see if the object is an instance of the class defined in the schema. In this way the package binds an object to an XML file, and XML schemas to classes. JAXB is not a part for the Java 2 Standard Edition 1.4, but must be downloaded from SUN and installed separately if used.

Castor is another data-binding framework that provides a mapping from XML schema to Java objects, like it is possible with JAXB. It is maintained by the open source organization Exolab. Castor has been available for longer time than JAXB, and has more features like Java to SQL binding.

The binding between Java Objects and XML documents through XML schemas is very important, because the use schemas as type description enables the use of complex self defined objects in web services.

Java 2 Enterprise Edition (J2EE)

Java 2 Enterprise Edition (J2EE) is like Java 2 Standard Edition (J2SE) a stand alone package. As J2SE focuses on standard application development, J2EE focuses on multi tier network application. Being a multi tier network application means that the application is divided into different layers capable of working on different systems, connected via a network. It is possible to implement such applications using tools available in the standard edition, however J2EE is supplied with tools that enable the user to obtain a higher abstraction layer when dealing with network connections and data structures frequently used in web based systems.

The latest release of J2EE is currently 1.3, it has only limited functionalities that are interesting when developing web services. The main focus in this release is on Java Servlets and Java Server Pages (JSP). Especially JSP is a very powerful technology, if the client is a browser, which main purpose is to render HTML. Servlets are Java programs that are able to run in an application server environment, where the application server will handle the low-level socket connection, and pass the connection to the servlet to handle the actual data processing. Servlets and JSP are closely related, as most application servers will compile the JSP to a servlet and handle all communication in the same way.

Because the application server simply passes the connection to servlet when established, it is possibly to develop a servlet that speaks web service language. What it requires from the servlet is that it has to be able to process the request, which the XML processing tools previous described can do. Further the servlet has to be able to unmarshal the data in the XML message into known data types, process the data, and send back a response formatted as XML to the client.

J2EE version 1.4 is currently in beta release, which means that it is not integrated in application servers yet. 1.4 has more tools available to help marshaling and un-marshaling data, including an API to handle the XML messages as SOAP messages.

Different packages exists to fill in the missing parts in J2EE 1.3, so it is possible to use Java as programming language for web services, further many companies have solutions available, some of these solutions are described in the web service section. Though, it is important to know that most application servers must have plug-ins installed, to support the different web service platforms.

Cryptography

Cryptography is the foundation for most of today's security systems. Cryptography can e.g. be used to keep information secret to unauthorized persons, and authenticate users in a system. Some of the many usages of cryptography will be discussed later in the rapport.

Systems that use cryptography must be carefully designed to use the full potential of the algorithms. The systems that use of a strong encryption algorithm can be insecure, e.g. if the key to decrypt the data can easily be obtained by unauthorized persons.

This section will describe the most common concepts and most used ciphers, but will not go into details on how the algorithms works and can be attacked, as this would go beyond the scope of this project.

Symmetric ciphers

Symmetric ciphers are ciphers that use the same key for enciphering and deciphering. Symmetric ciphers are usually fast, and therefore suitable when handling large amount of data. Examples of symmetric ciphers:

DES:

Digital encryption standard (DES) was developed in 1973 by IBM, it uses s-boxes with key length of 56 bits. DES is no longer considered secure as it can be broken in seconds on a standard PC. If using DES three times with different keys, makes the attack less feasible, because of the longer key. Another more commonly usage of the DES algorithm is triple DES, which uses double length key but enciphers three times. The action performed in triple DES encryption is: C=EK1(DK2(EK1(P))). Because the encryption is done three times, triple DES is relative slow compared to other ciphers.
With today's known attacks on triple DES with double length key, it is considered secure for many years to come.

AES:

Advanced Encryption Standard (AES) is a relative new algorithm build on the Rijndael cipher by Joan Daemen and Vincent Rijmen. Rijndael is a variable key size cipher, and the AES uses the cipher with 128, 192 and 256 bit key length. AES was developed as a replacement for the much older DES algorithm.

Asymmetric ciphers

Asymmetric ciphers use one key for enciphering and another for deciphering. This feature is useful when exchanging keys over an open network, because one key can be distributed to all that whishes to be able to send encrypted data to the owner of the other key. The challenged of using asymmetric ciphers will be discussed later in the rapport.
In general asymmetric ciphers are much slower than symmetric ciphers, and should therefore only be used on small amounts of data.

RSA:

Rivest, Shamir and Adelman are the names of the inventers of the RSA algorithm. The algorithm was published in 1978, and has been the most used asymmetric algorithm since. It operates with key lengths of 512,1024 and 2048 bits, where 1024 bits is considered save today. In a few years 2048 bits will probably be the standard key length for RSA, because faster computers will make attacks on systems using 1024 bit key length possible. The strength of RSA builds on the difficulty of factoring large integers.

Other asymmetric ciphers:

The second most discussed type of asymmetric cipher is based on elliptic curves, the strength of the cipher is that it offers the same level of security with a smaller key size than RSA. Analysis of the algorithm shows that an elliptic curve crypto system using a 234 bit key, will take much longer time to break than a 2048 bit RSA system. Elliptic curves implementations are becoming available in some of the most popular crypto API's, so we might see applications using elliptic curves in the near future.

Hashing algorithms

Hashing algorithms can calculate a fingerprint of data, the size of the fingerprint varies with the different algorithms, but is usually significant smaller that the data it is calculated from. The strength of a hashing algorithm is measured in how easy it is to find source data that result in collisions. A collision is when the calculated fingerprint for two different data sources, is the same. Collisions will always exists, because the fingerprint has a fixed length, and the input data can vary in length.

Secure Hash Algorithm (SHA):

SHA was developed by National Institute of Standards and Technology (NIST) in 1993, in 1995 a revised version was published under the name SHA-1. The SHA-1 algorithm is building on the MD4 algorithm and produces a 160 bit hash value from the input data. Other SHA algorithms exists that produces 256 and 512 bit hash values, but SHA-1 is still more commonly used. The algorithm has proven to be resistant to known cryptanalytic attacks.

Message Digest 5 (MD5):

MD5 was developed by Ron Rivest at MIT in 1991, as a revised version of MD4. It produces a 160 bit hash value. Various attacks have been attempted on the algorithm, some with success, though it has not been possible to generalize the attacks. The algorithm is designed to run fast on 32 bit processor architecture, and is faster than SHA when running on a standard PC.

Certificates, signatures and key exchange algorithms

The foundation of almost all security based on cryptography, relies on the algorithms described above. This section describes some of the most used terms and combinations of the algorithms.

Padding:

Most ciphers and hashing algorithms work on blocks of data with a fixed length. If the length of the input data modulus the block size of the cipher is not an integer, extra bytes must be added until the input data fits the cipher's working length. The way the extra bytes are added, is specified in special standards, so the operation can be repeated with the same result. PKCS is a set of standards used in the domain of cryptography.

Signatures.

Asymmetric ciphers can be used to sign data; the signing is done by encrypting the data to sign with the private key, and to verify the signature decrypting the signature with the public key. If the decrypted data is the same as the signed data, then the signature is verified. Because the asymmetric ciphers work relative slow, it is normal to calculate a hash value of the data to sign, and encrypt this value instead. As with padding, standards exist to define which algorithms and padding scheme to use when generating a signature.

Certificates

A certificate is a proof of possession of a private key. A certificate is generated by an issuer, and contains identity information and the public key of the owner. The information is finally signed by the issuer, to show that the issuer will guarantee the correctness of the certificate. Certificates can be used to establish trust between parts that trusts the certificate issuer. An example of a commonly used certificate standard is X.509.

Key agreement:

As previously discussed asymmetric ciphers are slow, and therefore seldom used to encrypt a connection between communicating parts. Asymmetric ciphers can be used to get the right symmetric key in place on the communicating parts, in different ways. If the parts have exchanged their public keys securely, the key can be generated by one of the parts, and be send to the other encrypted with the other parts public key. If the parts have not exchanged their certificates, they can use a key exchange algorithm like Diffie Hellman.

The first part generated a key pair and sends the public part to the other part. In the generation of the other parts key pair, the public part of the first part is used.. The public key of the other part is send to the first part. Because the key pair generated on the other part included the first parts public key, both parts can now generate the same symmetric key using the public key of the opposite part and their private key.

The key agreement can only be used to ensure that the same key exists on the communicating parts, the identity of the parts cannot be guaranteed.

Network security

Security is an issue in almost all cases, when publishing material on the internet. A normal way of thinking is that security is good, and therefore an application cannot be secure enough. The price of security means that the requirement of an application must be carefully balanced with what the application offers and requires knowledge of. The price of security in applications is not only the cost of the extra code lines, but also the extra time the end user possible uses to operate the application. In computer science security is usually divided into five requirements:

- Confidentiality – Keeping information private, so that an attacker cannot understand the meaning of the messages send.
- Integrity security – The integrity of messages is secured by the system, so that an attacker cannot change the contents of messages.
- Authentication – The system guaranties the identity of communicating parties, so that an attacker cannot claim to have another identity.
- Authorization – The system can limit the access to resources based on the user credentials.
- Non-repudiation – The system guaranties that data cannot be recorded and replayed, so an attacker cannot fake the system to think that the same massage was sent twice.

All requirements above have been known before the internet became what it is today, also a solution to the requirements have been known namely cryptography. Still very few applications have actually implemented all five requirements successfully. The reason for this is that the security can only be obtained if the user of the system is willing to take the time to get registered in a secure way. Further the user must accept to give up anonymity, when communicating with the system.

An example of wrong level of security on the internet is when a user has to register with name and e-mail, to gain access to a resource that is register-free elsewhere. Even if both resources are free, the user will usually choose not to register, to maintain anonymity. In general, if the level of security and the value of the service do not match, the service will probably be unpopular with the users or insecure.

There are different levels in a communicating system, where security can be implemented, all with benefits and drawbacks:

- Network level security
- Transport level security.
- End to end security
- Message level security.

Security implemented in the network layer is usually transparent to the user and application, because it is implemented in the lower level of the communication protocol. Network security can be implemented in hardware, and do therefore not affect the performance of the application using it. Being transparent to the application, the security mechanism does not require special security knowledge of the application programmer. The drawback of the technology is that the connection is not guaranteed to be secure all the way to the receiver, because not all routing points necessarily support network level security. IPsec is security related standards, published by IETF in 1995 that defines how to implement security below the IP layer. Support for IPsec is mandatory for IPv6, but not for IPv4.

Transport level security is security implemented between the IP layer and the application. Compared to network level security, this can guarantee a secure connection between two applications. The application programmer does not need extensive security knowledge, because the transport layer can do all the security related transport behind the back of the application. Secure Socket Layer designed by Netscape, is the most popular transport layer security protocol.

End to end security is when the application programmer takes care of all security used by the application. Like transport level security, this can ensure security between applications, but also between the users of the applications. This can be obtained because the application can ask the user for authentication, and thereby ensure the identity. An example of end to end security is home banking systems that ensure that only the right person has access to a certain account.

Message level security is security implemented in the application layer, but unlike end to end security, the application does not need to be the final destination for the secure content in a message. The same advantages as in end to end security can be obtained in message level security, because the application can decide how to handle the security content. Message level security is especially usable in multi tier systems, where different levels of security can be required in each tier. The possibility of having different security levels in one message also makes the technology interesting in multi user systems, because the same message can be shared in a group of people with different access rights to the message. XML security is a technology that can be used to obtain message level security.

Java Cryptography Extensions (JCE)

The US export regulations are regulations to ensure that American companies do not sell technology that can be used to harm USA in any way, to companies outside USA. Strong encryption algorithms implemented in software were included in the regulations, until a few years ago.

JCE is a pluggable framework that allows the user to plug in own crypto providers. Crypto providers are program packages that implements the cryptographic algorithms. This allows SUN to distribute JCE, but without providers that support strong encryption. Until Java standard edition 1.4 the JCE was not included in the standard java edition, but available for download from SUN’s web site. From Java 1.4 the JCE is included in the standard edition, but still without providers of strong encryption algorithms.

To enable strong encryption with JCE, it is possible to download crypto providers from open source organizations. Bouncycastle is one organization that has free implementations of strong encryption standards.

Java Secure Socket Extension (JSSE)

JSSE is an API that allows Java programs to communicate using Secure Socket Layer (SSL). The package is included in Java standard edition 1.4, but in earlier versions it must be downloaded and installed separately. The package contains crypto functions that can be used to for other purposes that SSL e.g. the possibility of verifying a certificate. Further the package contains a crypto provider with implementations of many components, used when establishing a secure connection.

XML Signature and Encryption

When sending private information over open networks, some protection is required as discussed in network security section. When formatting transport messages as XML documents, there are some advantages in choosing message-level security:

- It is possible to encrypt only secret parts of the message and thereby preserve the structure of the message, which makes it possible to understand the not-secret information.
- It is possible to use different keys for encryption of different parts of the message. This makes it possible to let the same message travel in a multi tier system, and let the different tiers handle only their responsibilities, as part of the message can be kept secret to some tiers.
- The end point of the security can be chosen by the system designer. Because all security information is kept in the message, it is possible to handle the anywhere in the system.

XML Signature and encryption are standards to add security information to XML documents, so Message level security can be obtained.

One problem the standards addresses, is that digest algorithms and ciphers work on binary data of fixed length. An XML document (or a part of it) can be handled as a array of bytes, like all other data, and be padded to a multiple of the fixed length, like data usually are handled when using cryptography. This approach is very useful if the data should not be modified during the transport, but it is not desirable for XML documents because the information can remain the same, if the values of the bytes that form the document changes. An example of this is if comments are added to the document, the document information would be the same because the comments would be discarded by the destination parser, but the byte value of the document would not be the same. The XML signature and encryption standards solve this problem by making a canonicalized object containing only information from the document. This object's byte value will always be the same for the same information, regardless of comments and other non informational bytes added to the document. This feature makes it possible to use standard crypto tools on XML documents to calculate hash values, sign and encrypt/decrypt.

The XML signature standard describes how to sign branches and entire XML documents. It also describes how to attach the signature and information about how it was generated, so the receiver can determinate how to verify the signature. In the same way that it is important for the receiver of a signature, to know which algorithm is used to calculate the signature, it is also important to the receiver of an XML signature to know which canonizer was used.

Example. Consider following XML node:

```
<data>
    <value>test</value>
</data>
```

What seems to be important is that the data element has the value test. But the node consist of an element data whit three children:

- A text node with the value "   \n".
- An element named value
- A text node with the value "\n"

The value element consists of a text node with the value "test". It seems natural that the canonizer can ignore the two new line text nodes, because they do not change the fact that data's value is "test". But if the receiving application knows that the value of data should be found in the second child of data, it would be catastrophic to remove the text nodes. Different canonizer standards exist, so the programmer can decide the best way to canonize the applications XML data. Because of the standards, the canonization process is not bound to a specific implementation of the canonizer, and can therefore be used independent of platform and implementing language.

The XML encryption standard described how to encrypt branches and entire XML documents. The standard suggests that the selected data for encryption will be encrypted and handled as one node until it is decrypted. This also makes the structure of the data secret. It is also described how to attach decryption information to the document, so the receiver known how to handle the document.

The two standards are not bound to specific algorithms and can therefore also be used with future algorithms.

Currently XML security is not implemented in the standard java distributions, but IBM offers a free implementation of the standards called XML security suite.

The two standards do not address how to do key management or how to establish trust.

XML Key Management Specification (XKMS)

XML key Management Specification (XKMS) is a specification that describes the protocol in a system, which will minimize the effort required by clients to obtain keys and verify trust. XKMS was developed by Microsoft, Verisign and webMethods, but is now controlled by W3C.

The protocols XKMS defines, can be used in systems that contain one or more servers, which can handle revocation information, validating chains of certificates and other things to help clients in establishing trust.

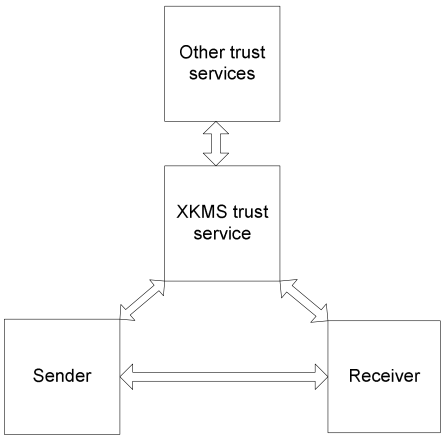


Figure 1

The XKMS trust service is build on two XKMS specifications the *XML Key Information Service Specification* (X-KISS) and the *XML Key Registration Service Specification* (X-KRSS). The X-KISS specification specifies the format of messages in a two tier system consisting of a locate service and validating tier.

- Locate Service.
  Handles key information. If a client whishes to know whether a signature is valid, but do not have the public key, it can send a reference of the certificate containing the public key to the X-KISS tier 1. The locate service will find the actual certificate using the reference; the certificate can be located on other XKMS trust services. The client will have to perform the validation of the signature itself.

- Validate service
  Validates trust and certificates revocation. If a client whishes to validate a signature, but does not have the facilities to do this, it can ask the X-KISS tier2 to perform the operation. The validate service can locate the public key in a certificate, check the certificate for revocation and check the certificate chain path. The certificates in the chain may be located in other XKMS trust services.

X-KRSS specifies the protocol to use in order to register certificates, and handle revocation in a XKMS trust service.

Parts of the XKMS specification can be used even if the application is not a fully XKMS system as described in the specifications. E.g. the syntax of a key reference for the receiver to look up may be useful in other types of systems.

Security Assertion Markup Language (SAML)

Security Assertion Markup Language (SAML) is a standard for transferring authentication, authorization and permissions, developed by OASIS Security Services Technical Committee. Permissions Management Infrastructure solutions (PMI) have been available for some time, but the protocols used in many of these systems are vendor specific, therefore the interoperability of such systems is poor. SAML defines an open protocol based on XML, to use in PMI systems. Single sign on allows users to enter authentication information once to be authenticated across multiple domains. Before SAML this feature was limited to systems that were able to speak the protocol of the proprietary PMI system.

When logging in to a SAML enabled application the permission information is stored in a SAML assertion. The assertion contains information about when, how and for which resources a permission was granted. A SAML token is generated, which is a unique identifier containing authentication and authorization data. When the user whishes access to a SAML application, the token is submitted to a Policy Enforcement Point (PEP), which is responsible for requesting access in the systems Policy Decision Point (PDP). If the user has access to the desired application, the PDP returns an authorization decision assertion, which is attached to the users SAML token, and the user can access the protected resource. If the authorization decision assertion returned by the PDP states that the user cannot use current login information to access the resource, the user is requested to log in to that specific resource.

SAML defines the interface between the user application, PEP, PDP and the resource application. Therefore parts of SAML can be used in most systems that require access control.

Application servers

As described in the J2EE section, application servers are usually used as platform, when deploying JSP sides and servlets. Many features on the available application servers can also be used in a web service environment. Following features are also usable when deploying functionality as web service:

- The application server will handle all network traffic, so the web service developer can concentrate on developing the application and web service interface.
- The application server enables more applications to run on the same network port.

- Most application servers have access control and supports security mechanisms like SSL.

Different vendors of application servers exists, the most popular are BEA Weblogic and IBM Websphere application server. Both servers are relative complex and focuses on the high end marked, where it is very important to be able to handle a large number of simultaneous users. Tomcat application server from Apache group, is a free open source server written in pure Java, the performance is not as good as on the commercial servers, but it is functional and good for testing purpose or smaller businesses. The Tomcat server can be combined with Apache web server, to remove the work load of static sides from the application server.

Web services

One common misunderstanding in the term web services is that when talking about distributed computing, web services do not mean all the services available via the world wide web. Rather it means a web of services, though it is true that most web services communicates via http on the Internet, like it is the case for most web pages.

As previous described distributed computing is the capability of computing data on a remote computer, using the same interface as if the computing was done locally. A normal way of doing distributed computing is using Remote Procedure Call (RPC), where the client calls a method, possible with some parameters, and receives the data the method returns (if any). To send the parameters via a network, the parameters and return value needs to be serialized. The way the objects are serialized must be known to both the server and client, further it must be assumed that both parts know the type of the parameters. The process of serialized and de-serialize is often referred to as marshalling and de-marshalling, and the components responsible for the process, stubs. figure 2 shows the model of a RPC call.

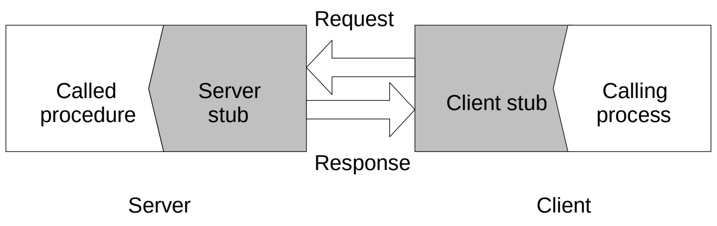


Figure 2

Web services are definitions of the communication protocols used by the server and client stub. The differences between web services and other protocol specifications are that web services is an open, platform and language independent protocol. One of the challenges in web services is therefore to marshal objects from one programming language to a universal understandable format, so the receiver always can de-marshal the message to an equal object in another programming language. Web services solve this challenge by using XML which is understandable in all language, because XML consist only of text. Web services uses some type primitives that is available in all programming languages like Strings and Integers, to use more complex types it is necessary to publish a definition of the type, and refer to this when using the type.

Web services must have an interface description that describes exactly how to communicate with the service. The description must include information about where the service is located, which methods can be called, and which types it expects and returns. This interface description can be published in a service registry, where clients can search for services that fulfill their requirements. Figure 3 shows a web service system

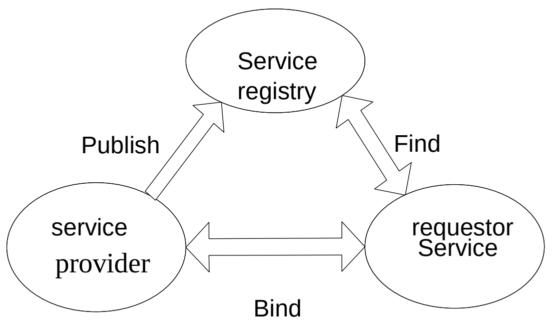


Figure 3

The service provider develops a web service, and publishes its interface description in a service registry.

The service requestor queries the registry and finds the service location.

The service provider and requestor start communicating using the protocol described by the service provider.

The service registry does not contain the actual descriptions of the services, only searchable keywords of the services. If a client whishes an interface description, it will be send from the service provider.

A web service system does not need to use a service registry. If a service should only be used within a small group of clients, the address and interface of the service can simply be given to the involved parts without publishing the information to other parts.

Web service Description Language (WSDL)

WSDL is a language based on XML used to describe the interface of web services. A WSDL document must have a definitions element as document root; the element contains definitions of the namespaces used in the document. Following elements is allowed in the definitions element:

- **Message:** A message represents variables send between the web service and clients. Messages contain one or more parts that specify the parameter name and data type. A message can both be a request and a response.
- **Porttype:** The message element does not specify how the data is associated with an operation, this is done by the porttype element. Porttype contains operation elements that links operations (methods) to messages; an operation element contains a name, an input message and an output message.
- **Binding:** The binding elements contain information about how to send the messages in the porttypes. The most common binding is SOAP RPC over http. In the binding element is an operation element that describes the encoding of the data
- **Service:** The service element specifies the URL address that clients must contact to invoke the web service. One address for each binding is required.

Figure 4 is an example of the structure in a WSDL document

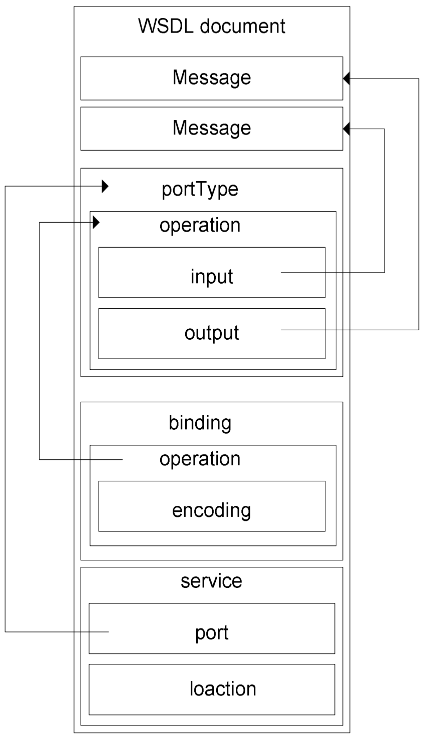


Figure 4

Notice the similarity between a WSDL portType and an interface in object oriented programming; they both describe a set of functions and the data types used with the functions. The operation element is the analogue to a method in programming languages.

The similarities to programming languages, makes it relative easy to develop applications capable of making WSDL document from existing programming code, and make programming code from WSDL documents. This makes it possible to work with web services without detailed knowledge about the web service description language.

Universal Description Discovery and Integration (UDDI)

UDDI is the most popular XML service registry to use with web services. UDDI makes it possible to register web services in a database that can be queried by others to obtain information about available web services. UDDI only registers the information used for describing what services do, not how to communicate with them. If a client finds a service it wishes to use, the binding information is requested from the web service. The binding information usually consists of a WSDL document.

UDDI registries share similarities with the search engines used for web pages, both can search for resources on a network. UDDI will however require the services to register, before it can be found. The communication interface to UDDI registries is based on XML, and can therefore easily be integrated in applications that whish to look up services. This is only the case for standard search engines if they are made accessible via web services.

Simple Object Access Protocol (SOAP)

SOAP is an XML protocol framework for the communication between distributed peer processes. SOAP defined a general structure of messages, which can consist of a header and a body. Both the header and body can contain SOAP blocks, which is used for the application data. SOAP blocks in the header can have an actor attribute that indicates the receiver of the block. The transport path of a SOAP message is not bound to the transport layer, a SOAP message can therefore travel between different SOAP nodes that will process some of the data in the SOAP parts, if they can take the role of the actor described in a SOAP part. If a SOAP node fulfills the role as anonymous, the node is considered the final receiver, and will process the SOAP parts without actor information. SOAP blocks in the body cannot have an actor, and is therefore always for the final receiver.

