Extending WebSphere Applications for Business-to-Business Integration

AS/400 Edition

July 31, 2000

An IBM e-business Experience Report

For questions, contact:
ebit@us.ibm.com

Preface . . . 5
Introduction . . . 6
Business solution . . . 6
Executive summary . . . 8
Key findings . . . 9
Scenario architecture . . . 11
Environment overview . . . 11
Run-time environment . . . 11
System hardware . . . 13
Key software products . . . 14
About the WebSphere Application Server . . . 14
About the WebSphere Payment Manager . . . 14
About Lightweight Directory Access Protocol (LDAP) . . . 15
About AS/400 virtual private networking (VPN) . . . 15
About eXtensible Markup Language (XML) . . . 15
About XML Lightweight Extractor (XLE) . . . 16
About LotusXSL (eXtensible Stylesheet Language) . . . 16
Application overview . . . 16
Application design decisions . . . 18
INSCO application . . . 19
WebSphere security using LDAP . . . 24
Database . . . 27
Underwriter and INSCO B2B applications . . . 28
eXtensible Markup Language (XML) . . . 29
XML Lightweight Extractor (XLE) . . . 30
Virtual private networking (VPN) and IP packet security . . . 30
INSCO application . . . 32
INSCO server setup . . . 32
Setting up the WebSphere Application Server . . . 32
Installing WebSphere Application Server . . . 32
About WebSphere Application Server security . . . 33
Configuring WebSphere Application Server security . . . 35
WebSphere security discoveries . . . 35
Setting up the LDAP server . . . 36
LDAP discoveries . . . 38
Setting up the Payment Manager . . . 39
Installing Payment Manager Framework and CyberCash cassette . . . 39
Creating a Payment Manager instance and adding the cassette . . . 40
Configuring and administering Payment Manager . . . 40
Payment Manager sources . . . 42
Payment transactions . . . 42
INSCO application details . . . 43
InscoLoginServlet . . . 44
Insco2Servlet . . . 44
PaymentServlet . . . 44

JavaBeans component . . . 45
INSCO application flow . . . 45
Login . . . 46
Home . . . 47
Update personal information . . . 49
View policy information . . . 50
Submit payment information . . . 52
Select customer information . . . 53
Create new customer . . . 56
Submit insurance application . . . 57
INSCO servlet development discoveries . . . 59
B2B applications . . . 62
B2B server setup . . . 62
Setting up e-mail . . . 62
B2B application details . . . 64
XML communication . . . 64
INSCO B2B application . . . 65
Trigger program . . . 65
Trigger program discoveries . . . 66
SubmitApplication . . . 66
XLE mapping files . . . 67
UnderwriterResponseServlet . . . 68
Underwriter B2B application . . . 69
SubmitApplicationServlet . . . 69
ManualApproveServlet . . . 70
XSL style sheet . . . 70
Network and security . . . 72
INSCO security requirements . . . 72
Security technologies . . . 73
INSCO security design . . . 74
Other security design considerations . . . 75
The scenario network . . . 76
Configuring IP packet filtering and NAT . . . 77
Setting up the VPN . . . 78
INSCO VPN configuration . . . 79
Underwriter VPN configuration . . . 79
Appendix . . . 80
Reference information . . . 80
System hardware . . . 80
System software requirements . . . 81
Application source code . . . 82
Application screen views . . . 82
Database details . . . 97
Data files . . . 98
Examples: LDIF import files . . . 103

Examples: XML coded messages . . . 106
Payment Manager 2.1 API . . . 108
Setting up AS/400 WebSphere applications for B2B integration . . . 110
Configuring WebSphere Application Server security . . . 110
Configuring HTTP Server for authorization services . . . 111
Securing web resources . . . 111
Configuring the network . . . 113
Configuring network security . . . 115
VPN creation steps (INSCO) . . . 115
VPN configuration details (INSCO) . . . 121
VPN IP filter rules (INSCO) . . . 122
VPN creation steps (underwriter) . . . 122
VPN configuration details (underwriter) . . . 128
VPN IP filter rules (underwriter) . . . 129
Trademarks . . . 130
License and disclaimer . . . 131

Preface

This scenario experience report is brought to you by a group of AS/400® developers in the IBM® Rochester laboratory, called the e-business integration test (e-bit) team. The scenarios, designed and constructed by the team, are derived from established business patterns (for example, user-to-business, business-to-business (B2B)). These scenarios demonstrate how AS/400 technology can be used to build e-business solutions.

The team takes the scenario through the design, implementation, evaluation, and deployment process much like an actual company would. Although the team is restricted to a test laboratory environment, great effort is made to reflect reality. Upon completion of the scenario, the team publishes an experience report, like this one, which documents the team's scenario experience.