The use of complex objects can be obtained in two ways: SOAP RPC and document style SOAP. Most Java API for SOAP messaging supports complex objects over SOAP RPC, if the objects are compliant to the Java Bean specification. This can however lead to complications, if other implementation languages is used, because the Java Bean to XML serialization is not well defined. Another way is to use document style SOAP, where XML-schemas is used to describe the complex objects. This describes the structures in an well defined way, independent of language, platform, environment and transform. The drawback of this solution, is that the developer will have to write an XML parser that can understand the schemas used, and turn this in to real objects.

The receiver of a SOAP message is expected to send a message back to the sender, if it encourages problems when processing the message. The message must contain a SOAP fault, which must describe the error occurred. Four standard types of SOAP faults exists:

- Env:Server temporary problem on the server.
- Env:DataEncodingUnknown if the server cannot decode a parameter
- Rpc:ProcedureNotPresent the server cannot find the procedure
- Rpc:BadArguments if the parameters send to the server do not match the parameters of the method to call.

Other faults can be used if required by the application.

Java API for XML-based RPC (JAX-RPC)

JAX-RPC is the package for developing web services in Java, it supports SOAP and WSDL, to make the interoperability requires in web services. The API is based on the Remote procedure call (RPC) model, but has some features that go beyond standard RPC. The possibility of extensible type mapping, makes it possible to use the API for self-defined types. Standard RPC is based on one request and one response; in JAX-RPC it is possible to split the call into more documents, which is practical if large amounts of data should be send.

A web service developed using JAX-RPC consist of two classes: The service implementation class, and an interface class that describes the methods available for remote call. The package includes a mapping tool that can be used to generate the tie class that can be deployed on an application server, it is also possible to generate a WSDL file that describes the service interface. To write the client, the mapping tool can be used to generate a stub class from the WSDL file, or from the interface class written on the server. The tool also generates a skeleton for the client application that uses the stub to remote calls.

JAX-RPC can map the primitives and some collection classes of the standard Java edition to XML/WSDL datatypes. Thus, it is important not to include self defined classes in the interface methods of web service implementations that should use JAX-RPC as web service platform.

SOAP with Attachments API for Java (SAAJ)

SAAJ is a framework to send SOAP messages over a network. The framework can be used in a standalone client that should communicate with a RPC based web service. When generating a client stub in JAX-RPC, SAAJ is used to form the message as SOAP messages. The framework ensures that the messages conform to the SOAP 1.1 specification, by using classes that represent the elements in a SOAP 1.1 message. The framework can both construct SOAP messages and parse data streams to SOAP objects.

The protocol used to communicate the SOAP massages assumes that receiving part will send a response for each message send. This makes the framework suitable of communicating directly with web services that sends a response for each request.

As the name indicates, the framework can include attachments to SOAP messages; the attachments can contain binary data, which is a better way of transporting large files like pictures and sound clips, than encoding the files to fit in an XML document.

Java API for XML Registries (JAXR)

JAXR is an API that enables the use of XML registries in Java. UDDI is the most common registry to use with web services, but other types of registries are available e.g. ebXML. JAXR includes the general concepts of XML registries, but is pluggable so the vendors of registries can write their own provider, and thereby make JAXR capable of communicating with the registry. The concept is the same as with JDBC that makes java applications capable of communicating with databases, if the database is supplied with a JDBC driver.

JARX includes functions to registering business information in a XML registry, so others can find the business information. The information to register is business name and classification scheme, where the classification scheme is used to describe what the business offers, so it is easy to compare the business of same kind. The API also includes functionality to search registries, and get binding information to the business.

Apache AXIS

Apache group have developed a SOAP engine called AXIS. The engine is available in Java and a c++ implementation is being developed. The SOAP engine does much the same as JAX-RPC, but is different in some ways:

- AXIS supports both RPC and document style web services.
- XML Schema support that enables easy use of external serializers and de-serializers
- AXIS wraps the service, and generates SOAP faults if exceptions
- Java classes can be deployed instantly, simply by copy the class to AXIS deployment directory.
- AXIS is capable of generating WSDL documents for deployed web services, on requests from the network.

AXIS includes an application for deployment on Java application servers, and a standalone server that can easily be used for testing purposes. Tools for generating WSDL documents from Java classes, and Java classes from WSDL documents is also included.

Java Web Service Development Platforms

Many large companies involved in web services have developed a web service development platform. The platform usually consists of different APIs to easily develop servlets communicating via SOAP messages.

IBM web service tool kit (IBM WSTK) also includes tools for registering services in a UDDI registry. IBM has a public UDDI registry server that can be used for test or publishing free of charge. The tool kit does not include any servers, but instructions on how to use the toolkit with the most popular applications server, and the IBM public UDDI registry.

SUN’s Java Web Service Development Package (JWSDP) is much like IBM WSTK, but includes a configured Tomcat application server and a registry server, which makes it easy to test web services with UDDI registry. The distribution makes use of all the web service packages from SUN, including J2EE 1.4 (currently in beta release). The platform is interesting, because it includes technologies that might become standard in the Java standard edition or enterprise edition.

The security tools in the package, makes it possible to sign and verify SOAP messages, but is nonstandard, because the web service security standard is only available in draft. The security is implemented as an extension to JAX-RPC and is called JAX-RPC-SEC. Currently only signing and verifying signature is available in the package.

Other companies have similar solutions, some free of charge, others included with their application server/ web service solution.

Web services Interoperability Organization (WS-I)

WS-I is an organization that is formed to accelerate the usage of web services, by defining best practice for development and usage of web services. The organization groups their recommendations into *profiles.* A profile is related to sets of web service specifications in a specific version.

For each profile the organization produces documents on how to read the related specifications, samples on how interoperability can be obtained, and test assertions. The test assertions can be used to test a working web service to see if it conforms to the profile.

The members of the organization are other organizations that work with web services.

Suggestion to a web service security standard

It is specified in the SOAP and WSDL specifications how data types should be exchanged between web services. To maintain the concept of having a standard way of describing how to interface to a service, it must be possible to describe security assertions in the same way as data types. Therefore Microsoft and IBM have worked together developing specifications for exchanging security information in a web service environment. The proposal has been accepted by OASIS that has formed a committee who currently is working on moving the specification to an open standard.

The specifications mainly describe enhancements to the SOAP standard. Where it is possible, the specifications make use of existing standards like XML signature and encryption. To simplify the overview, the specifications are divided into groups:

WS secure conversion	WS Federation	WS authorization	Federation
WS policy	WS Trust	WS privacy	Policy
WS security			Messaging

The model should be read from the button and up; underlying specifications provides the foundation for specifications above.

Following description of the model will only describe the concept of the horizontal layers.

Ws security

Web services security builds on the SOAP specification; it describes how a security token can be attached to a message. The token works as a claim from the sender of the message. An example of a claim could be that the sender attaches his certificate to a message, to claim that he has the private key of the certificate. Security tokens can also be combination of username and password, access tickets (used in kerberos) or something not standardized like an iris scan.

The specification suggests that encrypting and signing of the SOAP message follows the XML security specifications. It is only specified where the signing and encryption information should be stored in the message.

Ws security does not specify which modifications that should be done to the WSDL document that describes how to communicate with the secured service. This is discussed in the web service policy suggestions.

Ws policy

Web service policy describes a grammar for expressing capabilities and requirements in a web service system. The suggestion makes use of policy tokens that can describe a security policy of a system. A policy token could describe the encryption used in a web service. To specify how a web service will use the policy tokens, some assertions are defined; assertions can be used in combination with policies. Examples of defined assertions are “required” and “used”

The policy specification also describes how to attach policies to WSDL documents, so different parts of the WSDL can have different policies. This makes it possible to specify the security requirements of a web service, and make it readable to others.

Ws federation

Web services federation defines mechanisms to build trust between identities that do not necessarily use the same identification method. The specification defines a Security Token Service (STS), which can issue different types of security tokens. The trust between different STSs makes it possible for a user only to establish trust to one STS, to be able to authenticate in the federation of STSs.. The specification describes some of the same functionalities, which is present in SAML and XKMS

Because web services is a relative new area of distributed computing, the experience with the technology is limited, therefore the specifications take long time to define.

The suggestions are written based on the needs the two companies and business partners experiences when implementing web service systems.

Because none of the suggestions above are accepted as standards yet, very few have actually implemented the suggestions. In the web service platform from SUN digital signatures are implemented, but encryption is not. Because of the long development process of the standards, some people recommend companies to implement non-standardized security systems in their web services systems. Simply because they mean the standards will not mature before the end of 2005.

Integrating security tools

Looking at existing development tools, many have already implemented the possibility of deploying applications as web services, directly form the development environment. The challenge of deploying an application as a secure web service does not need to be much harder that without security; small steps in this direction have been made in Microsoft's .net development platform. Also IBM has added the possibility of deploying applications with XML signature and encryption. The configuration of the security applied to the application, and the interaction between the application and the security is very limited though.

In the requirement section, the integration between web service security and the web service application will be discussed in detail, and it will become clear that the application needs some interaction with the security, when developing payment systems.

Payment systems

Today many different payment models exist on the internet, and many factors indicates that there will be even more in the future. Probably the best known payment model on the internet, is selecting goods from different html pages, and navigating to a paying page that allows payment with credit card via an html form. Once the credit card information is submitted the Internet shop will process the order, and withdraw the money from the credit card. Through the last years, the payment model has proven to work, and is used by most companies that sell things that have a certain value.

Other systems exists that may not look like a payment system, but works as one for the company. An example is advertisement on web pages, when the company gets paid when the user sees or clicks on the advertisement. Another example is when a company requires information from the user before giving access to resources; this information can be valuable to the company when promoting their products, because they now know something about their target user group.

Common to all successful resources on the Internet is that they use a payment model and prize that is accepted by the user, which also must be the case for future resources and payment models.

So why look for new payment models when it works fine the way it is now? One answer to this could be that the future resources that are available on the Internet may vary from what we see today.

Resources on the Internet and future payment models

As mentioned before a very common way of buying things on the Internet, is by ordering it and pay with credit card. The only difference between this approach and going to a physically shop, is the fact that the purchase can be done when the shop normally would be closed, and that the geographically placement of the shop is less important.

If, in the future, we will see more companies giving access to their systems via web services, the user will have the possibility to instant change settings that normally would require human interaction. An example of this could be the cable television company, allowing users to change their subscription instantly. This could enable the user to have a button on the television, to turn advisement on and off, by changing the subscription on the cable television provider. Perhaps the television itself could search between cable television companies, to find the cheapest provider for the channel selected.

In general the controlling of most resources based on services, could be made accessible as web services, and possible be maintained by the users systems and not directly by the user, as it is the case in most internet based ordering systems today.

Today trust on the Internet is based on people's trust in large organizations, and the possibility of punishing persons who breaks law on the net. This model is closely related to the rules in the real world, where the government issues a coin that people in the country trusts, if somebody breaks the law, the government will punish the person. The model is not very good across country boarders, because there is no common law and perhaps not all governments can be trusted. Especially when more people in the non western countries gains access to the Internet, this becomes an issue.

The technology to give digital identities based on trust to a large organization have existed for many years, despite the many benefits of everybody having a digital identification that can be used everywhere, it has not really been applied yet. One of the reasons for this is that it would not be possible to be anonymous, if the digital identification should be used everywhere.

A solution to the trust problem is to have smaller trust communities, where the trust not necessarily is based on a digital id. The important factor is that security and anonymity must be balanced whit the value of the service.

In following sections it will be analyzed if web services is a good solution for payment systems, now and in the future.

Requirements to a web service payment system.

In this section requirements to a system that can help developing and deploying web services that require payment, will be defined. The purpose of the requirements is to be able to form a general model of the system; the requirements will therefore not be very specific at this stage. Use cases for the system will be defined later, combined with the requirements listed in this section, this will form the system specification.

The state of the art section shows that many API's are available to help develop secure web services. The use of the API's often requires intensive knowledge within the domain, because most of the API's are very flexible. To implement an application as a web service and apply a suitable payment model, requires knowledge most of these API. Most programmers are specialized within a more narrow area of computer science, and it is therefore typically necessary to combine the work of more specialized programmers when developing large web applications. One way of combining the work of more programmers, is by defining a set of interfaces and integrate the system, once the sub components have been developed. The integration process can be manual or automated; the automated process will usually require an integrating system. An example of an automated integration system is the Microsoft Java development platform that is able to integrate compiled Java classes with a Java runtime environment, which produces an executable file on the Windows platform.

One purpose of this system is to divide the knowledge required to develop a payment system as web service into smaller units, and define the interfaces between them. An automated integration tool capable of integrating the units into a web service, must be developed.

The payment model used by one application, is most likely not special for the application, the integrating system must therefore be able to reuse a payment model in different applications. The system should hide as much of the web service transport layer as possible, as the application and payment model programmers probably known little about developing web services. The system must therefore be able to combine the work of three groups:

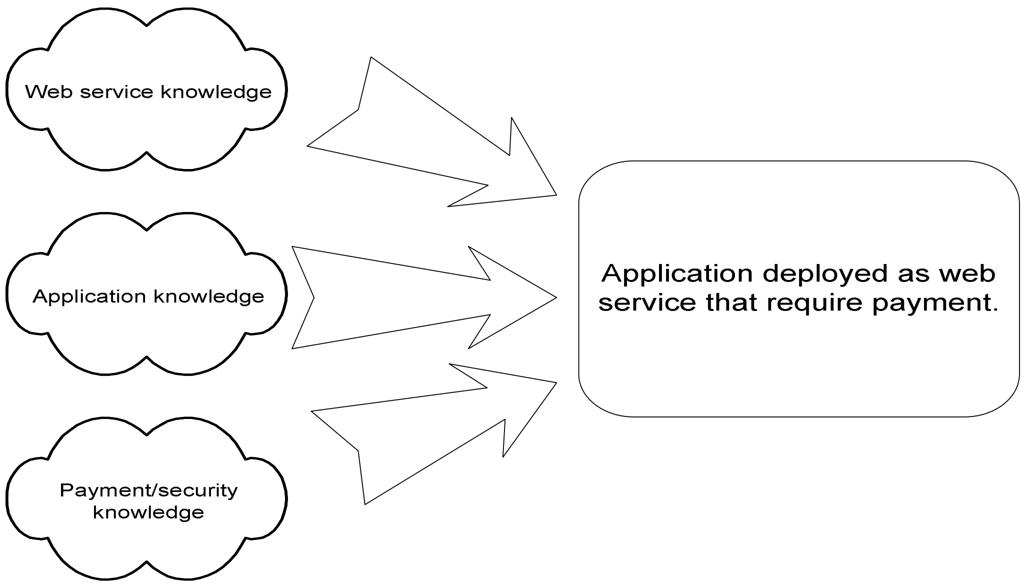

Web service knowledge
Application knowledge
Payment/security knowledge
Application deployed as web service that require payment.

Figure 5

The web service that the system generates, must meet the requirements of web services in general, which means that the system must generate a WSDL file, that can be used by other programmers to interface to the service.

The abstraction between the service application and the business layer must be dynamic. It must be possible for the developer of the payment model to decide which parameters the application has access to. Further it must be possible to establish a channel between the application and payment model, because the application may depend on data processed in the payment model and vice versa. This channel is essential when using message level security, because it gives the application programmer the possibility of getting and setting data in the security layer.

Once the application and payment model have been developed, and the web service environment has been configured, it must be easy to integrate the components into a web service. At this stage the flexibility is less important, and configuration options must be kept to a minimum.

Payment models most likely share features e.g. more payment models may require an encrypted channel, therefore it must be easy to reuse components in different payment models.

As seen in the state of the art section, the payment models may be different in the future. The system must be able to support future payment models, even if they are significant different from to days known models for payment systems.

Once a payment model have been developed, it must be possible to make minor changes to the way applications make uses the models, during the deployment phase. This is because a payment model should be able to fit a broad range of applications.

The system will not include client applications, or helping tools to develop client applications. The system must however be designed so components on the server can be reused when developing a client. The communication between server and client must be done and documented in standard web service way.

Requirements to payment models

Not all of the payment models will actually be implemented, but it is important to state their requirements, to make sure the system will be able to handle them. The following models are not all directly related with payment, the term “payment model” will from here be exchanged with the term “business model”, which better describes the functionality

Empty model

If an application should be deployed as web service without a business model, the empty model can be used in the integration process. The model must contain the same components as other business models, but must not modify data when used. It must be possible to use the model as foundation for other models due to the empty implementation.

Signature model

The need of signing data is present in many of today’s payment system, the signature can be used as proof of an agreement between the customer and supplier. The signature model must be able to verify signatures on incoming requests, and sign outgoing responses.

In the configuration of the model it must be possible to choose the action if the signature cannot be verified. The model must assume that the certificate of the client is present at the server.

Key agreement model

Before communication encrypted with symmetric ciphers can begin, the communicating parties must agree on which key to use. The key agreement model must be able to make the same symmetric key available to the client and server, in a secure manner.

In the configuration of the model it must be possible to choose how the symmetric key is stored on the server.

Encryption model

The encryption model must address the need for confidentiality of data transmitted between the client and server. The model must be able to decrypt incoming messages and encrypt responses before they are sent back. The model must use a symmetric key available on the server.

In the configuration of the model it must be possible to assign a static value for the key name to use, or choose to let the client decide the name.

Secure payment

The secure payment model must make use of the features developed in signing, key agreement and encryption model. The model must be able to establish a secure channel to the client, and use this channel to process the client request. The client request must be signed, and the model must be able to store the signature and the signed data. The model must be able to sign and encrypt the response message. The model must not be vulnerable to replay attacks.

In the configuration of the model it must be possible to choose to pass the name of the client to the application, if the application has a suitable interface. It must be possible to choose a public key to verify the certificate send by the client

Capability access

The model is must be aware of an applications entire usage, and based on how the clients interfaces to the model, give access to more or less of this usage. The model must be able to supply the application with default data, if the interface used is not exactly the same as the application’s interface.

In the configuration of the model, it must be possible to divide the application’s functionalities, and define different capabilities bases on the division. It must be possible to assign the default values which some capability interfaces may require

CPU cycle payment

The CPU cycle payment model enables the client to pay for a service by allowing a small program to be executed on his/hers computer, and sending the result back to the server. The model builds on the client’s good will, because the client will be able to cheat the system by sending wrong data back. The model must be able to supply the program on the client with data before the application starts; this enables the model to solve large problems that can be parallelized.

In the configuration of the model it must be possible to select a source (database) where the start data to supply to the clients is stored. It must also be possible to select a place to store the results form clients.

System design

The requirement section describes requirements to a system that will make it easier to develop, deploy and reuse components in a web service payment system. To realize these requirements, several design considerations must be taken. Many of these considerations will be discussed in this section and a design that conforms to the requirements will be made.

First the components of the system will be identified, and the interaction between the components will be discussed. The functionality of each component will be defined and listed as use cases. State diagrams will be used to describe flow in the components, if this is not obvious from the use cases. Class diagrams will be used to show the static structure of the components, but may be less detailed that the actual implementation requires. Sequence diagrams will be used to show the interaction between classes for each use case, again it may be less detailed that in the actual implementation. Where the components require interaction with the user through a graphically interface, simplified drawings will be used to show the interface.

Model of the system

The standard model of a web service system consists of a service, a client, and a discovery service. The SOAP specification specifies that a SOAP massage can be routed through a path of SOAP notes whit different responsibilities. This feature will not be used in this system; when a SOAP message is received it is always handled as if the node is the final destination. If more nodes are required to process a message, this must be handled in the web service application, and not by the SOAP message layer.

This system will not make use of a discovery service; once the system is running, it can therefore be described as a standard client/server system with one client and one server:

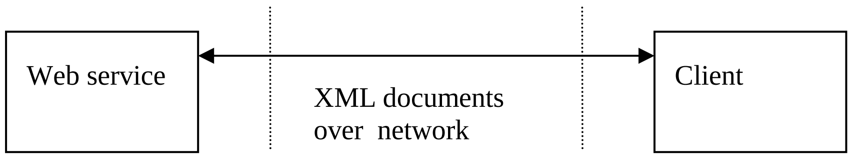


Figure 6

To fulfill the requirement of easy deployment of services, many of the existing tools supplied with the web service platforms described in state of the art section, could be used. But because the security model should be easy to change and reuse among different services, it requires a mechanism to integrate security models in existing applications in an easy way. One way to implement a security model in a system is to develop an API that makes the integration of the model easy to the application developer. The approach of separating a program's functionality in packages is commonly used, because it makes it easy to reuse the code in different applications.

If the security models were implemented as API's available to the application programmer the system would look like figure 7.

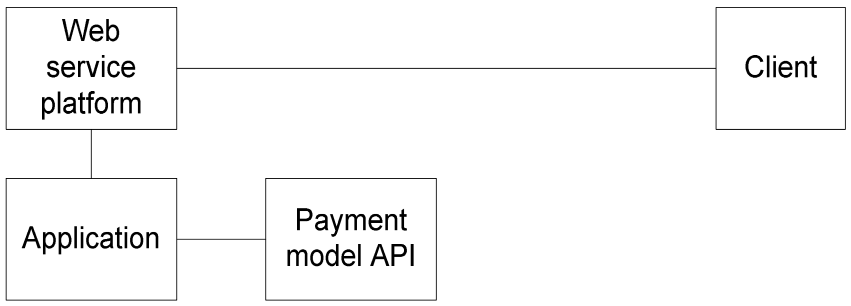


Figure 7 – model 1 Payment model as API

There are several advantages in this model:

- It is possible to use one of the many deployment programs available to most web service platforms.
- The API programmer does not have to concern about how the final application becomes a web service.
- The web service platform can generate the WSDL documents that describe the data types used to communicate with the service.
- The web service platform usually has tools to generate client interfaces and proxy classes, to make the client development easy.
- The application has full access to the security model, which means that it is easy to share information like authorization between the application and the security model.

The disadvantages of the model are:

- The security model is integrated in the application, which means that existing applications must be re-written to use the security model.
- The WSDL documents generated on the server will not contain information about security policy. Thus, it will most likely not be possible to develop a client that can communicate with the service, because the WSDL will not contain information about required authorization and encryption policy.
- The advantages of message level security cannot be obtained, because the security is implemented after the message is un-marshaled into types known to the application.

Especially the second disadvantage breaks with the web service concept of making the communication interface usable across businesses, platforms and language. It is clear that a mechanism to describe the security assertions must be applied to the model. Most Web service platforms generates WSDL documents of deployed web services when a client requests a document, the WSDL documents are not static files stored on the server. This means that the web service platform must be modified to generate WSDL documents containing security information. Further the deployment tool must be modified to pass security information to the web service platform, so it can generate the WSDL containing security information.

Instead of having the security models as an API available to the application programmer, the security models could be applied by wrapping the application with the security model, before the application was deployed as web service. The model would then look like figure 8.

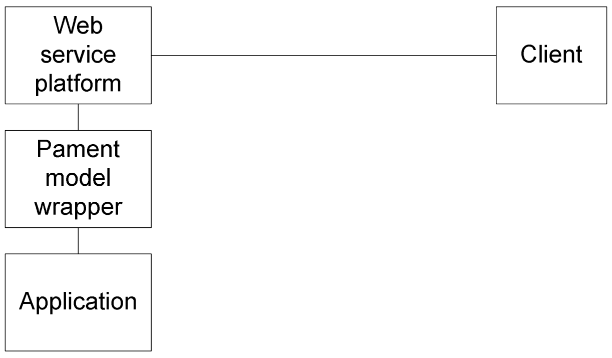


Figure 8 – Model 2 Payment model as wrapper

This model would solve the first issue of the API model, because the application would not need to be modified before deployment, but it would still be a problem to generate the correct WSDL for the web service. And message level security could not be obtained.

If the wrapper moves to the network layer and acts as a proxy, message level security is possible to implement in the proxy. The proxy can take the responsibility of generating WSDL documents, and can therefore generate WSDL documents with the necessary security information included. The model would look like figure 9.

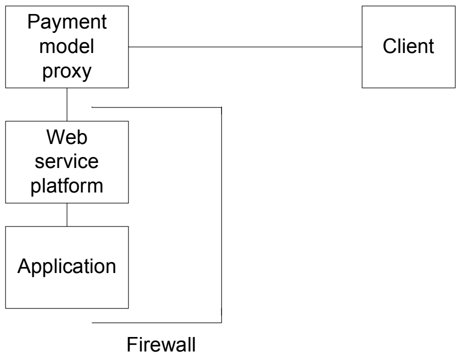


Figure 9 - Model 3 payment model as proxy

How the proxy speaks with the SOAP container is not important to the client, and the client should not be able to communicate directly with the SOAP container, so the network connections coming from other than the proxy, must be rejected by the SOAP container, this could be obtained by installing a firewall.

Another way to gain control over the web service in the message layer, is to develop the soap container from scratch, or modify an existing soap container. This approach binds the system to one specific web service platform, and requires modifications whenever a new version of the platform is developed.

The public interface to describe the communication with a web service, is of great importance in the web service domain, therefore a solution without possibility to describe the security assertions of the service, is not acceptable when developing a system for secure payments via web services. To maintain the possibility of choosing between different web service platforms and versions, the best solution is to implement the security layer as a proxy server in front of the web service platform, as described in solution 3.

A challenge in this implementation is that the application shall not only be deployed against the web service platform, but also make the proxy aware of how a request to a specific service should be handled and routed. This requires extensions to the deployment tool used to deploy applications on the web service, so it can handle deployment against the proxy as well.

The Proxy model

A standard way of implementing a proxy, is to create a server that listens on a specific network port, and handles all request on that port in some way. From the client a proxy should be transparent; how the proxy handles the request, is of no interest for the client as long as it produces a valid response.

If the web service security proxy should work in this way, it would require knowledge of the services behind. This is best illustrated by an example:

- The proxy works as proxy for two web services: Calculator and dictionary.
- The proxy listens on port 8880, the services are deployed on an application server that listens on another port.

What the proxy needs to do when a request comes to port 8880 is:

- Parse the request to see if the request is for calculator or dictionary.
- Look up in a local database to find the security model and configuration for the service.
- Apply the security model in the configuration found.
- Send the request to the web service.
- Apply the security model on the response from the web service
- Send the response to the client.

This approach has some disadvantages:

- The proxy needs its own network port.
- The proxy must be started as a separate application apart form the web service.
- The proxy must maintain a database of security models, security models configurations, web services and relations between them.

Instead of implementing the proxy as a standalone application, the proxy can be implemented as servlets on the application server that serves the web services. This will not require an extra network port, and the proxy will always be up and running with the web services, because they run on the same application server.

This implementation will require one servlet for each web service deployed on the web service platform, but the security models can be shared among more servlets.

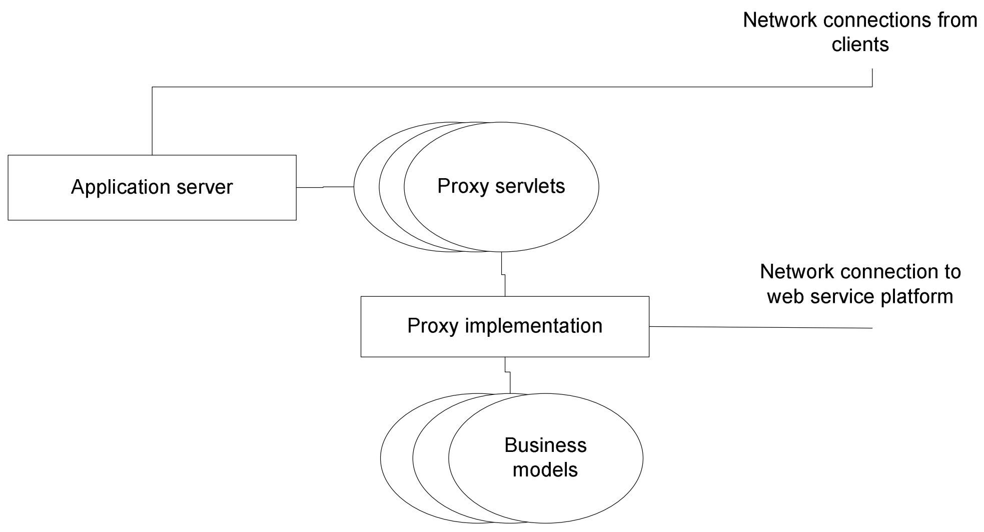

Network connections from clients
Application server
Proxy servlets
Network connection to web service platform
Proxy implementation
Business models

Figure 10 - The proxy as implemented as servlets

As previous described, the standard network proxy model will have to maintain a database, and look up for each request. This is not necessary in this model, because each servlet can hold information about which security model should be applied, and how it should be configured. Further the proxy will not have to parse the destination of the request, because it can also be given by the servlet. In this way one proxy servlet represents one specific web service with one specific security model applied in one specific configuration. If a web service should be accessible through more than one security model, it will be possible simply by deploying another proxy servlet pointing to the same web service.

Web services can consist of more operations (methods), and it may be required to apply different business models to each operation. To do this, different proxies must be deployed for the same web service

If one Java class is deployed as web service on the web service platform, publishing more than one method as web service, and it is accessed by different business model proxies, then it is possible to cheat the system. Because the servlet proxies do not look in the SOAP body for the destination of the incoming messages, it is possible to use a different proxy if more are deployed for the same Java class.

Example: A class containing two methods, m1 and m2, is published. m1 is configured to use the empty business model, and m2 is configured to use the secure payment model. Clients will be able to use the proxy servlet for m1 to access m2 without the security in the secure payment model.

To avoid this scenario the class must be deployed under two different names in the web service platform, which requires two deployment processes.

Having a proxy in front of each operation in a web service, makes the deployment procedure a little more complicated; it must now ensure that both the web service and the suitable proxies are deployed.

RPC and the business model proxy

Some business models may require a communication model that is not supported by standard RPC communication. An example of this is that for communicating secure, it may be required that the communicating parts must follow a key agreement procedure, which requires more than one RPC call. Because this step is concerned with the business model layer, it must be transparent to the application. Following figure shows a scenario, where one RPC from the application requires more calls in the proxy layer:

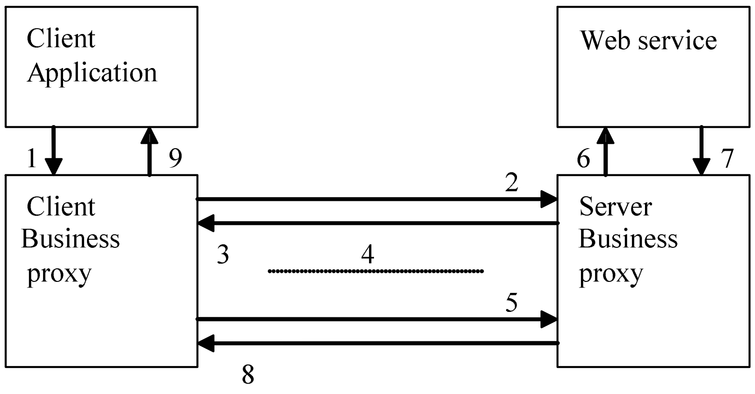


Figure 11

1) The client application sends a RPC request

2,3,4) The client business proxy is set to encrypt the channel, and therefore starts the key agreement sequence , which may be build on several RPC request to the server business proxy.

5) The client business proxy encrypts the message from the application and sends it to the server's business proxy.

6) The server business proxy decrypts the message and sends the message to the web service.

7) The web service processes the message and sends a response to the server's business proxy.

8) The server's business proxy encrypts the response and sends it to the client's business proxy.

9) The client's business proxy decrypts the response and sends the result to the client application.

From this scenario it is obvious that the business model proxy must be able to expand one RPC call in the application layer into more RPC calls in the business proxy. Further it is clear that it must be possible to deploy proxy servlets that do not have a web service in the application layer associated. Each of the proxy servlets must work as standalone web services, with WSDL documents associated, to conform to the web service standard.

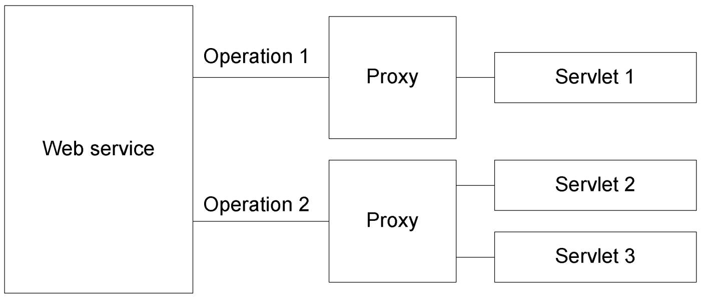


Figure 12

Figure 12 shows that one web service can result in more servlets, because one servlet will handle only one web service operation. The business model applied on operation two requires two RPC calls, therefore the servlet is split up in two parts, where only one calls the web service operation2.

Use case for the proxy

When the model of specific servlets holding the configuration for the proxy is used, the use cases concerned with configuration is moved to the deployment tool. This leaves two use cases for the proxy namely handling a request and answering requests for WSDL documents.

Use case 1 : Handling a WSDL requests from a client. **Actor:** Client application e.g. an Internet browser. **Pre. Req:** A proxy servlet must have been deployed, containing a valid configuration.	
Actor action:	**System action:**
Contacts the servlet on http with the parameter ?wsdl	
	Reads WSDL file generated during the deployment process, and sends it to the client
Processes the response.	
Variations of the normal flow	
Variation	**Result**
The system cannot read the WSDL file	The system will close the network connection to the client, and write an error to the application server.
Comments:	

Use case 2 : Handling a RPC requests from a client. **Actor:** Client application. **Pre. Req:** A proxy servlet must have been deployed, containing a valid configuration. A client capable of sending requests compliant to the WSDL document of the servlet

Actor action:	**System action:**
Sends a valid request to the servlet	
	1) Reads the configuration of the servlet 2) Processes the request in the right business model. 3) Sends the response to the web service application. 4) Processes the response from the web service application in the same business model. 5) Sends the response to the client.
Processes the response.	
Variations of the normal flow	
Variation	**Result**
The system receives an invalid request. (Not a SOAP message)	The system will close the network connection to the client, and write an error to the application server.
The configuration of the servlet defines that the request shall not be processed by the web service.	The system will process the message only in the business model.
An unexpected error occurs in the business model, while processing a request or response.	The system will close the network connection to the client, and write an error to the application server.
The web service application does not answer or produces an unexpected answer.	The system will close the network connection to the client, and write an error to the application server.
Comments: The reason for closing the network connection in all cases of errors, is that the business models are able to generate SOAP faults if a controlled error occurs. If an error occurs elsewhere something is wrong in the configuration of the server, and the client will most likely have little benefit of an error message.	

Class diagram for the proxy

Figure 13 shows the class diagram for the main classes of the proxy

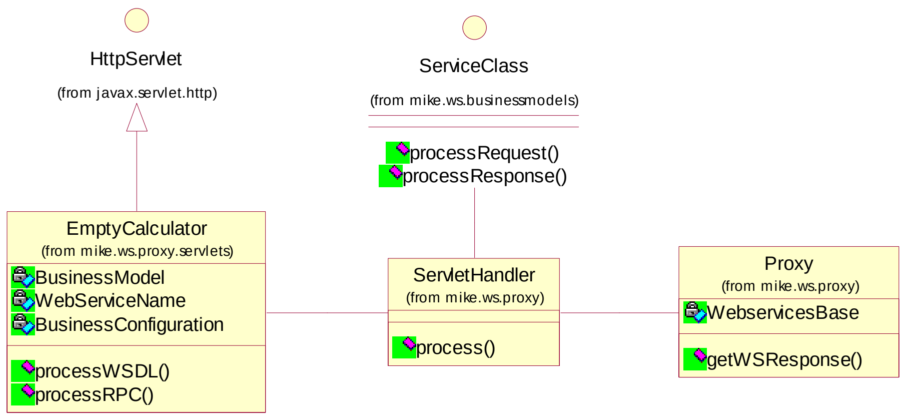


Figure 13

The EmptyCalculator class illustrates a proxy servlet generated during the deployment process of a calculator application, using an empty business model. Notice that it holds information about which business model and configuration that should be used. It also holds information about which web service application the request should be processed in. The class can answer to WSDL requests and RPC requests, only when processing RPC requests, the class needs to communicate with the Servlet hanlder.

The ServletHandler class is responsible for the flow of the RPC calls, the flow can vary depending on the configuration of the servlet as described in use case 2 for the proxy. It will make instances of the correct business model to process the request from the client and the response from the web service application.

The ServiceClass interface comes from the business model package; it defines the two methods to process requests and responses, which must be present when writing the application that forms the business model layer.

The proxy class holds information about where to find all the web service application, this information combined with the information in the servlet can be used to locate the right web service application. The class can send the request to the web service application, and receive the response.

Sequence diagram for the proxy

The sequence diagram shows the flow in the proxy described in use case two for the proxy. The sequence diagram for sending WSDL documents is not shown, because it only involves the client and the Servlet (EmptyCalculator).

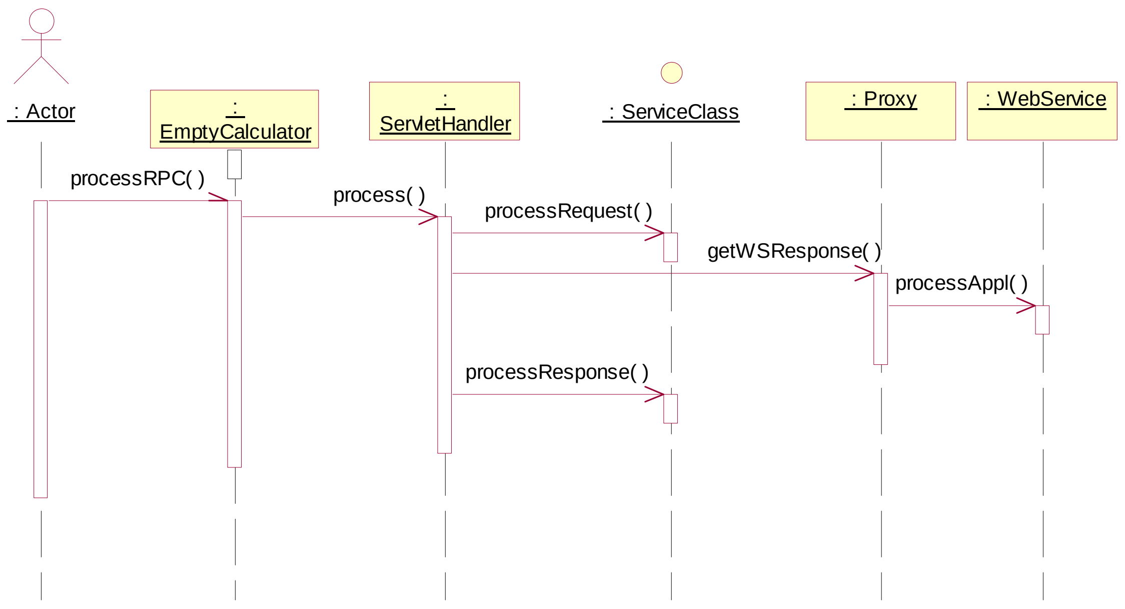


Figure 14

The web service class is not a class in the system, but represents the web service application that offers only one functionality: To process a request in the application deployed as web service. The diagram shows that the proxy class offers exactly the same functionality, and therefore works as a transparent component.

The Business model framework

In the requirement section the requirements to different payment models are listed, common to all of them is that they must be able to be used and configured in the system. That raises the demand of one common framework that can be applied to all business models, and be used in the system.

From the requirements and proxy design it is obvious that a business model consist of some code to process requests and responses, this code will be called the service component. Because a business model must be able to expand one RPC call into more RPC calls, a business model must be able to contain more than one service component.

The business models must be configurable during the deployment process. Not all models requires the same configuration flow, therefore a business model must be able define its own configuration flow. Some models may have a static flow while others have a configuration flow, depending on the selections made, therefore a business model must have a configuration flow controller that is used during the deployment phase.

To save the configuration collected in the configuration flow, a business model must have a configuration component. This component has the responsibility of parsing the selections made in the configuration flow, and produces a configuration that is understandable to the service components.

Because the proxy works in the network layer and business models may modify the structure of the messages, a business model must have a component that can generate WSDL documents that describe how to communicate with the proxy. The configuration of the business model can change the interface to the proxy; therefore the WSDL generator component must be able to build the WSDL based on the configuration of the business model.

The components described above must all be able to work together, and the deployment tool must have one access point that can tell where the components of the model are located. Therefore a business model must have a collection component, which contains all the static references.

The class diagram in figure 15 shows the relations of the components described. Notice that the components described are interfaces, which makes it possible to developers of business models to implement the interface and use the model in the system.

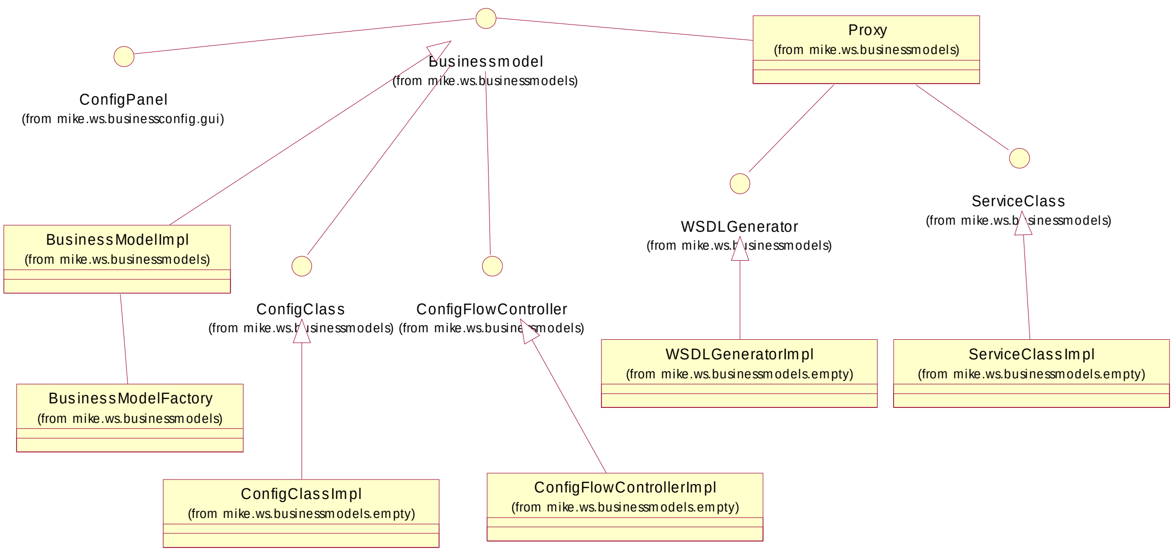

ConfigPanel
(from mike.ws.businessconfig.gui)
businessmodel
(from mike.ws.businessmodels)
Proxy
(from mike.ws.businessmodels)
BusinessModelImpl
(from mike.ws.businessmodels)
WSDLGenerator
(from mike.ws.businessmodels)
ServiceClass
(from mike.ws.businessmodels)
ConfigClass
(from mike.ws.businessmodels)
ConfigFlowController
(from mike.ws.businessmodels)
BusinessModelFactory
(from mike.ws.businessmodels)
WSDLGeneratorImpl
(from mike.ws.businessmodels.empty)
ServiceClassImpl
(from mike.ws.businessmodels.empty)
ConfigClassImpl
(from mike.ws.businessmodels.empty)
ConfigFlowControllerImpl
(from mike.ws.businessmodels.empty)

Figure 15

The implementation classes in the diagram are all located in a different package (in this case the empty model) except the BusinessModelImpl and Proxy. The implementation of BusinessModel is not located in the model package because it does not contain functionality, only descriptions and references. The BusinessModelFactory is a factory that builds businessmodels, this makes it possible to store the information of the businessmodel in an XML file, and construct an instance of the model from this file.

Notice that the proxy class is not in the same package as the one from the proxy design. The proxy class in this package is used as holder-class for the information related to one proxy. One proxy must have a WSDL generator and a service class as previous discussed, and one business model can have more proxies to be able to expand the RPC call.

Only methods in the ServiceClass are defined in the diagram, because they are given from the design of the proxy, the methods in the other interfaces will be defined in the design of the deployment program.

The use cases and sequence diagrams for the business model framework, are depending on the implementation of the model. These components can therefore be found in the design of each business model.

Business model XML description

As described the business models must have a description file that can described the static components of one business model. It must be possible to make instances of the implementations of the BusinessModel interface from an XML file. The file must therefore contain information about all attributes in a business model and associated classes. Following describes the structure of an XML file that can be used to make instances of the BusinessModel interface:

The XML document must contain a root element called businessmodel with following attributes:

- name: The name of the business model
- description: The description of the model.
- version: The version of the business model.

- configflowcontroller: The implementation class of the ConfigFlowController interface.
- configClass: The implementation class of the ConfigClass interface.

The root element must also contain one or more proxy elements that describe one proxy servlet The proxy element must contain following attributes:

- ID: An integer number that defines the order of which the proxies must be deployed and used.
- name: The suggested name for the proxy servlet.
- serviceclass: Name of the implementation of the ServiceClass interface to use in the proxy.
- wsdlgenerator: Name of the implementation of the WSDLGenerator interface to use to generate the WSDL file for the servlet.
- Processinwebservice: property that defines if the proxy should call the web service application, or if it should process the request only in the business model layer.

The root element can also contain references to implementations of the configpanel interface. The structure of these elements is discussed in the design of dynamic configuration panels.

The web service deployment tool

As discussed in the general model of the system, the deployment tool for the selected web service platform must either be extended or re-developed, to fulfill the requirements to the system with a business model proxy. Many web service platforms have a deployment tool that can deploy web services by executing a program with the correct parameters. Others require user interaction. If the existing deployment tool can work as a command line program, or do deployment from a script file, it will be relative easy to extend its functionality by wrapping a new deployment around as showed in figure 16.

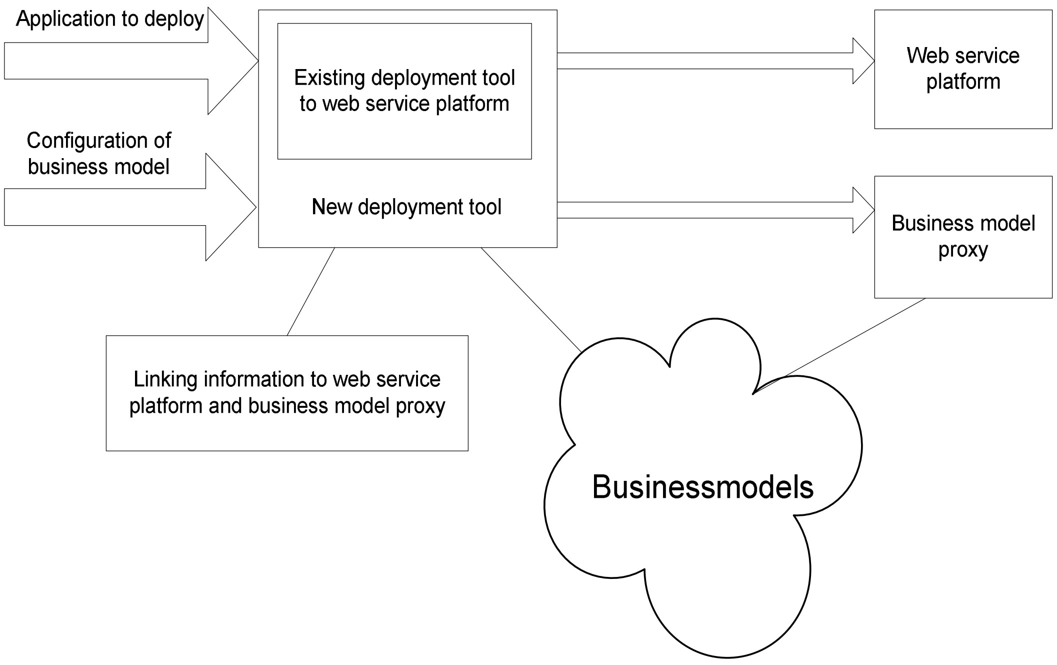

Application to deploy
Configuration of business model
Existing deployment tool to web service platform
New deployment tool
Web service platform
Business model proxy
Linking information to web service platform and business model proxy
Businessmodels

Figure 16 - Deployment tool using existing tool.

If the existing deployment tool for the web service platform requires user interaction to deploy web services, the tool must be used separately to deploy the application against the web service platform. In the following design it is assumed that it is possible to re-use the existing deployment tool as shown in the figure above.

The figure shows a system that is able to integrate applications, with business models (security/payment) and web service knowledge into web services, as described in the requirements section. The main purpose of the tool is to make it easy to deploy a application, the more static components like web service linking knowledge and business models are therefore represented as resources. The resources could be less static, but it would make the deployment process more complicated if e.g. the user should point to the proxy installation directory every time an application should be deployed. The drawback of the static resources is the flexibility of the tool, if the tool should be used to deploy to different systems, it would require low level configuration of the tool every time the system changes.

The deployment tool can therefore be described in three main use cases:

1 Deploying applications as web services using a business model.

2 Making business models available to the tool.

3 Changing static settings for the tool.

The first use case is complicated, and can be divided into more use cases:

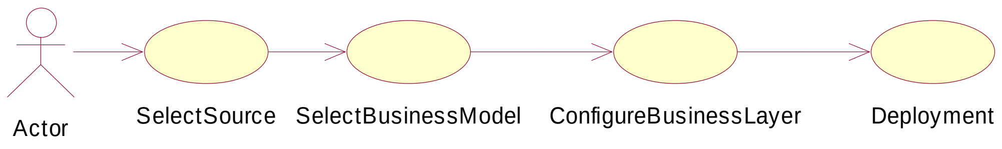


Figure 17

Use case diagram for deploying services

As the diagram shows, the use cases depends on each other, it would therefore be natural to implement the deployment tool as a state machine.

Use case descriptions

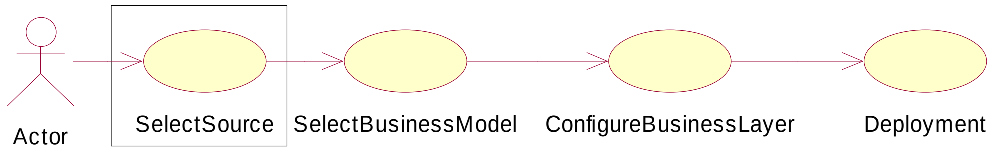


Figure 18

Use case 1.1 : Selecting an application to deploy. **Actor:** A person who whish to deploy an application. **Pre. Req:** The deployment tool has been correctly configured (U.C. 3) **Expected result:** The system is in a state where U.C, 1.2 can begin.	
Actor action:	**System action:**
Starts the deployment tool, by executing a command from a command line.	
	Reads the configuration of the deployment tool and shows a graphical user interface containing two text input field and a button. The GUI also shows a button panel containing three buttons : Back, Cancel and Next.
Presses the button to browse for a class file (the application)	
	Shows a file dialog.
Navigates to a class file and selects it.	
	Updates the first text field with the complete path to the selected class file
Types the base directory of the selected class. The base directory is where the package structure for the selected class begins. Presses the Next button	
	Removes all the GUI components

	except the button panel. Reads the names of all business models applied to the system. Analyses the selected class and shows a drop down list for each public method in the class. Each drop down list contains all the names of applied business models, and a "Do not deploy" item.
Variations from the normal flow	
Variation	**Result**
The actor selects an other file than a class	The system does not update the text field, and informs the user of valid file types.
The selected file is not a valid Java class, or the file can not be read.	The system informs the user, and does not update the GUI.
The base directory for the selected class is incorrect.	The system informs the user, and does not update the GUI.
The class does not contain public methods.	The system shows an empty list.
The actor presses the back button.	Nothing.
The actor presses the cancel button	The system closes.
Comments:	

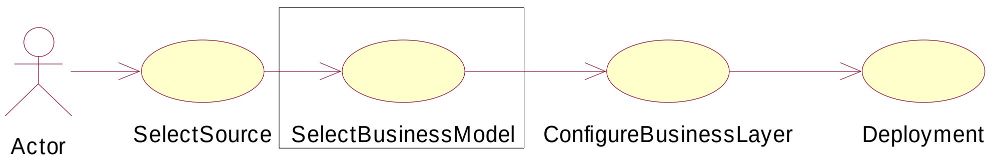


Figure 19

Use case 1.2 : Selecting a business model to used. **Actor:** A person who whish to deploy an application. **Pre. Req:** U.C. 1.1 has ended without variations. **Expected result:** The system is in a state where U.C, 1.3 can begin.	
Actor action:	**System action:**
For each method in the list: Selects which business model should be used. If a method should not be published as web service: Selects the "Do not deploy" item. Presses next	
	For the first method that have another business model than "Do not deploy",

	load the business model and ask it's configuration flow controller for the first configuration panel. Removes all GUI components except the button panel Shows the first configuration panel from the business model.
Variations from the normal flow	
Variation	**Result**
The actor selects "Do not deploy" for all available methods.	The system closes.
The list is empty and the actor selects next.	The system closes.
The actor presses the back button	The system returns to the state in U.C 1.1 before the next button was pressed.
The actor presses the "Cancel" button	The system closes.
Comments:	

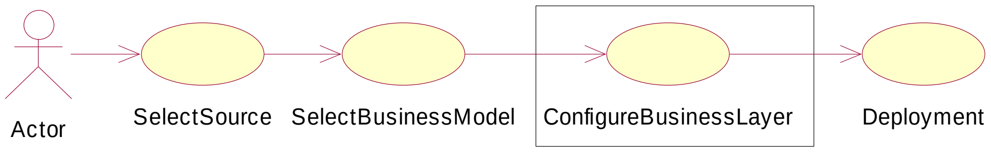


Figure 20

Use case 1.3 : Configuration of the business model. **Actor:** A person who whish to deploy an application. **Pre. Req:** U.C. 1.2 has ended without variations. **Expected result:** The system is in a state where U.C, 1.4 can begin.	
Actor action:	**System action:**
Fill in all necessary fields of the panel and presses the "Next" button.	
	Calls the business models configuration class and asks it to store the information selected in the panel. Removes all GUI components except the button panel. Calls the business models configuration flow controller, and asks for the next panel. Show the panel. If the controller returns an empty panel: Check if more methods should be deployed and re-run the use case for them, else: Display a text input field.

Variations from the normal flow	
Variation	**Result**
The actor presses the "Cancel" button	The system closes.
The actor presses the back button	The system calls the business model flow controller to get the previous panel, and shows this.
Errors occur in some of the business model components.	The system informs of the error and closes.
Comments: The flow in the business model configuration is described in the design of each business model.	

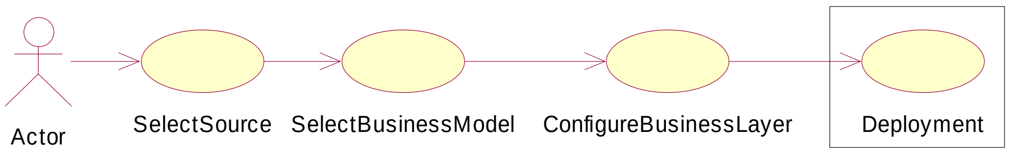


Figure 21

Use case 1.4 : Deployment. **Actor:** A person who whish to deploy an application. **Pre. Req:** U.C. 1.3 has ended without variations. **Expected result:** A web service deployed on the web service platform and a proxy to call the web service deployed.	
Actor action:	**System action:**
Types the name of the application part of the web service in the text field. And presses Next	
	Stores the name of the service, and deploys the selected application from U.C. 1.1 on the web service platform. Displays a new text input field. Asks the user to type the name of the proxy servlet for the first method selected in U.C. 1.2
Fills in the name of the first proxy, and presses Next.	
	If only one method was selected in U.C. 1.2, the system will start the deployment. Else it will continue to ask for names of the proxy servlets. Deployment: For each method selected in U.C. 1.2 the system will deploy proxies. Each method can result in more servlet proxies, if the selected business model requires more that a single RPC call.

Variations from the normal flow	
Variation	**Result**
The actor presses the “Cancel” button	The system closes.
The actor presses the back button	The system returns to the beginning of U.C.1.2
Errors occur in some of the business model components.	The system informs of the error and closes.
Comments: The exact flow of deployment is described later by a state diagram.	

Use case 2 : Appling a business model. **Actor:** A person who whish to apply a business model. **Pre. Req:** A business model and configuration file has been developed.. **Expected result:** The list of available business models in U.C. 1.2 contains the name of the business model.	
Actor action:	**System action:**
Copies an XML description file of the business model to the business model description directory. Copies the compiled business model implementation to the deployment tool’s program directory, and to the business model proxy program directory.	
	Next time the deployment tool is started, the business model is available.
Variations from the normal flow	
Variation	**Result**
The business model description file is not valid	The business model will not be available in the deployment tool, or the deployment tool will generate an error when U.C. 1.3 is executed.
Comments:	

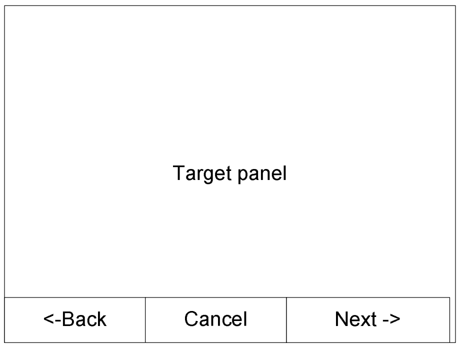
Use case 3 : Changing configuration of the deployment tool. **Actor:** A person with knowledge of the systems to deploy to. **Pre. Req:** The deployment tool is installed on a computer. **Expected result:** The deployment will use the configuration next time it is started.	
Actor action:	**System action:**
Edits the configuration file for the deployment tool, using a text or XML editor	
	Next time the deployment tool is started, configuration will be used.

Variations from the normal flow	
Variation	**Result**
The structure of the configuration file is modified so the deployment tool does not understand the content.	The deployment tool will show an error, when executing.
Comments:	

Graphical user interface design

The use cases of the deployment tool describes the tool's interaction with the user, all normal interaction is done via a graphical interface, only configuration of the system is not done by the tool's user interface. The graphically user interface consists of two parts: The static part that always will be executed independent of the users selections, and the business model configuration part that depends on which business models the user selects.

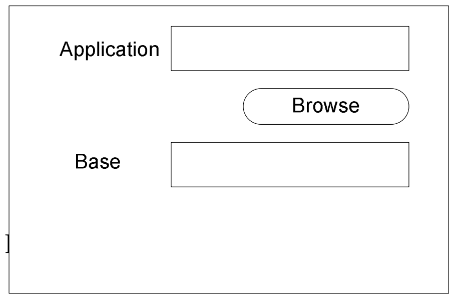
The static panels:



All windows in the system consist of a target panel, and a button panel. The target panel is where the panel of the systems state is shown. The button panel is used to navigate between the states of the system.

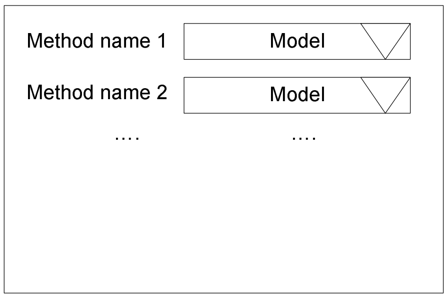
Following panels are all panels to be displayed in the target panel.

Figure 21



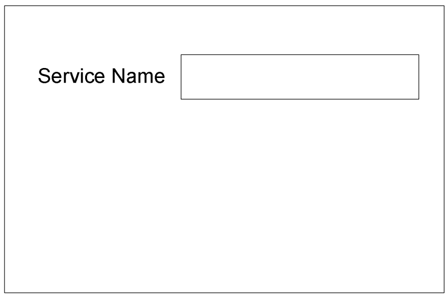
Panel for use case 1.1: The user must select a class file, and type the base of the class. The file dialog that can be used to select a class file is not shown, because it is a standard file dialog for the operation system

Figure 22



Panel for use case 1.2: The user must select which business model to use for each public method. If the selected class contains more methods than the panel can contain, a vertical scroll bar will become visible.

Figure 23

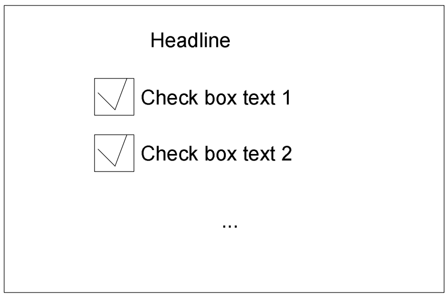


Panel for use case 1.4: The user must type the name to use when deploying the web service. A similar panel is used to get the name of the proxy servlets.

Figure 24

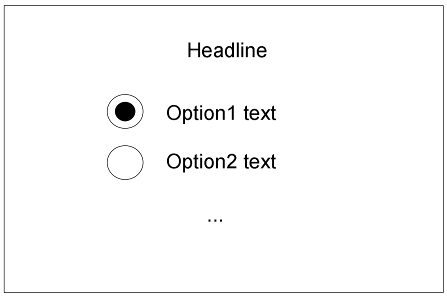
Dynamic configuration panels

Because the deployment tool must be able to handle different business models, with different configuration flows, it must be possible to define configuration panels when developing business models. The deployment tool must provide some standard components, which can be used by business models to build configuration panels that can be used with the tool. A dynamic panel can consist of following components:



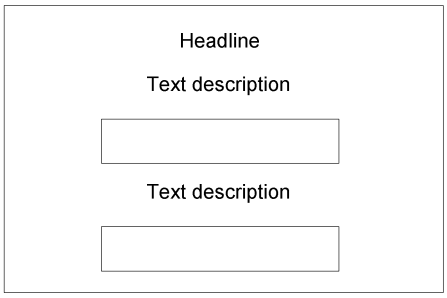
A Checkbox panel can contain a headline and zero or more check boxes. A check box has a check field and a text. The first check box will appear in the top of the panel, following check boxes will appear below.

Figure 25



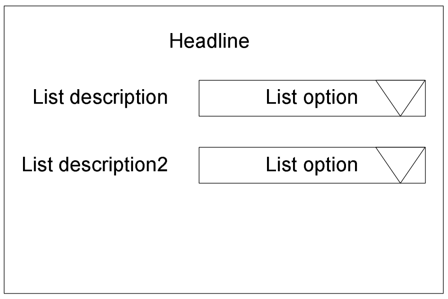
An Option panel can contain a headline and zero or more option buttons. Only one option can be selected at a time. The first option will appear in the top of the panel, following options will appear below.

Figure 26



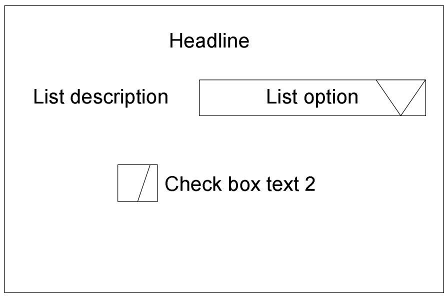
A text field panel can contain a headline and zero or more text input fields. A text input field has a text description and an input field below. The first text field will appear in the top of the panel, following text fields will appear below.

Figure 27



A choice box panel can contain a headline, zero or more choice boxes and zero or more options. A choice box contains a description text. The options for all choice boxes in the panel are the same. The first choice box will appear in the top of the panel, following choice boxes will appear below.

Figure 28



A base panel can contain a headline and zero or more configuration panels. The example showed in figure 29, is a base panel with a headline, a choice box panel and a check box panel without headlines.

Figure 29

Interaction with business models

The design of the business model framework introduces the ConfigFlowController interface. Implementations of this interface must be able to generate the configuration panels for a business model. Because the flow within a business model is relative static, the coding of the ConfigFlowController can be very simple, if the definition of the panels is stored in the configuration file for the business model.

When loading a business model from a configuration file, the BusinessModelFactory will make instances of the classes, and make a list of static configuration panels available to the ConfigFlowController. Only if a dynamic flow is required by the business model, the ConfigFlowController will have to actually build the panel itself.

The interface of the ConfigFlowController must have one method:

- `ConfigurationPanel getNextPanel(currentpanel):` To implement a dynamic flow, the implementation of the interface can get access to collected properties in the ConfigClass, through the BusinessModel class. To implement a static flow, the flow controller can simply ask the business model for a specific static panel.

The ConfigClass in a business model is the component that collects information from the configuration panels. The definition of the GUI components makes it clear that the output from a configuration panel depends on the type of panel: A text input field returns a String and a check box returns a Boolean. The ConfigClass must be able to convert the output of the configuration panels to a configuration understandable to the proxy. To do the conversion the ConfigClass must know the structure of all configuration panels, and the configuration panels must have a unique id, so the ConfigClass can do the conversion.

The interface of the ConfigClass must have two methods:

- `addProperties(ConfigurationPanel)` : The configuration panel has an id that can be used by the configcontroller to determine how to convert the panels input fields into configuration properties.
- `Configuration getProperties():` When the configuration of a business model is done, the configcrontroller must be able to return the collected properties, to use when deploying the proxy servlet.

The configuration of a business model must primarily be used in the proxy servlet, but the structure of the WSDL file that described how to interface to the proxy servlet, may also depend on the configuration of the servlet. Therefore the WSDL generator class of a business model must have access to the configuration of the business model. The easiest way to obtain this is by giving it as argument when generating the WSDL. The WSDL generator is only responsibly for the changes that the business model will do to the WSDL file, therefore the WSDL for the application without the business model applied must also be available when generating the WSDL.

The interface of the WSDLGenerator must have one method:

- `WSDL generateWSDL(ApplWSDL, configuration)` : An implementation of the WSDLGenerator interface, can generate the WSDL document for one proxy servlet. If a business model consist of more servlets (as discussed in the proxy design section), the business model must have more implementations of the interface.

Class diagrams

The class diagrams in this section show the overall structure of the program. The diagrams are kept very simple, and can not be used as a programming reference. For a more specific description of the classes please refer to the implementation section or the Java documentation on the CD.

The class diagram for the deployment tool can be divided into four categories:

- GUI Classes related to the static flow of the tool.
- Classes that does the functionality of the tool.
- GUI Classes available to the business models.
- Business model classes

The class diagram for the business model classes has been discussed in the business model design section, and will not be described here.

Static GUI classes

The static GUI classes are already partly described in the design of user interfaces section. The purpose of the class diagram is to see the overall structure, and to show how the configuration panels from the business models can be used.

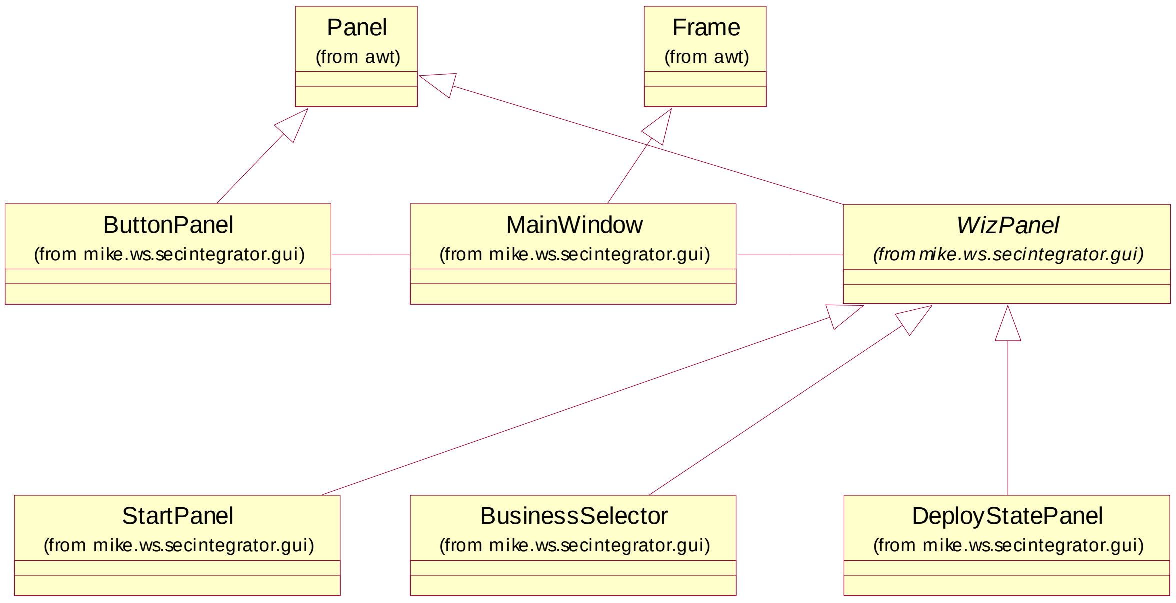

Panel
(from awt)
Frame
(from awt)
ButtonPanel
(from mike.ws.secintegrator.gui)
MainWindow
(from mike.ws.secintegrator.gui)
WizPanel
(from mike.ws.secintegrator.gui)
StartPanel
(from mike.ws.secintegrator.gui)
BusinessSelector
(from mike.ws.secintegrator.gui)
DeployStatePanel
(from mike.ws.secintegrator.gui)

Figure 30

Notice that the MainWindow only uses two panels: The static button panel, and a WizPanel that is super class for all panels used in the target area of the main window. The three panels that are used in the static flow of the tool therefore extend the abstract class.

Functionality classes

The functionality classes are all related to the MainWindow class, either directly or through other functionality classes. The reason for this is that the MainWindow is the class where the state machine that controls the overall flow of the system is implemented.

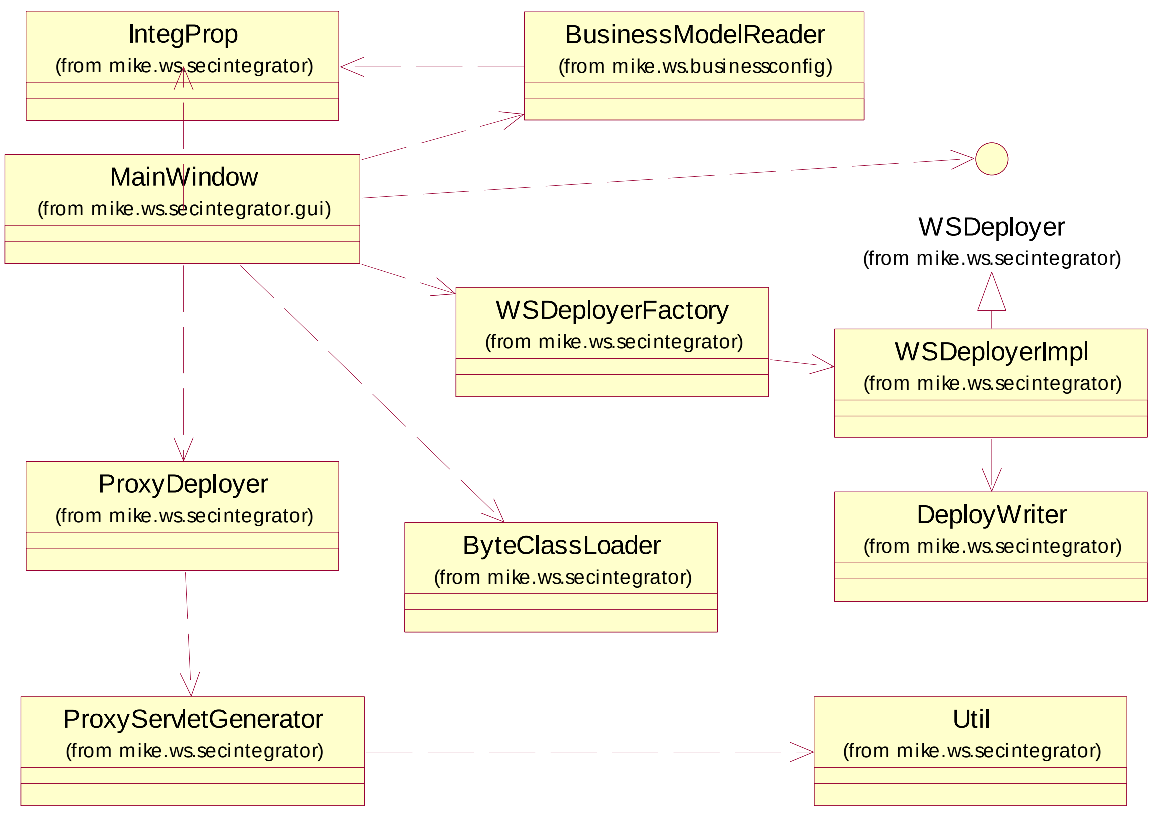

IntegProp
(from mike.ws.secintegrator)
BusinessModelReader
(from mike.ws.businessconfig)
MainWindow
(from mike.ws.secintegrator.gui)
WSDeployer
(from mike.ws.secintegrator)
WSDeployerFactory
(from mike.ws.secintegrator)
WSDeployerImpl
(from mike.ws.secintegrator)
ProxyDeployer
(from mike.ws.secintegrator)
ByteClassLoader
(from mike.ws.secintegrator)
DeployWriter
(from mike.ws.secintegrator)
ProxyServletGenerator
(from mike.ws.secintegrator)
Util
(from mike.ws.secintegrator)

Figure 31

The WSDeployerFactory is a factory class that can build web services deployers. The factory design pattern is chosen because it must be possible to easily develop new deploy-classes if the system should be used with different web service platforms. The Factory builds instances of the interface WSDeployer that describes the functionalities a web service deployer must have. The WSDeployerImpl is an example of an implementation of the interface, it uses a DeployWriter to write files used to use when deploying.

The deployment tool must be able to analyze classes that are not part of the system, to display the methods available for web service deployment. The ByteClassLoader must be able to convert a byte stream into an instance of a class, so the class can be analyzed.

To make proxies for the application deployed by the WSDeployer, the ProxyDeployer class can be used. The class will use the ProxyServletGenerator to generate and compile a proxy servlet with the correct name and configuration, and then copy it to the servlet container of an application server. Because the ProxyServletGenerator must be able to compile the servlet, it must call an external compiler.

The Util class makes it possible to call external applications, the class make the deployment tool wait for the external application to end, before it continues its execution. This is useful e.g. when the compiled servlet must be copied; this can not be done before the compiler has compiled the source code.

The BusinessModelReader class is the class that can read an XML description of a business model, and make an instance of the model, if the correct classes are available.

The IntegConfig is a class that can read XML documents and make the content available in a standard Property manner to the application. The class is constructed using Singleton design pattern, which makes the same instance of the class available to the entire application. The class diagram only shows relations with the MainWindow and BusinessModelReader, other classes also have relations, but they are not shown in the diagram due to the complexity

GUI Classes available to the business models

The class diagram of GUI classes available for business models is rather large due to the number of panel types (described in the design of business model configuration panels). The diagram showed here only contains one type of configuration panel (Text field panel), but the structure is the same for all panels. Refer to the CD-Rom for the complete class diagram.

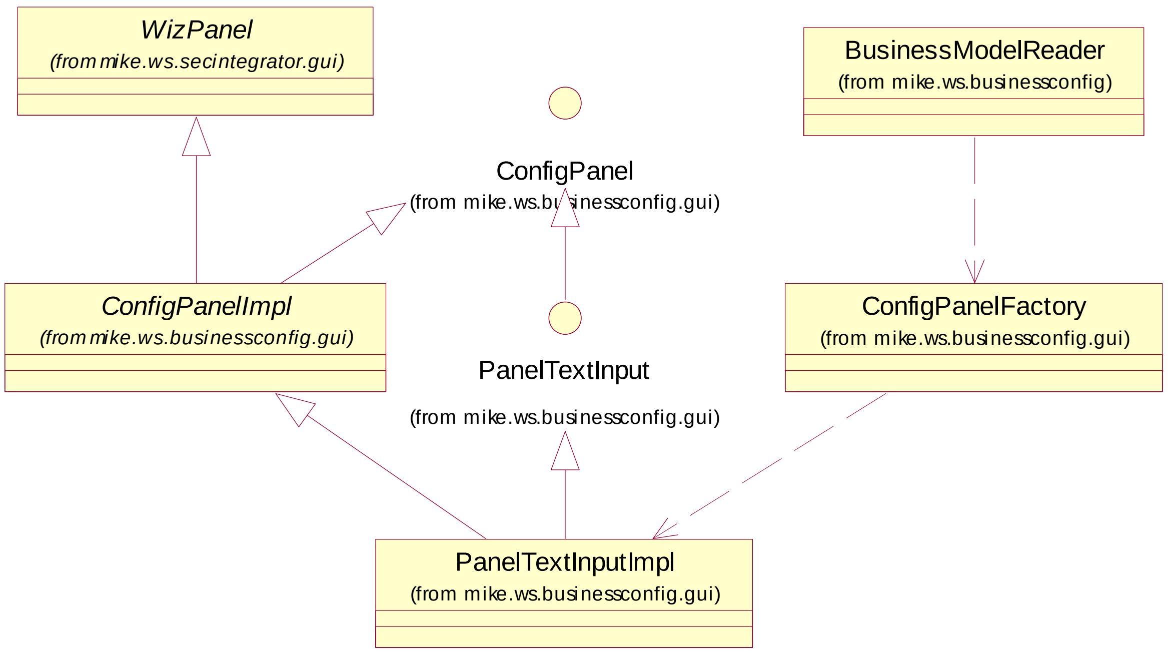

WizPanel
(from mike.ws.secintegrator.gui)
BusinessModelReader
(from mike.ws.businessconfig)
ConfigPanel
(from mike.ws.businessconfig.gui)
ConfigPanelImpl
(from mike.ws.businessconfig.gui)
PanelTextInput
(from mike.ws.businessconfig.gui)
ConfigPanelFactory
(from mike.ws.businessconfig.gui)
PanelTextInputImpl
(from mike.ws.businessconfig.gui)

Figure 32

The ConfigPanel interface defines the behavior that all business model configuration panels must implement. As described in the design of the ConfigClass, all configuration panels must have a unique ID and other attributes, this interface can be used to get these attributes.

The ConfigPanelImpl is an abstract implementation of a ConfigPanel. The Class also contains some general methods that can be used to compose GUI components with behavior like describe in the GUI design section. The class extends the WizPanel described in the static GUI class diagram, this makes it possible to use sub-classes in the deployment tool.

The PanelTextInput interface extends the ConfigPanel, to specify the behavior of a Text field panel. The interface can be used by the ConfigClass, to access the attributes of a Text field panel, without having knowledge of the entire implementation.

The implementation of the PanelTextInput, PanelTextInputImpl, extends ConfigPanelImpl, because it makes the usable in the deployment tool. It also contains the implementation of the ConfigPanel, which makes the coding a little easier. Beside the implementation of the methods described in PanelTextInput, PanelTextInputImpl contains some construction methods that are not visible through the interface.

The ConfigPanelFactory is a factory class for all configuration panels. It can build panels from XML elements that describe the content of the panels. The class makes an instance of an implantation, and configures this instance after the roules described in the XML element. The BusinessModelReader, also described in the functionality class diagram, makes use of the ConfigPanelFactory to generate the list of static panels available in a business model.

XML descriptions of dynamic panels

In the design of the business model framework, it is described that the XML file describing the business model must be able to describe the static configuration panels available to the business model. All static panels must be defined in the root element of the business model file, and use an element called configpanel. A configpanel element represents a Base Panel (See design of dynamic configuration panels on page 52) and must contain following attributes:

- ID: The suggested order which the ConfigFlowController must use the panel
- headline: The headline of the base panel.

As base panels can contain zero or more configuration panels, the element configpanel can contain elements that describe dynamic configuration panels. Following elements describes configuration panels:

Checkboxpanel:

ID: The order of adding the panel to the base panel.

Headline: The headline of the panel

To add check boxes to the panel, the element checkbox must be used. The attributes of a checkbox are ID and text, where ID is the order which the checkbox is added to the panel, and text the text displayed.

Optionpanel:

- ID: The order of adding the panel to the base panel.
- headline: The headline of the panel

To add options to the panel, the element option must be used. The attributes of an option are ID and label, where ID is the order which the option is added to the panel, and label the text displayed.

Textboxpanel:

- ID: The order of adding the panel to the base panel.
- headline: The headline of the panel

To add text fields to the panel, the element textfield must be used. The attributes of a textfield are ID and label, where ID is the order which the text field is added to the panel, and label the text displayed.

Choicepanel:

- ID: The order of adding the panel to the base panel.

- headline: The headline of the panel

To add choices the choice element must be used. The attributes of a choice element are ID and text, where ID is the order which the choice elements must be displayed and text the text used.

To add choice boxes to the panel the element choicebox must be used. The attributes of a choicebox are ID and text, where ID is the order which the box is added to the panel and text the text displayed in front of the choices. Choice boxes must be added to the panel before the choices defined by the choice element are displayed.

Sequence diagrams

The class diagrams did not describe the methods or attributes of the classes. The sequence diagrams will show some of the methods required, but more may exists which is discussed in the implementation section.

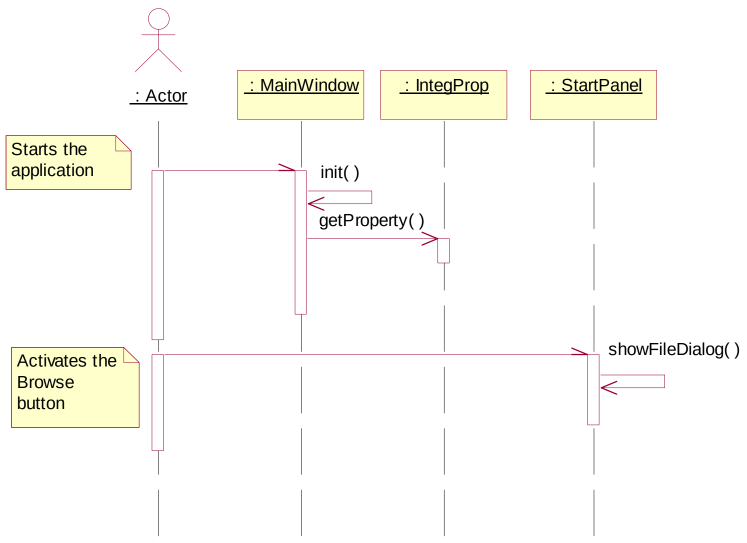


Figure 33 - Sequence diagram for use case 1.1 -first diagram

The diagram shows the first part of use case 1.1. The init method must set up the environment of the application; to do this requires knowledge of size and placement of the GUI components. Some of the parameters to set up the system can be hard coded in the MainWindow class, while other can be stored in the configuration file, and retrieved through the IntegProp class. The diagram does not show the construction of all GUI components, due to the number of classes involved. It is obvious that the init method must construct all GUI components, and add actions listeners to all buttons. The methods called on actions from the button panel must be placed in the MainWindow class, because this class controls the flow of the application. The action method for the StartPanel can be placed in the StartPanel itself, because the action of browsing for files is only related to that panel

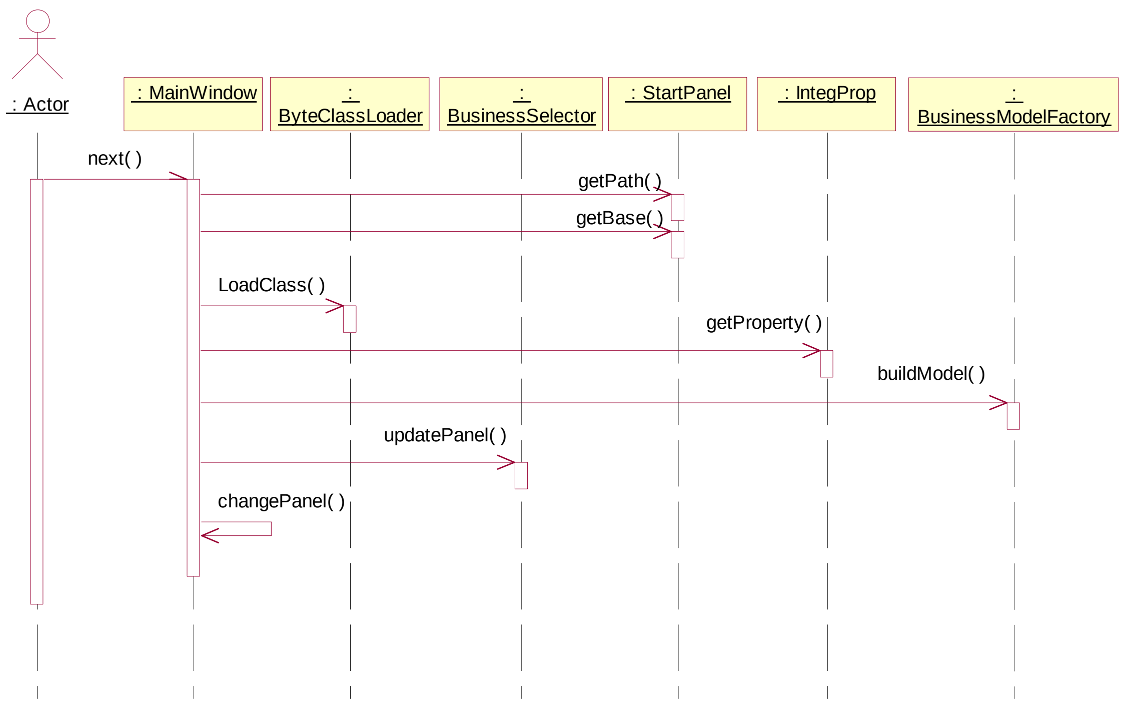


Figure 34 - Sequence diagram for use case 1.1 -second diagram.

The diagram starts when the user has selected the application class and base, and the next button. The application information from the start panel, is collected and stored in the MainWindow class. Usually when loading a class in an application, it is required that the class is in the Java Virtual machines classpath. The tool must be able to load classes that is not in the JVM's classpath, to analyze the content of the class, the loadClass method in the the ByteClassLoader must be able to do this.

The systems configuration contains information about where the business model configuration files are located, and is asked for this property to be able to load all business models. The MainWindow runs through all XML files in the business model directory, and tries to build a business model instance using the BusinessModelFactory. All the instances is stored in the MainWindow.

The method information from the loaded application class and the names of the business model instances is used to construct a BusinessSelector panel. The changepanel removes the StartPanel from the main window, and inserts the new panel in its place.

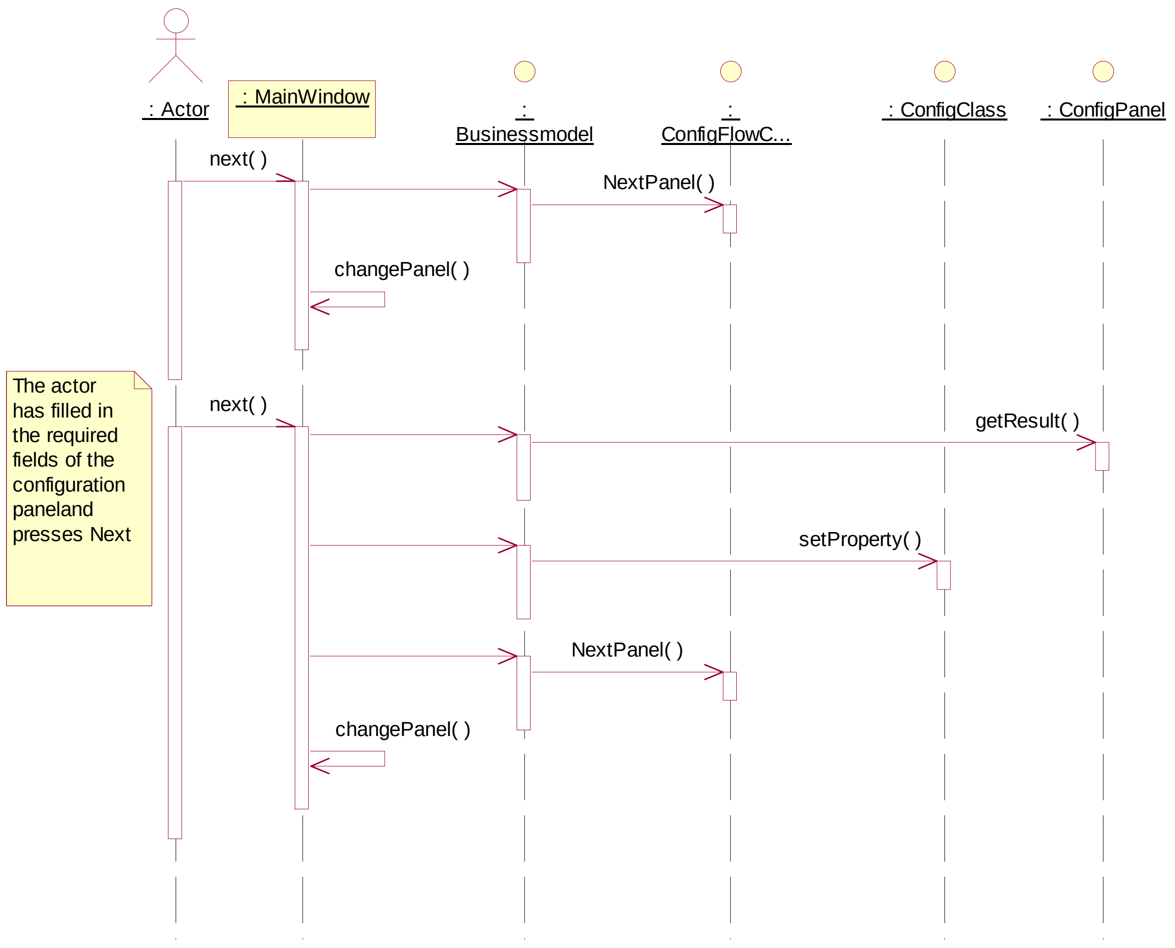


Figure 35 - Sequence diagram for use case 1.2

The diagram starts when the actor has selected which methods to make available, and which business models to use. What the diagram does not show is that the MainWindow stores the information about the selected methods and business models.

Notice that the MainWindow accesses all classes in the business model through the BusinessModel interface. The panel is changed to the first configuration panel from the first selected business model, the flow of configuration panels can now be controlled by the business model. Every time the actor changes to the next panel in the configuration flow, the result of the panel is collected by the MainWindow and passed to the ConfigClass, which will format the result as properties understandable to the ServiceClass implementation. This will be done for each panel returned form the ConfigFlowController.

When one business models has been configured, the ConfigFlowController will not return more configuration panels, and the MainWindow will load the next business model, if more methods have been selected for deployment. When the last selected business model has been configured, the MainWindow will change the to deployment state, which is described in following state diagram.

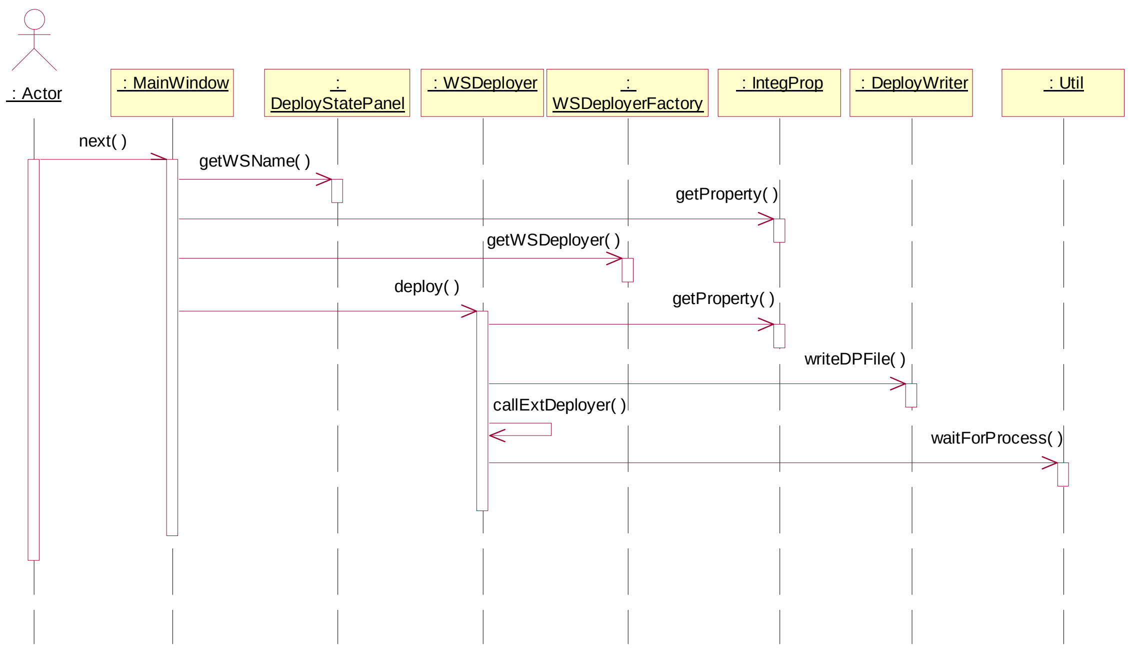


Figure 33 - Sequence diagram for use case 1.4 first diagram

The state diagram starts when the actor presses the next button, after having given a name to the web service. The name is used as identification of the application in the web service layer. The MainWindow quires the deployment tools configuration to get information about the web service platform where the application must be deployed.

The WSDeployerFactory returns an instance of a WSDeployer that can deploy applications on the selected web service platform. The rest of the application deployment process is done using the WSDeployer interface. In this case the implementation of the interface is for deployment to a AXIS web service platform.

The AXIS deployer quires the deployment tools configuration to get information about the AXIS installation e.g. where to place the application class, and how to get WDSL files from the platform. The application delivered with AXIS to deploy web services is a command line Java application that uses a configuration file as descriptor for how to deploy the application. The DeployWriter can write these files based on the configuration passed form the MainWindow and the deployment tool’s configuration.

The WSDeployer calls the AXIS deployment tool as an external process, because the Java program terminates the JVM when the application has been deployed. If the application was not called in an external process, it would result in the business model deployment tool would terminate as well. The synchronization between the two processes is handled by the Util method waitForProcess that pauses the execution of the deployment tools process, until the AXIS process has ended.

Unlike the deployment of the proxy servlets, deployment of the web service application must only be done once, independent of how many methods and business model that have been selected for deployment.

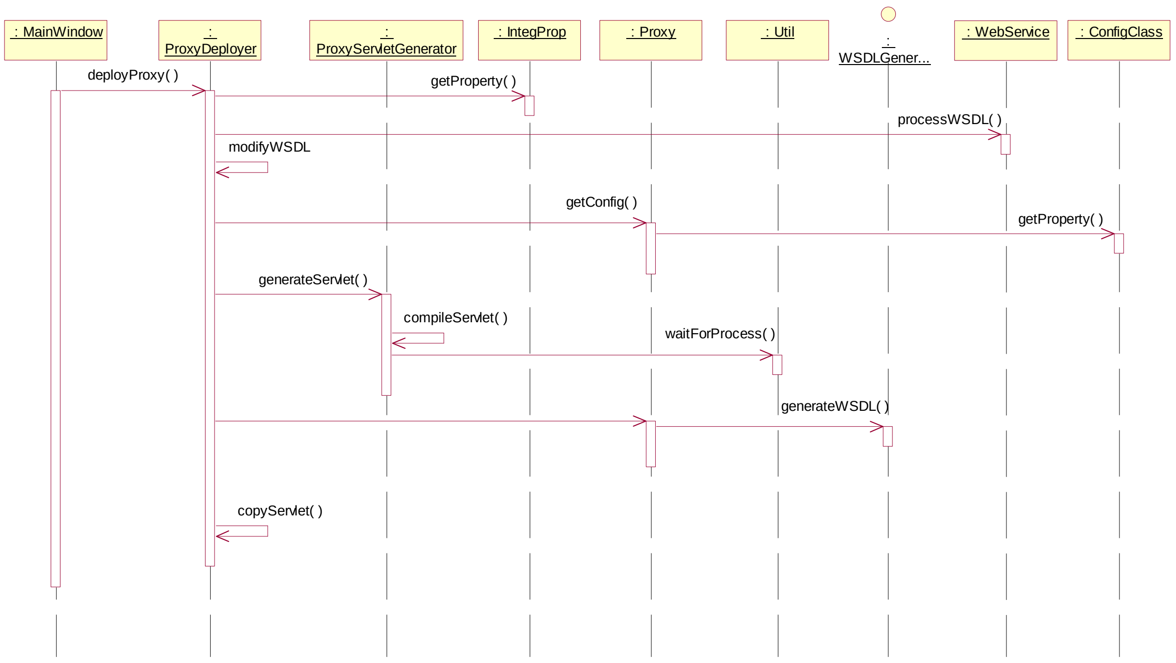


Figure 37 - Sequence diagram for use case 1.4 – second diagram.

The diagram shows the flow when generating and deploying the proxy servlets, which is done after the deployment of the web service application. The ProxyDeployer is responsible for deploying all servlets in one business model; this sequence diagram will therefore be repeated for each method selected for deployment. The ProxyDeployer will first do the things that are common to all the proxies that may be in a business model:

- Query the deployment tool's configuration, about where to deploy the proxy servlets.
- Query the web service platform for the WSDL file that describes the interface to the web service.

For each servlet proxy in the business model the ProxyDeployer will do the following:

- Modify the end point in the WSDL document to the address of the proxy. If the WSDL document contains other operations than the one used by the servlet, they must be removed.
- Get the configuration from the proxy. Notice that the proxy asks the ConfigClass for the configuration collected during the configuration process. This means that all proxies get the same configuration, even if some of the configuration values are not used in all proxies.
- The ProxyServletGenerator is set to generate and compile the servlet, based on the information collected. The process requires an external compiler to be called, to keep the application synchronized, the deployment tool is set to wait until the compiler ends.
- The WSDLGenerator implementation for the proxy is used to modify the WSDL document again. Because the generator is associated with the proxy, it has information about how to generated WSDL for exactly this proxy.

- The generated servlet and its WSDL document are copied to the servlet container.

Sequence diagram for use case 2

Use case 2 describes how to apply a new business model to the deployment tool. Because the process does not require directly interaction with the deployment tool, or other system components, a sequence diagram is not necessary.

Sequence diagram for use case 3

Use case 3 describes how to configure the deployment tool. Because the process is done using an external text editor, a sequence diagram is not necessary.

Design of business model implementations

The implementation of the interfaces described in the business model framework, varies a lot from business model to business model, and therefore requires individual design process.

Many of the business models described here require additional information to be attached to the SOAP message, used by the web service application. Because most specifications related to attaching security information to SOAP messages are incomplete or in draft release, the security information will be attached using parts of different specifications. The same problem exists with the WSDL document that must describe a service. If the security information changes the communication interface in a way that cannot be described in standard WSDL, the WSDL language must be extended. The best way to extend the standards is discussed for each business model.

Empty model

The purpose of the empty model is to verify that the system can deploy services, and communicate through a proxy. The ServiceClass implementation must therefore return the same messages as it receives in both the processRequest and processResponse method. The configuration of the model does not contain any configuration panels, therefore the configFlowController implementation must never return a panel. The ConfigClass implementation must be able to return an empty set of properties, when deploying the proxy servlet. The implementation of the WSDLGenerator must return the WSDL document passed to it, without modifying it. This can be done because the deployment tool has modified the service address and operations not used.

The XML document that describes the business model, must point to the correct implementations of all the interfaces in the business model package except the business model itself.

Signature model

The signature model must be able to verify signatures and sign messages. There are many approaches for implementing these two requirements, following considerations must be done:

- What must the client sign? Is it allowed for the client to sign more than one element of the message?

- Must the client send its certificate with the message, or can the server locate the certificate from a reference?
- What are the requirements to the client's certificate? Must the server allow self signed certificates?

The questions could be asked to the person that deploys an application using the model, this would require a rather complicated ServiceClass implementation and a complicated configuration flow when deploying. To keep the model relative simple, following choices has been made:

- The client must sign exactly one element of the SOAP message namely the body, which means that the content concerning the web service application must be signed. It is not allowed to sign sub elements of the body.
- The client must send a reference to the certificate, which must be present in the server's key storage.
- The certificate must be issued by a part trusted by both the server and client. The certificate will be checked by the server for revocation before use, but the issuer signature will not be checked. It is not necessary to check the signature of the issuer, because the certificate always comes from the server key storage. The process of putting certificates in the key storage must ensure that the signature is valid.

Flow diagram and user interface for configuration.

In the configuration of the model it must be possible to decide which file to use as key storage, what to do if a signature cannot be verified and whether the response should be signed. The configuration flow is therefore static and consists of three panels:

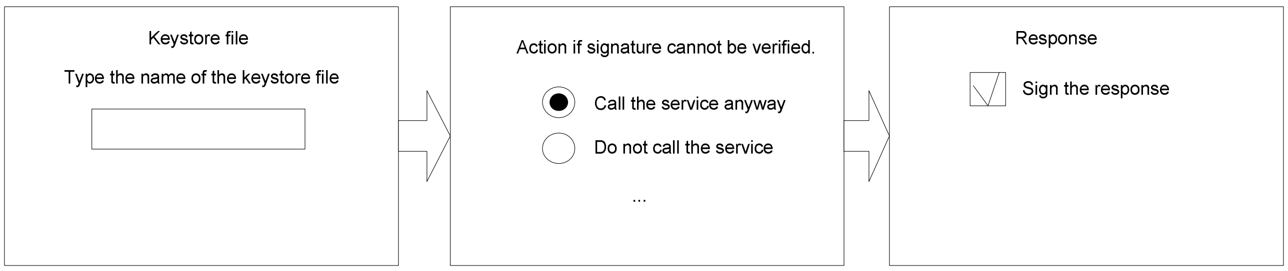


Figure 38

To keep the implementation of the ConfigClass in the business model simple, only standard configuration panels is used. The configuration flow could have done in less that three panels if the panels where combined (see design of configuration panels page 51).

Because all panels are static, they can all be defined in the configuration file for the business model, and the ConfigFlowController can run the flow by loading the panels in the same order as they are defined in the file.

Sequence diagram for request

The function used to sign and verify signatures, must be able to be reused by other business models and possibly also on the client, this functionality is therefore placed in another package than this business model.

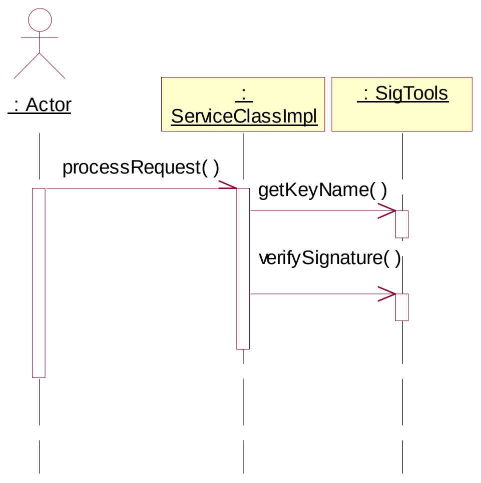


Figure 39

Sequence diagram for response

The request had a reference of which key to use to verify the signature, when signing the response, the business model must decide which private key to use for the signature. This could be done during the configuration of the model or in a property file; but for this test purpose the key name is hard coded in the business model. This makes the sequence diagram very simple, the service class simply calls a signBody in the SignatureTools class. The diagram is too simple show!

Design of interface and WSDL

The web service security specification version 1.0 defines a security element in the header of a SOAP message; this element can contain a digital signature in the format defined in the XML signature specification. A digital signature from the XML signature specification can contain information about how to retrieve the public key, which can be used to verify the signature. Figure 40 shows the expected content of a message coming from the client, the same structure will be used to send the signed message back to the client.

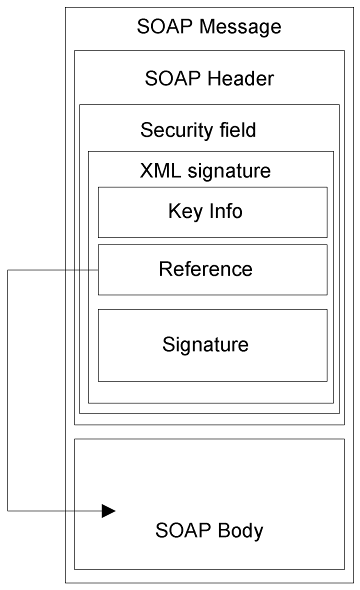


Figure 40

Notice that the structure of basic SOAP message is not changed, the only difference is that a security field is added to the header. The reference of what is signed must always point to the entire SOAP body.

The WSDL file to describe the interface to the servlet proxy, can be based on the WSDL document generated by the web service platform. When the address fields have been changed by the deployment tool (as described in design of the deployment tool), only a description of the security field must be added to the document, figure 40 shows that the content of the SOAP body does not change.

Carlisle Adams and Sharon Boeyen, Eurotrust have written a suggestion to some simple extensions to WSDL and UDDI that makes it possible to include security information. They suggest using a securityParameters element to contain security information. Because the binding element of the WSDL document describes the format of the message, it will be natural to place the securityParameters element in here.

The security element must contain following information:

- References to all operations that must be signed (one in this case)
- Information about how to sign (XML signature as described in above).
- Information about which signatures can be verified

Encryption model

The encryption model must be able to decrypt incoming messages, and encrypt outgoing messages. The key to use for both de- and en-crypt must be a symmetric key located in the server's key storage.

The model will assume that the entire body of the incoming SOAP message is encrypted, and it will encrypt the entire body on the outgoing messages. It must be possible to select a static key name to use with the model, or choose to let the client decide the name of the key. The possibility of letting the client decide the name makes the model easy to combine with the key agreement model described next.

Flow diagram and user interface for configuration.

The two first configuration panels are static, because the user must always decide which key storage to use, and if a static key must be used. Only if the user chooses to use a static key name the last panel must be showed.

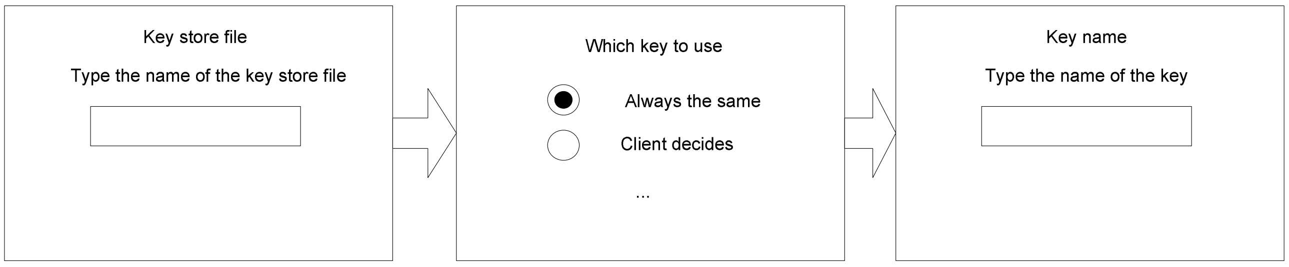


Figure 41

The ConfigFlowController must use the configuration from the ConfigClass to decide whether the last panel must be loaded. Like in the signature model, only standard panels have been used, to make the implementation of the ConfigClass simple.

Sequence diagram for request

The sequence diagram for a request in the model shows the scenario where the user has deployed a configuration that will use the key name send by the user.

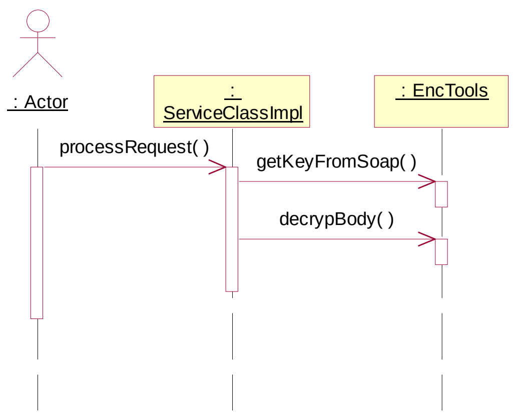


Figure 42

The actual functionality of the model is placed in a common class (EncTools), because the functions can be reused in other business models and by the client.

Sequence diagram for response

Like the request sequence, the sequence for the response relies on functions placed in the common class EncTools. Because the encryption must be able to use different encryption templates, e.g. if not the entire body should be encrypted, the template is loaded from a class responsible for the templates.

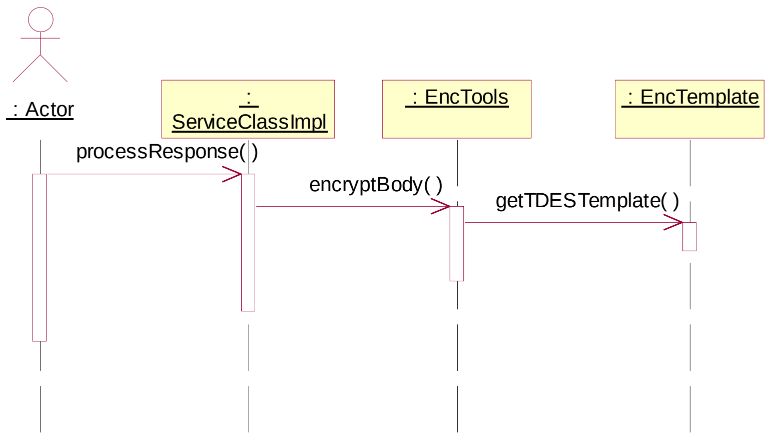


Figure 43

Design of interface and WSDL

The XML encryption standard specifies how to encrypt XML documents or parts of documents. The way it is done is by replacing the clear text node with an XML encryption node that holds the cipher text and information about it. One of the requirements to this business model is that the entire SOAP body is encrypted. The structure of an encrypted message send to or from the server will therefore have the structure showed in figure 44

Notice that the SOAP specification requires a body in a SOAP message, therefore the message is actually not a SOAP message before it is decrypted.

The key info is not used by the server if it is set to use a static value as key name. To keep the generation of the WSDL document simple, the WSDL will always require the key info to be present

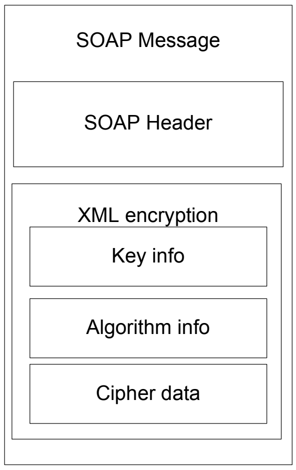


Figure 44

Like in the signature model the security information is natural to place in the binding element of the WSDL document, where also the encoding of the message is specified. Following information is required in the security field in the binding element:

- Link to the operations that need encryption. Because a servlet can handle maximum one operation in this system, the reference will always point to only one operation.
- Information about which algorithms that are supported.
- Information about how to attach the key reference.

Key agreement model

Key agreement can be done in different ways, depending on the facilities available on the communicating parts. If a number of secrets already have been exchanged, the key agreement can simply be to select which secret to use for the communication. It is however unusually that several secrets have been exchanged securely before the secure communication begins, therefore the agreement method is only used in systems where the number of sessions is relative limited. An example of a system that uses a variation of that type of key agreement is home banking systems, where the customer receives a paper with a number of one-time keys printed.

Key agreement can also be done if the parts have exchanged their public keys. The first part can generate a key and send it to the other part encrypted with the other parts public key. The other part can decryp the key using its private key. To avoid re-play attacks, the first part will have to include a unique serial number or similar, which the other part must be able to check, to see if it has been used before.

Diffie Hellmann key agreement algorithm is shortly described in the state of the art section. Compared to the other two key agreement mechanisms Diffie Hellmann has an advantage because both parts contribute with the generation of the shared secret. This means that a session cannot be replayed. As mentioned in the state of the art, the key agreement method does not ensure the identity of the communications parts, which makes systems based solely on secure agreement woundable if it is possible to spoof identity on the network. This weakness can however be overcome if the key agreement process is followed by an identity check

Diffie Hellmann is selected as key agreement method, because it is important to have a way to establish a encrypted channel between to parts that have not exchanged secrets or certificates.

Sequence of key agreement.

Client	Server
Generates a Diffie Hellmann key pair with no initial key. Sends the public part to the server	
	Generates a Diffie Hellmann keypair using the public key of the client as initial key. Generates a symmetric key from the client's public key and own private key. Sends own public key back to the client.
Generates the same symmetric key as the server from the server's public key and own private key.	

From the sequence it can be seen that the key agreement can be done in one RPC call, but to use the key, it will require more calls. Notice that the client call does not contain any data for the Web service application; therefore the servlet proxy can handle the request alone, and can avoid calling the web service layer.

User interface and flow during configuration

Only the key storage file name can be configured in this business model, the user interface for this is similar to the key storage dialogs used in the signature and encryption model.

Sequence diagram for request

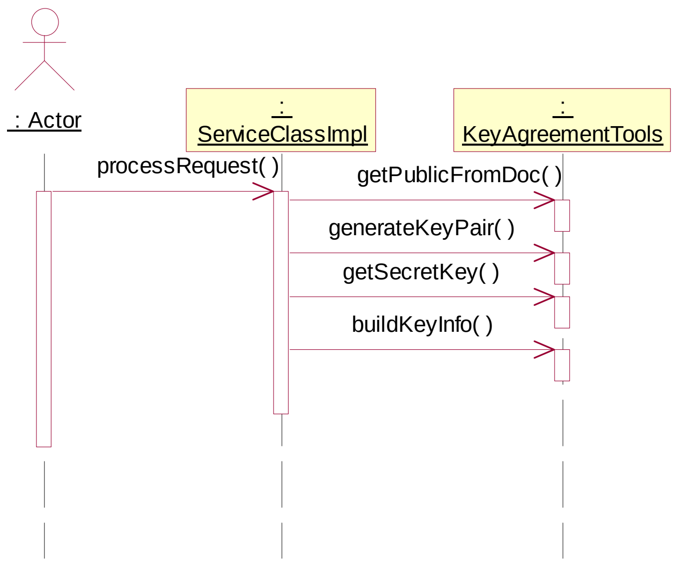


Figure 45

Sequence diagram for response.

The model does not send the request to a web service; the response is therefore the same as the request. All the processing will therefore be done while processing the request.

Design of interface and WSDL

As discussed only the public keys of the client and server will have to be send over the network, this means that there is actually no need for sending a SOAP message. A SOAP message is used anyway, to enable the model to process application data using the same message. The Public keys are therefore placed in the same security field in the SOAP header as the signature model uses. The structure of the messages is showed in figure 46

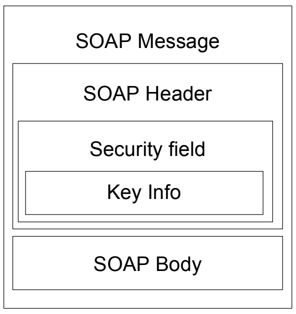


Figure 46

The WSDL document generated by the WSDLGenerator in the model, must add the same security field as used in the encryption and signature model, to the binding element. The security element must include following information:

- A key info description that shows how the model expects to receive the values of the public part of a Diffie Hellmann key pair.

The use of the SOAP header for exchanging the public keys is done because all other security related information is transported in the header. Another way of transporting the keys could be by using the SOAP body; the business model layer would then have to act like a standard web service operation, where both the input and output would be a DH public key.

Secure payment

The secure payment business model must be able to establish a secure connection based on the security mechanisms used in the other business models. The five security points described in the state of the art section can be used as checklist for the security in this model.

- Confidentiality: The encryption facilities from the encryption business model will be used to ensure the confidentiality.
- Integrity: The signature model can ensure the integrity by checking the signatures on signed data.
- Authentication: The signature model can easily be extended to check the client’s certificate against an issuer’s public key.
- Authorization – The business model must use the name from the client’s certificate to perform an authorization check. The database of the business model must store valid user names.
- Non-repudiation – The facilities in the key agreement business model can be used to make the same key available on the server and client. If the key is only used in one transaction, then the system is not vulnerable to replay attacks.

One transaction using the model will have the flow showed in figure 47.

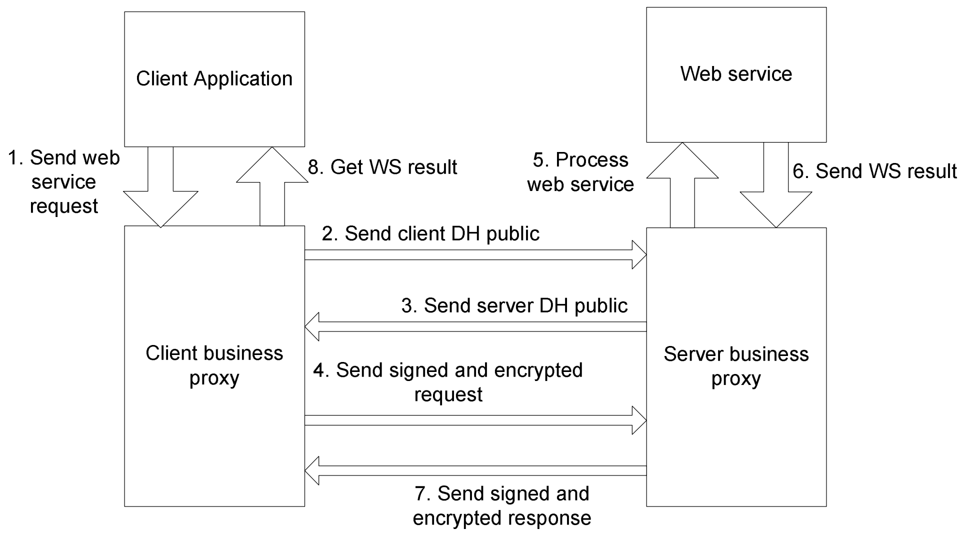


Figure 47

Notice that the client must send the web service request before the identity of the server is proven by its signature. The client can only assume that the network address is correct and it is actually the server receiving the encrypted message. This problem can be solved if the server signs the message with the public Diffie Hellmann key it sends back (3); the client can then stop the communication if the signature cannot be verified.

When the client prepares to send the web service request, it must first calculate the signature on the clear text message, and then encrypt the message. It is important to sign the message before it is encrypted, because the server will store the clear text version as proof for the transaction, and the signature must be used as a link between the client and this message. The client must attach its certificate with the web service request, because the server will use the public key of the certificate to check the signature of on the message. To ensure that the certificate can be trusted, the server must check the certificate with the public key of a third part that is trusted by both the server and client.

Flow and user interface for configuration.

In the configuration of the model it must be possible to select the key storage to use for the symmetric key, and the public key to verify the client's certificates. It must also be possible to type the name of the public key. The model must be able to send the name of the client to the application in on of the parameters used when calling the application. If the method to deploy does not take a String as one of its call parameter, the configuration must offer to pass the username. If the method is called with one or more Strings, the configuration must ask the user if one of them should contain the username from the client's certificate. Figure 48 shows the flow and user interface for the configuration.

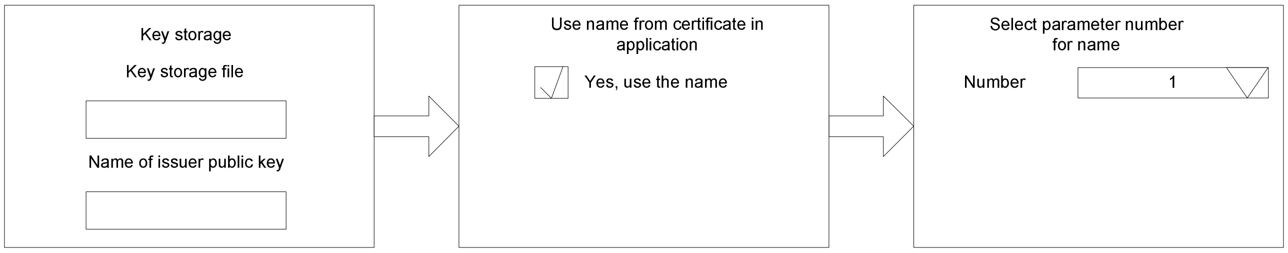

Key storage
Key storage file
Name of issuer public key
Use name from certificate in application
Yes, use the name
Select parameter number for name
Number
1

Figure 48

The second panel is only showed if the method takes a String parameter, and the third panel is only showed if the user chooses to use the name in the application.

To keep the business model simple, it is chosen to hard code many parameters that could be configured when deploying. Of things to hard code is:

- Database name and driver.
- Keystorage password and key passwords.
- Behavior of the model if users are revoked or signatures cannot be verified.

Sequence diagram for request

From figure 47 it can be seen that the business must have two servlet proxies; one for the key agreement and one for the web service call. Figure 49 shows the sequence diagram for the first servlet proxy.

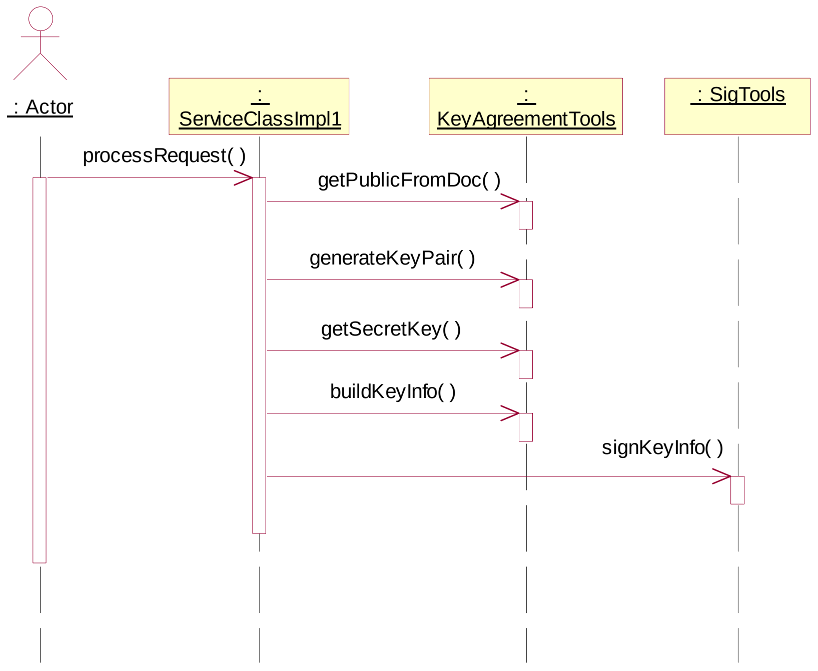


Figure 49

Notice that the key agreement functions are all used in the key agreement business model. The model places the public key information in the SOAP header, and the signature business can only sign the body of a SOAP message, therefore a new method that can sign a key info field must be added to the SigTools class.

The second proxy servlet relies on the symmetric key which the first put the server key storage. Figure 50 shows the sequence diagram for the second servlets request process.

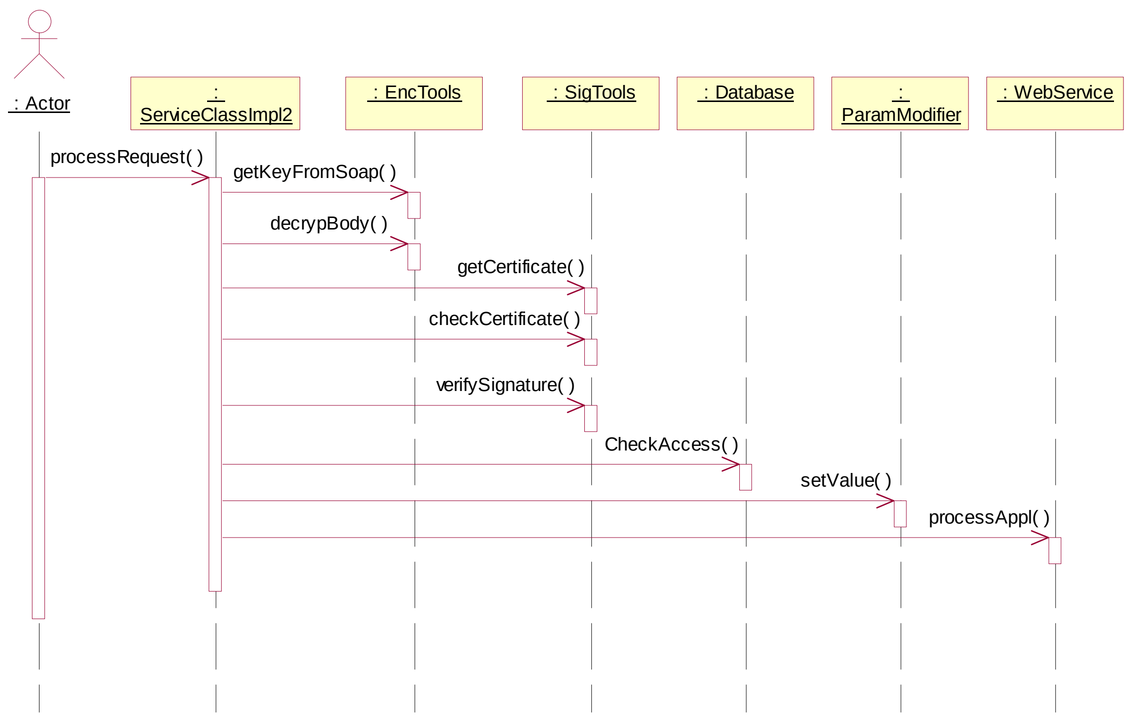


Figure 50

The EncTools class is used to get the name of the symmetric key used to encrypt from the message. The message is decrypted, and the SigTools class must check the validity of the certificate attached to the message. The SigTools also checks the signature on the SOAP body, to verify that the client has signed, and the integrity of the data is intact. The database class must check that the client has access to the web service

In the configuration it can be selected to let the business model overwrite on parameter passed to the web service with the name of the client. The paramModifier class can read and modify text values in the SOAP body. The class does not use any of the functionality from the web service platform; therefore can only Strings and Integers be read and modified. The client name is a String, and can be modified with the name form the certificate, if it is selected during the configuration of the model.

The ServiceClassImpl2 must save the request message until the response has been send to the client, because the signature and request must be saved in the database. The request is not saved in the request process, because the web service can fail, and the request stored in the database, will be used to charge the client for one processed request.

Sequence diagram for response.

The first proxy servlet does not call the web service application; therefore can the whole process be done in the request process.

Figure 51 shows the sequence diagram for a response in the second proxy servlet.

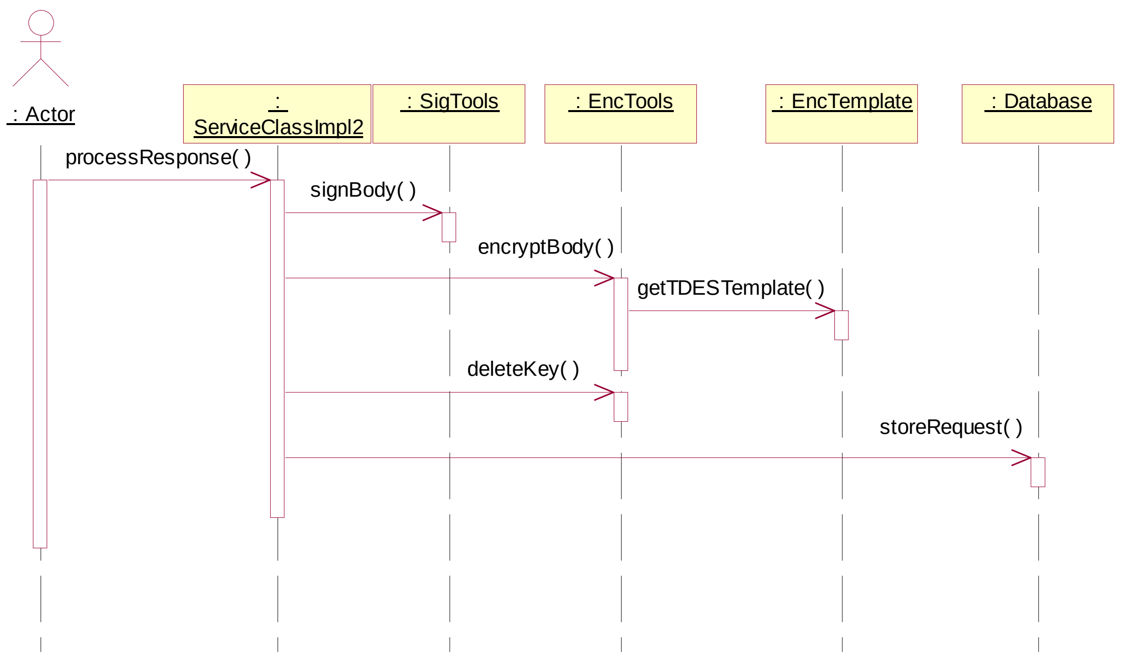


Figure 51

The response needs to be signed and encrypted like it is done in the signature and encrypt business model. The ServiceClassImpl2 must used the key name from the request to encrypt the message. It is important that the key is deleted from the key storage before the session ends; else it would be possible to replay the request. At last the request is stored in the database, to use as proof for a successful completion of the application call.

Database design.

The database design is relative simple if it is only used to information used in the business model. The user table only contains the name and a revoked field; it is assumed that the name is the unique identifier of a user. This is possible because the business model can only verify certificates from one issuer; this issuer must ensure that two certificates do not have the same name. The transaction table contains a reference to a user, and the entire SOAP message after it is decrypted. Figure 52 shows the database design.

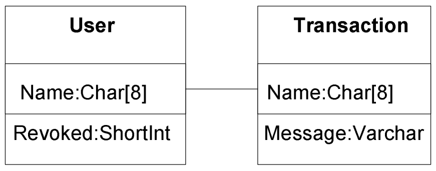


Figure 52

Design of interface and WSDL

As discussed the format of the messages used in the first proxy servlet, is almost similar to the messages used in the key agreement business model. The only difference is that the response from the proxy is signed. The response message will therefore have the structure showed in figure 53

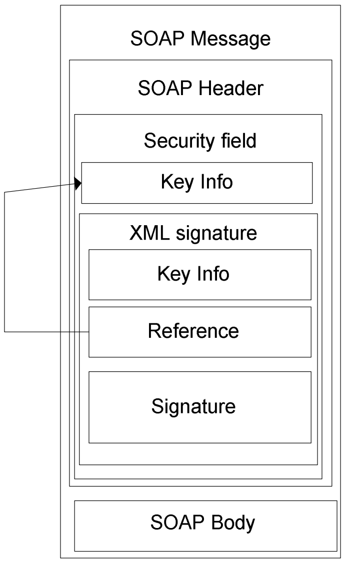


Figure 53

The WSDL document to describe this message must extend the document used in the key agreement model, by specifying that the response includes a signature in security field in the SOAP header.

The WSDL document does not need to specify that the signature is for the key info field. This information is contained in the signature itself.

The second proxy servlet will send and receive messages that are encrypted and signed, the structure of the messages will therefore be a combination of the messages from the signature and encryption model. Incoming and outgoing messages will have the same structure. Figure 54 shows the structure of the messages.

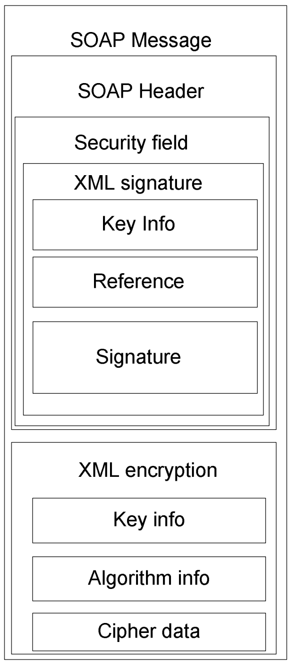


Figure 54

Notice that the reference in the XML signature cannot point to an element in the message, because the body element is changed to an XML encryption element. The receiver of the message must therefore know that the encrypted element must be decrypted, before the signature can be verified.

The WSDL document that described the message must contain both the elements defined in the signature model, and in the encryption model.

In the binding element of the WSDL document, both the encryption information and signing information must therefore point to the operation element. In this way will the client know that the SOAP body must be signed before it is encrypted, because the encryption process removes the body (the operation in WSDL), and makes the signing impossible.

Capability access

The philosophy of capability systems is that users with knowledge of the address and protocol used in a system should be granted access to the resource. In a standard Java web service system, all the deployed methods in a class will be described in a single WSDL document, and thereby be defined as one service   One way to make differentiated access based on capabilities would be to publish the same class one time for each capability access point.

Example: An application can search in a database using the method search(query_string1,query_string2, max_result), where the query strings is used to define what to search for, and max_result is the maximum number of results to return. Two capabilities must be defined for the method, one that gives directly access to the method, and one that will limit the use to one query string and a fixed number of results.

One way to implement this in a web service system, is to add another method in the application that calls the search method, but uses static values for the last two parameters. The application must now be deployed one time for each method, where only one method is allowed to be called in each web service.

This will result in two web services with different WSDL descriptions, if the application were deployed only once it would result in one WSDL description with both interfaces described. It is important to have two WSDL descriptions, because the access control is based on the communicating interface described in these files.

Because this system can deploy more servlets for one method, capability access can be implemented relative easy without adding new methods to the application. The business model to capability must be able to add default values to the application method, if required. Figure 55 shows the model of the capability system, with the method from above.

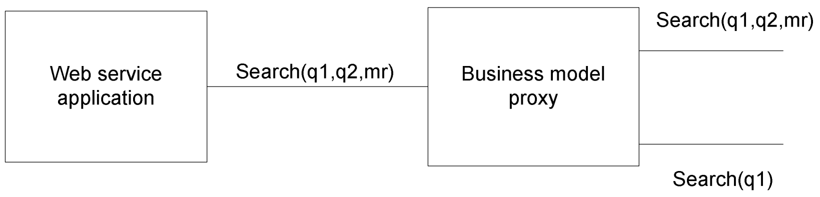


Figure 55

User interface and flow during configuration

In the configuration of the model, it must be possible to select which capabilities to deploy for one method. For each capability to deploy, the user must select which parameters to include in the interface specification, and which parameters the business model must assign default values. Figure 56 shows the flow and user interface of the configuration of the model.

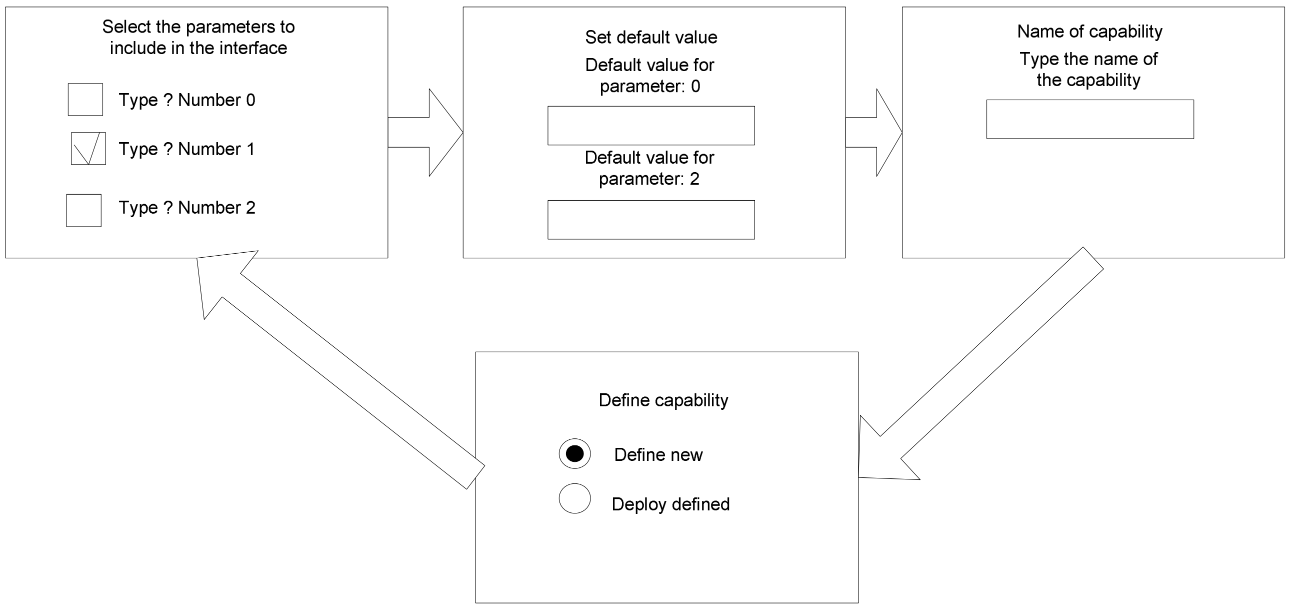

Select the parameters to include in the interface
Type ? Number 0
Type ? Number 1
Type ? Number 2
Set default value
Default value for parameter: 0
Default value for parameter: 2
Name of capability
Type the name of the capability
Define capability
Define new
Deploy defined

Figure 56

The first panel must be build dynamically by the ConfigFlowController, because the choice boxes must match the parameters used in the method to deploy. The second panel must be build dynamically using the choice made in the first panel and the parameters required by the method. The last two panels can be defined in the static description of the model, because they do not change with the selections made.

The last panel must be able to end the dynamic configuration process, if the user selects to deploy the defined capabilities.

Sequence diagram for request

The sequence performed on an incoming request to the web service application, depends on the configuration of the servlet. Figure 57 shows the sequence diagram when request parameters needs to be added by the model.

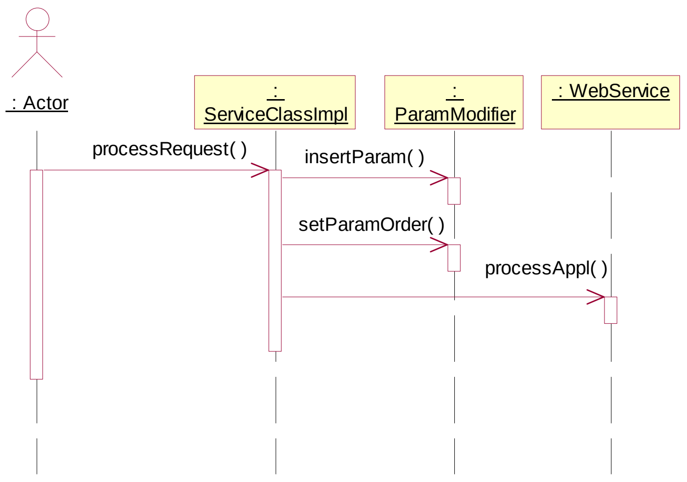


Figure 57

The parameterAdder method must be able to in insert parameters in the SOAP request, but it must also be able to change the order of incoming parameters. In this way it will be possible for the model to construct SOAP messages that fits the interface of the web service independent of the structure of the incoming message.

Sequence for response

The capability access is done alone in the request, the response must therefore be send to the client without modifications.

Design of interface and WSDL

The interface of each deployed proxy servlets, depends on the selection made in the configuration, but it will always be a sub set of the original method. The WSDLGenerator must therefore be able to remove parameters in the WSDL message element. The elements to remove must be selected using the selections made during the configuration of the model.

CPU payment

The CPU payment business model will ask the client to do some calculation before it gives access to the web service. The model must therefore be able to send a problem to the client, and first call the web service when the client delivers an answer. A model of the communication is showed in figure 58

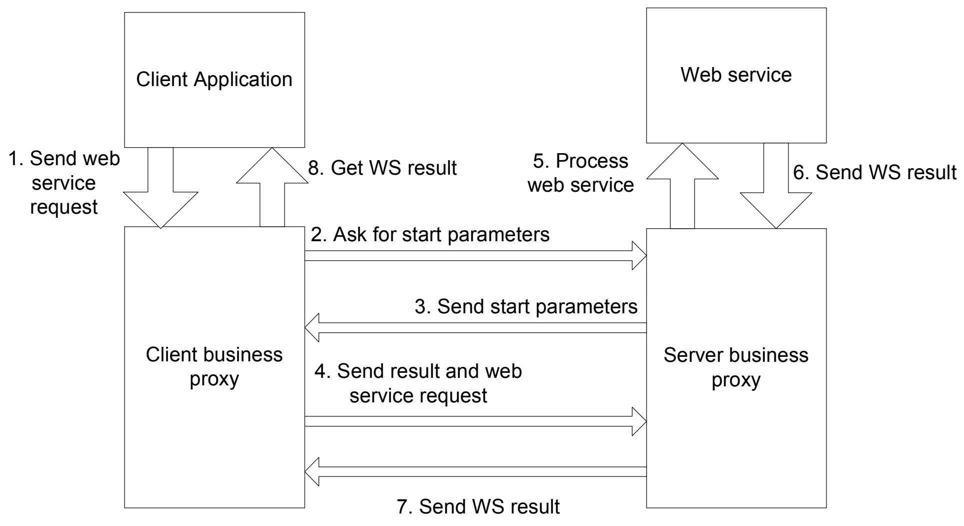


Figure 58

The figure shows that the business model must consist of two proxy servlets for each web service operation: One that can deliver start parameters, and one that can receive results and process the clients request in the web service layer. The model can easily be extended by a check of the result the client sends; this is especially useful if the check is requires less calculation than the operation done by the client.

User interface and flow during configuration.

The simple implementation of the model requires only knowledge about where to find the start values for the client, and where to store the result. This can be done by two text input fields in one static configuration panel, as described in the design of the configuration panels.

Sequence for request

The diagram on figure 59 shows both the calls that are necessary to make to the service. Notice that only the second call communicates with the web service application. The database implementation is showed as one class, but the actual implementation may require more classes, depending on how to communicate with the database.

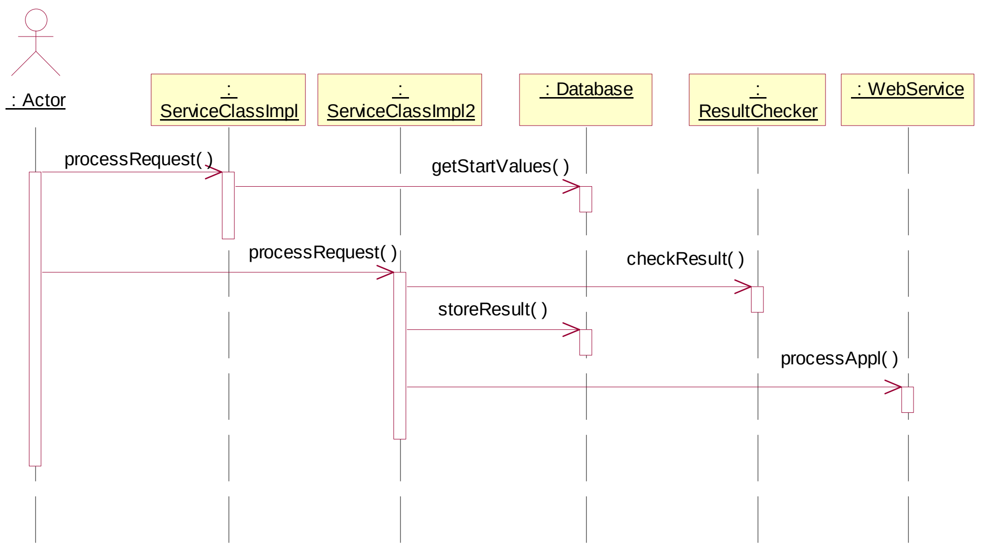


Figure 59

Sequence for response

Because the first proxy servlet of the model do not call the web service, there is no need for processing the response.

The second proxy servlet does call the web service, but the client does not need more information form the business model layer, so there is also no need for processing this response.

Design of interface and WSDL

The interface to the business model is divided into two parts, because it consists of two proxy servlets. The message send to the first servlet can be empty, because the servlet does not require any information from the client. The response must however contain a start value to the client, and it must be described in the WSDL how to interpret the value. The description of the value depends on which problem the client is asked to solve, therefore a new namespace may be required for each problem to solve. Following information must be available to the client in the description of the service.

- Data format and encoding of the data returned from the service.
- Reference to an application or algorithm that can use the data as input.

The second servlet proxy must be able to receive the result from the client in the same message as the clients request to the web service application. Transport of the result can be done in the same way as security information: In the header of the SOAP message.

The WSDL document for the second servlet proxy must describe the data format and encoding of the result value, in the same way as the first described the format of data it returns.

Package structure

The package structure of the system is already partly described in the class diagrams; an important thing to notice is that the helper classes used in the business models, must be placed in a separate package. The separation of the functions in the helper classes is important if the components must be reused, but also for separating the web service knowledge from the business implementation. Following packages is used in the system:

Businessconfig:	The configuration panels available to the dynamic configuration of the business models
Businessmodels:	The business model framework
Businessmodels.dh	The key agreement model
Businessmodels.dsig	The signature model
Businessmodels.empty	The empty model
Businessmodels.encryp	The encryption model
Businessmodels.secpay	The secure payment model
Secintegrator	The functionality of the deployment tool
Secintegertor.gui	The gui classes of the deployment tool
Security.xml	The common functionality classes for the business models

Client considerations

The clients are not considered as a part of the system, but the design of the system should make it easy to develop clients capable of communicating with the proxy servlets. Further it is necessary to develop test clients to ensure that the business models works and fulfills the requirements.

As discussed the server uses a web service platform to do the marshalling and de-marshalling, the business model layer can therefore concentrate on the security and payment related tasks. The same model can be used on the client, and thereby obtain the same benefits of using the web service platform.

The client will also have to run a proxy that handles the business model related tasks, but not necessarily on an application server. The web service platform will usually provide an API that will do the marshalling and send directly to the network. The best solution for the client business model proxy, would be if the proxy could replace the network layer of the web service API. This would however require modification of the web service platform. Instead the proxy will be developed as a network proxy.

For testing purpose the client must be able to change the business model relative easy, therefore the client proxy is developed in much the same way as the server proxy, with a pluggable service layer. The client model will look like figure 60

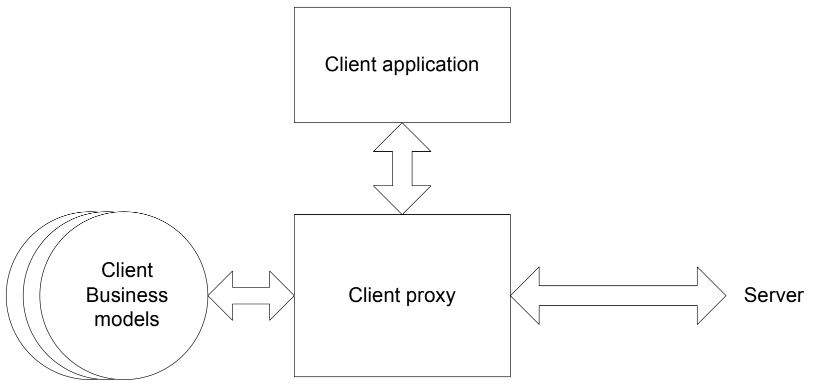


Figure 60

Notice that the proxy is able to have more service layers installed, this is used when a business model requires more than one RPC call. The service layeres must have full control over the messages like it is the case on the server proxy. This makes it possible for the business model send exactly what is required by the receiving servlet, and not necessarily the SOAP message generated by the client application.

WSDL to Java application considerations

If the web service platform is used on the client side as described in the client considerations, it will be natural if the WSDL to Java application that is usually available on the web service platform, can be used to help generate the client application. The generating application will have to be extended to be able to generate the right service class to use in the client proxy layer.

One of the challenges of the code generator is to make clients that use more that one web service to one client RPC call. The information of the relation between two web services is not described in the WSDL of the services. An example of this is the secure payment model where the last servlet requires that the first servlet is called to set up the transport key. The WSDL of the second servlet only describes that the message must be encrypted, and that the client must send the name of the key under which the message is encrypted. To enable a code generator to generate proxy code for coupled web services, a language to describe the relations is therefore necessary.

Test applications

To verify that the proxy, deployment tool and business models, the test environment needs to be set up properly. The applications described here will help in testing the system components.

Key storage generator

Some of the business models rely on the availability of some cryptographic keys and certificates. It is out of the scope of the project to develop applications to securely distribute these components, instead one application capable of setting up key storages for the client and server is developed. How the key storages is made available to the server and client is not considered. Content of the server key storage:

- One private RSA key belonging to the server. This key is used to sign messages in the signature and secure payment model.
- One certificate containing the public key of a certificate issuer, signed by the issuer. The public key is used to verify certificates send by the client

- One certificate containing the public key of the server, signed by the issuer. The certificate is attached in the signature and secure payment model. (But only used by the client in the secure payment model.)
- One certificate containing the public key of the client, signed by the issuer. This certificate is only used in the signature model, to verify the signature. In the secure payment model the certificate send by the client is used.
- One symmetric key. To use in the encryption model. The secure payment model will use the symmetric key generated in the model.

The content of the client key store is similar, except the private RSA key that must belong to the client.

Calculator

The calculator must be able to add and subtract two integers and return the integer result. The application can be used for testing the deployment tool and business models.

Text application

To test the capability of changing the user name in the SOAP message in the secure payment model, the web service application must take a string parameter as argument. The text application must take a text as parameter and return the same text plus some extra text that indicates that the text has been processed in the web service layer.

Implementation and test

This section describes how the components designed in previous section can be implemented. The implementation is not explained in detail, only if the implementation requires more than basic programming knowledge, it is explained. For a more detailed description of the implementation, refer to the Java documentation or the source code on the CD-ROM (see appendix a).

The design of the system shows that the system must rely on external software components. The system must be implemented in Java, and the most obvious choice is the standard edition in the newest release which currently is 1.4. Choosing the newest release means that the system will not run on systems where only older versions are installed. This is a major drawback especially for the web components, because updating the Java virtual machine on an application server usually changes the environment for all applications running in its context. Updating the JVM on an application server is therefore not as easy as for a standalone application, and requiring the latest version of Java may conflict with the existing configuration. The web components must therefore be able to run on an older version of Java than the standalone applications. Java standard edition version 1.2 or newer can be found on most application servers, and is therefore selected as platform for the proxy and business models. The proxy must be able to run as servlets on an application server, therefore it relies on the Java enterprise edition. Version 1.3 of the J2EE is chosen as J2EE platform, because 1.4 is currently in beta release, and 1.3 is available on most application servers.

For the deployment tool Java standard edition version 1.4 is selected as platform, mainly due to the support of regular expressions, which is not standard in earlier versions.

The choice of using Java version 1.2 on the application server means that all XML processing APIs must be downloaded separately and included with the web applications. The same is the case for the cryptographic support required in some business models.

The system is designed to work with different web service platforms and application servers, in the implementation and test Apache AXIS will be used as web service platform, and Apache Tomcat as application server. The choice is made because they can be downloaded for free, and the source code is included, if unexpected problems should occur. AXIS includes a small proxy that can echo the messages send through, this is used for debugging

As development application IBM WebSphere Application Studio Developer is used. The application offers Java development with instant debugging facilities on different Java platforms and application servers. Development of XML documents and different web-related files is also possible. Concurrent Versioning System (CVS) is used as code repository for the development process.

Order of implementation

In the design section it is mentioned that not all the designed components will be fully implemented. The reason for this is that the purpose for the project is not to implement a fully functional system, but to focus on some of the issues of developing secure web services. The order of the implementation is therefore focused on implementing the main functionality, and not on making the system stabile or user-friendly.

Due to the size of the project, the two last business models (Capability access and CPU payment) will not be implemented.

Business model proxy

The priority order of implementing the proxy is as follows.

1. Core proxy functionality as an http servlet on an application server.
2. Analyze the messages that go through the proxy. Build XML documents from the messages.
3. Create instances of the serviceclass from different business models, and process the messages in the instances.
4. Reply on WSDL requests.
5. Create a reusable API with functions used in different service classes.
6. Error handling that will inform the client of errors occurred by sending SOAP fault messages.
7. Extended error handling with local logging.

The success criteria for the proxy is that 1,2 and 3 are implemented and functional.

First success criteria:

To implement point one, a test environment consisting of an application server must be installed and configured. This application server must be set up in debug mode to enable step-wise debugging of the server applications.

The application server can handle http request, therefore it is not necessary to code a specific http layer to handle the mime headers. On the connection to the web service platform, the proxy can use the same mime headers as on the incoming connection. The incoming data stream can be copied directly to the web service connection's output stream, using a byte array as buffer.

Second success criteria:

The incoming data stream must be feed to an XML document builder. The IBM XML security package contains a document builder that can build DOM representations of XML documents from a data stream. It is possible to use other API's that can build SOAP messages from XML data streams, but some of the business models require the messages as DOM representations, so this format is selected from the start.

Third success criteria:

The proxy servlets can have the business model classes in the class path, but must be able to make instances of the classes from a text description. When doing this the instance must be cast to a ServiceClass implementation. The classpath must also include all the classes used by the ServiceClass.

Fourth success criteria:

The reply on WSDL requests will be handled by the servlets, and is therefore described in the implementationof the servlet template in the deployment tool.

Fifth success criteria:

The reusable API will be implemented when implementing the business models.

The remaining success criteria will be considered "nice to have", and will not be discussed nor implemented!

Test cases

The test cases for the proxy are defined by its use cases

For all tests performed the empty model will be used to as business model and the calculator application as web service. The calculator client will be used without business model proxy attached, to generate the test requests and show the returned result.

In the test of handling WSDL requests, an Internet browser will be used.

Deployment tool

The priority order for implementing the deployment tool.

1. Deployment of a Java application on a web service platform.
2. Generation of a servlet with a changeable configuration. Deployment of servlet on an application server.
3. Analyze of a compiled java class.
4. Use of external business models in the application.
5. GUI to collect all information required to generate a servlet and deploy the application.
6. WSDL modification.
7. Storing system configuration in a XML file.
8. Error handling to display errors in a user-friendly way.
9. Extended GUI that can navigate both forward and backware in the deployment process and show help.

The success criteria for the deployment tool is that 1 to 7 are implemented and functional.

First success criteria:

It is essential for the system to be able to deploy applications on the web service platform (AXIS). The deployment tool must be able to use the existing AXIS deployment tool, which can use XML files as deployment descriptors. The files must contain information about the service name, application name and which methods to publish. The system must therefore be able to write such XML files, and call the external application using this file.

As discussed in the design the application must wait for the external application to end, before the execution can continue. This can be done by polling the external process for it’s return value, and continue when the value is returned by the process. The polling is done every 0.5 second, and the application is set to sleep between the polling. It is important control the polling interval, because if the application is not set to sleep between the polling it will use unnecessary CPU time that could be used by the external process.

Second success criteria:

To generate servlets with a variable configuration, it is necessary to have a template containing keywords to use when adding the configuration. Regular expressions from Java 1.4, is used to find the keywords in the template, and inserting the configuration parameters. Java properties with only Strings as keys and values, is used as type for the configuration, which means that string values is inserted in the Java source template.

This can result in an error if the user uses escape characters in the configuration values. Example: The user must type where a key storage is placed and types:

```
C:\keystore.ks
```

Which will result in the following code is added to the servlet:

```
Properties p= new Properties();
p.put(”keystore”,”c:\keystore.ks”);
```

The code will not be able to compile because the correct notation for the backslash is '\\' when writing Java source code. The problem can be omitted by simply parse the strings before they are used in the source code, or base 64 encode the string before it is used in the source. Example of servlet source if an encoded value is inserted:

```
Properties p= new Properties();
p.put(”keystore”,new String(Base64.decode(”Dhas1dV76Zbb9hilG”)));
```

The compilation will also succeed, if the user of the application remembers to type \\ instead of \. In the first version of the program nothing will be done to avoid the problem in the application.

The template must contain the code that uses the proxy (and thereby the business models), but it must also be able to process requests for WSDL files. To tell the servlet that it must read the WSDL file instead of calling the proxy, the user must type the address of the servlet followed by ?WSDL in a browser. The application server will pass the "WSDL" as an attached query string. The servlet can check for this query string every time it is called, and thereby know when to read the WSDL file associated with the servlet.

The location of the WSDL file is decided by the deployment tool, but even if the file is placed in the same directory as the servlet, it is not necessarily the default directory for the servlet. The default directory is decided by the application server, and can therefore not be used. Instead the WSDL directory is passed to the servlets using an environment variable called WSDL_HOME.

The compile procedure must be done with the correct classpath, which can be controlled, because the compiler is called as an external process, which the deployment tool can give parameters. The command string to execute is therefore build from the parameters in the configuration file and the parameters collected from the user. Example of a simple compile command:

```
”c:\java\Javac.exe –cp c:\lib” ”c:\source\servlet.java”
```

The placement of the java file is decided by the source generator, and must be a place where the application server can find the servlet.

Some application servers requires that the servlets are contained in a web project, and that the servlet is included in the XML description file for the web procject. The information required in the XML file is much the same as required for the AXIS web service deployment tool. Because not all application servers requires this description file, the deployment tool cannot modify such files in the first version.

Third success criteria:

The deployment tool must be able to analyze classes located outside the tools class path, which requires a way to analyze classes from byte read from the disk. To determinate the package for the class, the byte loader must use both the directory where the package begins, and the path to the class. The loader must subtract the base directory, and thereby build the package name using the java.lang.ClassLoader. The analysis of the class once it is loaded must be done using java.lang.Class and java.lang.reflect.Method.

Fourth success criteria:

GUI support for business model configuration is implemented as described in the design by having one target panel, and changing the panels that is showed in the panel. The deployment tool must simply use the interface of the ConfigFlowController, to get the panels to display, when the controller returns ‘null’ as the next panel, the deployment tool must continue to deployment phase. The GUI of the configuration flow must be implemented using Java AWT.

Fifth success criteria:

The visual design of the static GUI makes it possible to used some of the components made available to the dynamic configuration of the business models. Only the first panel that contains a “Browse” button con not be implemented using the standard components form the business model configuration panels.

Sixth success criteria:

The deployment tool must ask the web service platform to get the WSDL document from the application, so the business model can make its model specific modifications. In AXIS the web service must be called with the query string WSDL to get the document, this must be done by using a URL connection.

The WSDL document must be modified before it is passed to the business models WSDL generator, this must be done using XPATH expressions for finding the correct elements to modify or remove.

Seventh success criteria:

To store the configuration of the deployment tool in an XML file makes it easy to organize the properties logically; by it requires a way to refer the properties in an easy way from the application. The configuration class will convert text string into an XML path, and thereby be able to look up values in an XML file. The class must use ‘.’ As element separator, and assume that the last token in a string is an attribute name. Example of part of a configuration file:

```
<deploymenttool>
   <deployment>
      <proxy port=”8088”/>
   </deployment>
</deploymenttool
```

To get the proxy port value from the application the following identity string must be used in the application:

```
deploymenttool.deployment.proxy.port
```

The standard java Class java.util.Properties could have been used, but it would require the configuration file to be in standard Properties format, which is harder to keep organized.

Eight success criteria:

The error handling in the deployment tool can help the business model developer to debug the business models, therefore must the error handling be detailed and include a program location of where the error occurred. The error must be displayed in a modal dialog box that will exit the application when closed.

Ninth success criteria:

To activate the back button in the static configuration flow is relative easy, it simply requires that the states will be saved on a stack. The back button must set the be able to change the state to the previous state. To enable the back button in the dynamic configuration flow requires that the ConfigFlowController is able to return the old panels, which is not supported by the interface. In the use case for the dynamic flow it is however stated that if the back button is activated during the dynamic configuration, the application must go to the beginning of the dynamic flow, which is possible using a stack for the static panels.

Test cases

The test cases for the deployment tool are given in the use cases for the application.

The variations described in the use cases will not be implemented, but the application still conforms to the first seven success criteria, because the variations occurs on errors or pressing the back- button. Screen dumps of the application running can be seen in appendix e

Empty model

The priority order for implementing the empty model.

1. Manual deployment of a working (but transparent) model
2. Interaction with the deployment tool.
3. ClientProxyLayer implementation.

The success criteria is that the first two success criteria are implemented and functional

First success criteria:

The manual deployment requires a functional business model proxy, with a servlet capable of accepting connections and telling the business model proxy to make instances of the empty model implementation of ServiceClass. This criteria is fulfilled when testing the proxy.

Second success criteria:

The implementations of ConfigClass ConfigFlowController and WSDLGenerator must be usable in the deployment tool. The model can also be used as test for static configuration panels defined in the business model description file. The implementations can also be used when testing the flow and functionality in the deployment tool.

Third success criteria:

The ClientProxyLayer that will make the client business model proxy work as a transparent proxy, is not necessary when testing the business models on the server. If a business model on the server must be tested with a empty client proxy, it will be easier to use the client without a proxy. The empty ClientProxyLayer can however be useful when testing the client business model proxy.

Test cases

Test 1: Requesting through the ServiceClass

Pre Req: The ServiceClass implementation of the business model is made accessible for the business model proxy on the server. A standard proxy servlet that uses the model and points to a valid web service is developed and deployed.

Test: A client capable of using the web service directly is requesting the web service through the proxy servlet.

Expected result: The client receives the same result as when requesting directly to the web service.

Test 2: Deploying an application using the model

Pre. Req.: The business model description and Classes is made accessible to the deployment tool.

Test: The deployment process is done as described in the design of the deployment tool, using the empty model.

Expected result: The static flow of the deployment tool is completed, and the application is available as web service in the same way as described in the Pre. Req. for test 1.

The Empty model have passed both tests.

Signature model

The priority order for implementing the signature model.

1. Signing and verifying of signatures in one standalone application. Source: XML document from a file.
2. Signing and verifying on both client and server. Implemented as a ServiceClass on the server and a ClientProxyLayer on the client. Source: SOAP message generated by the AXIS client API.
3. Implementation of ConfigClass, ConfigFlowController and WSDLGenerator, to configure the model and generate WSDL files.
4. Functions to reuse in other models moved to common class.
5. Error handling if the signatures cannot be verified.

The success criteria is that 1 to 4 is implemented and functional.

First success criteria:

To enable the necessary cryptography in older Java Runtime Environments, JCE and a crypto provider must be downloaded and placed in classpath. JSSE contains providers that can do the necessary cryptography for signing and verification of signatures. The default providers are listed in the java.security file, can also be loaded dynamically during the execution of the application. When the providers are shipped with the application (as in this case), the best way is to remove all providers, and add exactly the needed providers. A lot time can be saved by doing this, because some providers generate runtime errors, while others works fine with the same code.

IBM XML Security suite is used to generate and verify the signatures. In the reference to the signed object it is important to use an XPATH expression to point to the SOAP body, because this ensures that the receiver will find the same element, even if other elements have changed in the document.

Second success criteria:

When dividing the application into a client and a server, two different JVM will be used it must be ensured that the necessary crypto providers are present in both classpath'. If the crypto providers were not loaded dynamically, both java.security files would possible have to be updated, which is problematic for some application servers.

Third success criteria:

The configuration flow of the model is static, which will make the implementation of the ConfigClass and ConfigFlowController relative easy. The WSDL generator must always add a static value to the binding element of the WSDL generated by the web service platform. As discussed in the design section the SecurityParameters tag must be used to describe the security information. In this field the XML signature element can contain information about the algorithms and references to the element that needs to be signed. The signature can contain certificate information e.g. when using the <X509Data> tag, this can be used to give information about which certificate issuer's that are accepted.

Because the proxy servlet represents only one method in the web service application, the reference will always point to the one operation element in the WSDL document. The whole security element can therefore be static for all servlets using this model, and can be hard coded in the WSDLGenerator class.

See appendix ??? for an example of a WSDL file generated by the model.

Fourth success criteria:

The function of signing and verify a signature of a SOAP body can be used in other models, but also the capability of adding an element to the binding element of a WSDL file can be reused.

Fifth success criteria:

The best way of reporting an error is by generating a SOAP fault and send this to the client. The logging facility of the application server can also be used to store the information on the server.

Test cases

Test 1: Deployment

Pre. Req. The deployment tool is correctly configured and the application server is started. The signature model is made available to the deployment tool.

The test: The deployment process is run, the calculator class is selected as application and signature model as business model for one of the methods.

Expected result: The calculator application is deployed on the web service platform. A proxy servlet using the signature model is deployed, pointing to the calculator web service.

Test 2: Requesting through the ServiceClass

Pre req. Test 1 is completed successfully. Two key storeages are prepared using the CertMaker application.

The test: A request is send to the proxy servlet using the calculator client and the client proxy with the signature model.

Expected result: The client will receive the result as if it was communicating directly with the web service.

Test 3: Modifying a signature reference

Pre.req. Test 2 is completed successfully. The server code is changed so one byte is changed in the SOAP body of the incoming message.

The test: A request is send to the proxy servlet using the calculator client and the client proxy with the signature model.

Expected result: The signature element can be verified, but the first (and only) reference will fail to verify.

The Empty model have passed the tests.

Encryption model

The priority order for implementing the encryption model

1. Encryption and decryption in one standalone application using a symmetric key. Source: XML document from a file.
2. Encryption and decryption both on client and server implemented in a ServiceClass and a ClientProxyLayer. Source SOAP message from the AXIS client API.
3. Implementation of ConfigClass, ConfigFlowController, to configure the model. And WSDLGenerator to generate WSDL documents for the model.
4. Functions to reuse in other models moved to common class.
5. Error handling if the decryption fails.

The success criteria is that 1 to 4 is implemented and functional

First success criteria:

Strong symmetric encryption is not supported by the default crypto providers in JCE, BouncyCastle must therefore be used to generate triple DES keys and to do the en-decryption. IBM XML security suite contains the API to apply the encryption on XML data.

Second success criteria:

Like in the signature model, the challenge using a client –server model is to ensure that the crypto providers are available in both Java Virtual Machines. Further the encrypted element must be transported in a SOAP message, but the encryption breaks the SOAP structure, so the message must be handled as DOM.

Third success criteria:

The configuration flow depends on the selection made in the second panel; the third panel must only be loaded if the right selection is made. The ConfigFlowController must therefore use the properties from the other panels to decide whether to load the last panel.

Like the signature model, the WSDL document of this model must use the SecurityParameters tag to the security information. The EncryptedData tag from XML Encryption namespace can be used to describe the format of the encrypted data. The element must contain an EncryptionMethod element, a KeyInfo element and a reference element. Like in the signature model the security field is the same for all servlets using the model, therefore can the element be hard coded in the WSDLGenerator class.

See appendix f for an example of a WSDL document generated by the model.

Fourh success criteria:

The function of encrypting and decrypting a SOAP body can be used in other models, and must be moved to a common class. Also the hard coded SecurityParameter element can be moved, for easier reuse.

Fifth success criteria:

As described in the signature model.

Test cases

Test 1: Deployment

Pre. Req. The deployment tool is correctly configured and the application server is started. The encryption model is made available to the deployment tool.

The test: The deployment process is run, the calculator class is selected as application and encryption model as business model for one of the methods.

Expected result: The calculator application is deployed on the web service platform. A proxy servlet using the encryption model is deployed, pointing to the calculator web service.

Test 2: Requesting through the ServiceClass

Pre req. Test 1 is completed successfully. Two key storeages are prepared using the CertMaker application.

The test: A request is send to the proxy servlet using the calculator client and the client proxy with the signature model.

Expected result: The client will receive the result as if it was communicating directly with the web service.

Key agreement model

The priority order for implementing the key agreement model

1. Implementation of key agreement in one standalone application.
2. Implementation of key agreement in a ServiceClass on the server and in a ClientProxyLayer on the client.
3. Implementation of ConfigClass, ConfigFlowController, to configure the model. And WSDLGenerator to generate WSDL documents for the model.
4. Functions to reuse in other models moved to common class.
5. Error handling if something goes wrong.

First success criteria:

Implementing Diffie Hellmann in one application is relative easy, because the transport and encoding of the public keys is not necessary. Diffie Hellmann key agreement is supported by SUN's JCE provider, and is therefore available if JCE is installed in the JVM. The secret key generated in the process is a triple DES key and this requires JCE 1.2 or later to successfully store the key.

Second success criteria:

IBM XML security suite does not support Diffie Hellmann key agreement; the exchange of the public keys generated in JCE is relative easy though. The DHKeyValue from the XML encryption namespace can be used as element for the transport, after the key is base 64 encoded.

Third success criteria:

The implementation of the configuration process is not really interesting for this model, the web service application to deploy will never be called, because the model is build to ignore the application. The configuration of the model is however important if the functionality is used within other business models; the configuration flow is therefore implemented even though its usability is limited.

The WSDL document is more interesting, because it must describes a web service that has no operation, but will use a SOAP message to transport a public key. The way it is done, is by removing all operations from the binding element, and add a SecurityParameter element containing a KeyInfo element that describes the valid key format.

See appendix f for an example of a WSDL document generated by the model.

Fourth success criteria:

The key agreement functions, and the XML formatting of Diffie Hellmann public keys must be moved to a common class to reuse by other business models.

Fifth success criteria:

As in the other models.

Test cases

Test 1: Deployment

Pre. Req. The deployment tool is correctly configured and the application server is started. The keyagreement model is made available to the deployment tool.

The test: The deployment process is run; the calculator class is selected as application and key agreement model as business model for one of the methods.

Expected result: The calculator application is deployed on the web service platform. A proxy servlet using the encryption model is deployed, pointing to the key agreement web service.

Test 2: Requesting through the ServiceClass

Pre req. Test 1 is completed successfully. The ServiceClass and the client proxy are temporary modified to echo the agreed key value.

The test: A request is send to the proxy servlet using the calculator client and the client proxy with the key agreement model.

Expected result: The client and server proxies will echo the same key value. The client application will terminate with an error, because the model will not call tha servlet.

Test 1 fails because the configuration of the model is not completely implemented. After manual deployment the second test is completed without errors.

Secure payment model

The priority order for implementing the secure payment model.

1. Combination of Key agreement, signature and encryption model, both on server and client
2. Implementation of ConfigClass and ConfigFlowController, to configure the model.
3. Modifying parameters in the SOAP body.
4. Access lookup in database and storing requests in database.
5. Implementation of WSDLGenerator class to generate the WSDL file for the model
6. Error handling in proxy servlet.

First success criteria:

Combining the business requires two proxy servlets and two ClientProxyLayers, as described in the design of the model. This will test the ability to deploy more than one proxy servlet for one web service application in the deployment tool. The ability of expanding on RPC call in the client proxy will also be tested.

Second success criteria:

The configuration flow of the model depends on which types the selected method takes as parameters, the ConfigFlowController must therefore use its knowledge of the method to create the configuration flow. When analyzing methods in a compiled class using the java.lang.reflect.Method it is not possible to get the name of the parameter, therefore will the panel used to select a parameter be based on the index of the parameter.

Third success criteria:

The model must be able to modify a string parameter in the soap body. XML is in text format, which makes it possible for the business model to modify the parameter directly without using an external marshaller. The correct parameter is found using XPath, which is relative simple, because the SOAP message will always contain a call to the same application method.

Apache SOAP API could have been used to modify the message relative easy, but the business model requires the SOAP message to be in DOM format e.g. when encrypting. Transformations between Apache SOAP API and DOM have proven to break the signature, because it removes text nodes that contain only spaces.

Fourth success criteria:

IBM DB2 personal edition is used as relational database system to store the information used in the business model. The database is created using the configuration application included with DB2. Access from the business model is gained by using JDBC, which is the Java frame work for accessing databases. The query in the database is relative simple, but must use prepared statements, to avoid conflicts if the query parameters contain SQL reserved characters.

The size of the request to store in the database can vary, because it contains parameters decided by the client application, the database must therefore be able to store data of variable size. One way to obtain this is by selecting a fixed max size for the column in the database, and pad the request to fit the column. The database can also be set to store data of variable size, but this reduces the performance of the database. For testing purpose the performance is not an issue, and the database is therefore set to store data of variable length.

Fifth success criteria:

The WSDL document for the first proxy servlet, is the same as the one used in the key exchange model except that the model signs the public key send back to the client. One new element must therefore be added to the binding element of the WSDL document, namely a Signature element pointing to the KeyInfo field.

See appendix f for an example of a WSDL document generated by the model.

Sixth success criteria:

Error handling must be done in the same way as in the other business models.

Test cases

Test 1: Deployment

Pre. Req. The deployment tool is correctly configured and the application server is started. The secpay model is made available to the deployment tool.

The test: The deployment process is run; the text application class is selected as application and secpay model as business model for one of the methods. In the configuration it is selected to let the model modify the calling parameter to the owner of the certificate.

Expected result: The calculator application is deployed on the web service platform. A proxy servlet using the secpay model is deployed, pointing to the calculator web service.

Test 2: Requesting through the ServiceClass

Pre req. Test 1 is completed successfully. Two key storeages are prepared using the CertMaker application. A suitable database for the model must be available, and the client must be created as user in the database.

The test: A request is send to the proxy servlet using the text application client and the client proxy with the secpay model. The client must call the web serive using a string parameter different from the name in its certificate.

Expected result: The client will receive a result string, where the name from its certificate is included. The database will contain the decrypted request from the client.

Test 3: Variations of test 1 and 2

Test 1 is run with another source application, to verify configuration of the parameter modification will change.

Test 2 is run with different settings on the client and server to verify that the model will reject the client based on its certificate validation, authority in database and message integrity.

Capability access model

The capability access model will not be implemented and therefore not discussed here.

CPU payment model

The CPU payment model will not be implemented and therefore not discussed here.

Client proxy

The priority order for implementing

1. A transparent standalone proxy, capable of forwarding http requests.
2. Building of XML documents from content.
3. Processing of XML documents in more ClientProxyLayers, enabling expanding of one RPC call from the application to more RPC calls to web services.
4. Integration with client application.

5. Error handling

First success criteria:

The server proxy makes use of the application server for network connections; one thing the application server does is that it reads the http headers, and makes the content of the request available as an input stream. To do this, the server makes use of the HttpServletRequest, which is available from the Java 2 Enterprise Edition, and therefore cannot be used in the client proxy. The proxy must rely on plain Sockets, and do the separation of headers and data itself. The headers are separated by \r\n, and the data stream begins after the first empty header.

The proxy must be able to establish a new connection to the server and send the same mime headers, because it will not make changes to the content of the data. The data content must simply be read from the socket input stream, and send directly to the socket connected to the server.

Second success criteria:

One of the mime headers contain information about the length of the content data being send. This information is used to make sure that the entire message is read, before trying to build an XML document. The value of this mime header must be changed before it is send to the server, because the length can have changed during the transformation from byte data to XML document and back. Because the client proxy will do the same cryptographic functions as on the server, the SOAP messages will be handled as DOM objects.

Third success criteria:

To expand the RPC call from the client application, the proxy must implement a chain of ClientProxyLayers. They must be used in the same order as they are added to the proxy. The first layer will process the message from the client application and the response from the server, and pass the result to the next layer. In this way can information from all server RPC call be accumulated before the client receives the result.

Fourth success criteria:

The proxy must act as part of the client and should not run as a standalone application. Because it listens for connections on a network port, it will have to run in a separate thread when it is integrated in the client application. The client must initialize the proxy with the correct ClientProxyLayers and start the proxy, in the initialization of the client.

Fifth success criteria:

The proxy must be able to recognize messages containing SOAP faults. If a the server sends a SOAP fault the proxy must send it directly to the server without processing it in the business model layer. If an error occurs on the client side, it must generate a SOAP fault and send it to the client application.

Test

The purpose of the client is to be test tool for the business models on the server, and will not be tested further than when used in the test of the server.

Evaluation

The purpose of this project was to obtain knowledge form the web service domain in general, and use this knowledge to develop a system capable of deploying Java applications as web services, and apply a security model.

In the state of the art section, many of the technologies used in the web service domain were described. It is clear that the domain is very large, and exactly where it starts and stops is very difficult to define. The section described both specifications and Java tools that support the specifications; not all specifications and tool were described, but the section shows that there exist many specifications and tools. To fully understand the domain therefore requires a lot of study, and because the technology moves relative fast, it is hard to be up to date with all work being done within the domain.

The work being done in securing web services is very focused on trust across domains, which is natural because one of the sales points for web services is the ability of business to business communication. The suggested systems which should help in establishing this trust are often complicated, and the description in the state of the art section is very short, compared to the complexity and the focus many companies have in this area. The reason for this is that the system developed in this project had to be less complicated than the solutions described in e.g. XKSM and SAML, due to the time limit of the project.

The use of discovery service with web services is possible with the technology today, but the focus is still on enabling applications to speak web services. This is also the case when looking at security in web services, the focus is on the messages send between the communicating parts, not on how to describe the communication in WSDL or securing the discovery service.

The section with requirements to a web service payment platform, described why there is a need for a payment system platform. The requirements were based on existing payments systems, and assumptions to future payment systems made in the state of the art section. In the considerations were also the technology available, especially what web services offers, and the influence many people think web services will have on the future way of doing business.

The selection on business models to implement was made based on existing ways of doing business on the internet today, and assumptions to the future payment models.

The purpose of the design section was to design a system that conforms to the requirements stated. The selection of a proxy model was made mainly because it do not require any modifications to existing systems, but also because it is independent of the web service platform used. One of the drawbacks of the proxy is that it is not capable of serialize and de-serialize complex objects, which makes it hard to work with the content of the messages in the proxy. A solution to the problem could be to use the web service platform to get and set parameter values in the proxy; this will however make the proxy dependent of the web service platform. The proxy mainly works with the security header of the messages, where the complex objects are known by the business model; serialization and de-serialization of these objects can therefore be done without using the web service platform. The problem only occurs when the proxy needs to access the data meant for the web service application.

The proxy with a changeable business model fulfills the requirement of separating the business model from the service application; because it gives the business model full access to the messages that is send to and from the application. To develop a business model requires some knowledge of web services, the requirement of separating web service knowledge from the payment knowledge is therefore not fulfilled completely. Because the commonly used functions related to security and SOAP messages, is moved to an API package available to all business models, the required knowledge about web services is little when developing new business models. Only if a business model must do something that is not included in the API, the business model developer will have to know something about web services.

The model of having at least one servlet for each method in the web service application, was selected, because the proxy will not have to read the destination method in the incoming messages to select the right business model. In the capability access business model, it also turned out to be usable when defining different capabilities for the same method. As described, the drawback is that if more business models are applied to different methods in the web service application, it is possible to use a wrong business model to call the web service. The solution is to let servlets check the name of the method in the incoming messages.

The challenged of making easy deployment of new application in the system, required a new deployment tool. The tool is designed so users with very little knowledge of web services can deploy applications and combine them with business models. The cost of the easy deployment is less flexibility. Some flexibility is however possible with the tool, because the developer of the business modes decides what needs to be configured in the proxy servlet, the deployment tool can furthermore be configured using the configuration file. The deployment process is kept simple, because it is possible to deploy applications manual, if flexibility is preferred instead of the fast procedure.

The business model framework is designed to make the development of business models easy and yet flexible. Most of the framework is used only in the configuration and deployment process; this shows that the price of the easy deployment using the deployment tool is relative high. The use of interfaces in the framework is essential, the different business models implemented using the interfaces, shows that the framework can be used in many of today’s business models, and possible also in future models.

In the design of the business model implementations cryptography is widely used, the standards for using security information in XML documents are mature and API’s exists to easily implement the standards. The suggestions on how to use security information in web services are incomplete, and typically not standardized. Most suggestions are however based on XML security, so the security design in the business models is based on the XML security specifications. Attaching security information to SOAP messages using XML security specifications is relative easy, but the description of the security in WSDL files is harder.

The web service policy description suggests a way to attach security information to WSDL files, the way it is implemented in the business models is similar to the suggestions, yet not completely as described in the suggestion. The description of an empty SOAP body with a signed public key in the header, can not be described in WSDL with the web service policy extensions.

The need for extensions to the WSDL specification is also clear in the CPU payment model. The model does not use cryptography, by it is required that the client has solved a calculation problem to get access to a web service. WSDL can describe the interface of both the proxy servlets in the model, but the connection between the two services cannot be described.

The client and client proxy are not considered as components in the system, the reason for this is that the system should make it possible for everyone to develop clients capable of communicate with the system. The client proxy is developed for testing the system, but is also designed to be pluggable, so a future WSDL to Java tool will be able to generate the required plug-ins to support a business model.

The development and test environment is essential, the time spent configuring test and development system compared to the time spent designing and coding the system is relative high. The reason for this is that many components from different vendors have to work together, and when a new release of a component must be added, it will typically affect many parts of the system. Because web services is a fast evolving area, the frequency of updates to the used components is relative high.

Future of the system

The system developed fulfills most of the listed requirements, but the system needs to be extended and adjusted to have real value in a production environment. Especially the generation of WSDL documents needs improvement; the concept of using the WSDL generated by the web service platform is useable if only minor modifications needs to be done. If the business model communicates via a completely different interface, like it is the case for the secure payment model, it is simpler to let the business model do the whole generation.

The developed system is functional, and indicated that web services indeed can be used in payment systems. It shows that it is possible to separate the knowledge of applications, web services and security, and combined it when deploying the web service. The system does not address the need for trust across domains or the need for security when communicating with an UDDI registry. It is clear that this is necessary if the visions of web service shall come true, but until the security specifications are standardized and general accepted, implementing such systems will be infeasible.

Bibliography

Books:

Deitel - Java Web Services for experienced programmers

Keogh – J2EE complete reference.

Stelting Maassen – Applied Java Patterns

Serge Abiteboul – Data on the Web

William Stallings - Cryptography and network security

Blake Dournaee – XML security

Donald E. Eastlake III – Secure XML

Robin Sharp – Principles of protocol design.

Douglas R.Stinson – Cryptography theory and practice.

Tutorials:

Eric Armstrong and others - The Java Web services tutorial

Internet ressources

http://java.sun.com

http://www.ibm.com/developer

http//www.ibm.com/alphaworks

http//www.w3c.org

http://www.oasis.org

Appendixes