Who should read this report?

This report provides content that is useful to a broad set of people who are involved in e-business solution implementations:

- **System architect** who designs a new web-based application that needs to incorporate existing applications, data, and technology. Recommended sections: Introduction, Scenario architecture
- **Solution provider** who configures a complete e-business application for a customer. Recommended sections: All sections of the document
- **Other I/T architects** who need to understand the technology and products used for e-business application development. Recommended sections: Introduction, Scenario architecture
- **Technical specialist** who needs detailed information on how to configure an application server or needs source code examples. Recommended sections: Introduction, INSCO application, B2B application, Network and security

Introduction

In the first implementation, we depicted an existing insurance company that was interested in providing customer self-service via the web. The insurance company was able to provide a web site where its customers could use a browser to obtain insurance policy information and update personal information. The first implementation report, titled “Scaling Up e-business Implementations with WebSphere: AS/400 Edition,” is available on the web at:

http://www-4.ibm.com/software/ebusiness/scalingwebsphere.html

For this scenario, we are extending the implementation of a fictitious insurance company called INSCO. The insurance company was satisfied with its first experience in using the web to improve its relationship with its customers. Customers are now satisfied because they can review their policy information anytime they want. Agents do not have to handle as many inquiry calls from their customers, which allows them the time to support even more customers.

The next step for the insurance company is to continue to exploit the Internet as a new way of doing business. This will eliminate many of the inefficiencies associated with traditional insurance processes and transactions. The goal is for the insurance company to extend its customer service capabilities for external customers and to provide service capabilities for the insurance agents. In addition, it would like to tie all its partners and systems together to make running its operations more efficient and cost effective. The insurance company will extend its e-business implementation by providing the following services:

Policy access and online payments. In addition to being able to view policy information and update personal data, customers will now be able to pay their semiannual insurance premiums over the web using a credit card and a secure electronic payment method. Agents will now have web access to policy information for the customers that they support. Account administrators will be able to view and update information for all customers. With the addition of service capabilities for agents, the goal is to continue to improve customer satisfaction while at the same time reducing personnel costs to support these customers.

Online application submission. Agents will now be able to submit new insurance applications over the web using a simple browser interface. Once the application is submitted and accepted, a request is sent to an insurance underwriter for final approval. The goal is to improve customer service while increasing the efficiency and reducing the costs of submitting a new application.

Business solution

This report describes the experience of enhancing an advanced e-business Customer Relationship Management (CRM) solution. This solution implements a user-to-business and a business-to-business (B2B) scenario. User-to-business and business-to-business are two of the

business patterns that IBM customers and partners are focusing on. These business patterns and scenarios are a key component of the Application Framework for e-business, enabling IBM and business partner technical consultants to match proven architectures and designs to a given business problem. The business patterns constitute a set of reusable best practices for building e-business applications today. This report attempts to capture some of these best practices in an AS/400 environment. For more information on the different business patterns, see the Patterns for e-business web site located at:

http://www.ibm.com/software/developer/web/patterns

This interactive patterns site acts as a guide to aid you in the selection of the pattern and topologies most relevant to your needs. It describes a system architecture and design structure for various classes of applications. The patterns are categorized as follows:

User-to-business. Users interacting with enterprise applications and transactions
User-to-online buying. Businesses selling packaged goods through an online catalog
Business-to-business. Businesses programmatically linking between businesses
User-to-data. Users extracting useful information from large amounts of data
User-to-user. Users collaborating with each other using e-mail and shared documents

At the time of this writing, the web site has material for the user-to-business and user-to-online buying patterns.

The solution described in this report is an implementation of the user-to-business pattern. It deals with users interacting with an enterprise insurance application. The user-to-business pattern is simple in concept. It has a presentation layer, represented by a browser communicating with an HTTP Server, and an application layer, which is responsible for the business logic and data access. This is an example of the thin client model in that only a small portion of the presentation logic runs on the user's browser. Most of the presentation logic and all of the application logic are on the server. This model allows for easy scalability by increasing the power of the server machine or the number of servers used to handle requests. It is also easy to adapt this model to new kinds of clients such as personal digital assistants (PDAs) and cell phones. The actual business logic never has to change.

For more information on the user-to-business pattern, see the following redbook:
Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition, SG24-5864, at:

http://www.redbooks.ibm.com/abstracts/sg245864.html

This solution also represents an implementation of the B2B pattern. It deals with a programmatic link between the insurance company and an underwriter. The B2B pattern is still being finalized, but is fairly well defined. It consists of two application layers on different business machines communicating with each other using mutually agreed-to messages. Each message represents a unit of work that must be performed at the target system. The message format and transport

mechanism must be predetermined and agreed to by both sides. The message format should be made as general as possible so as not to expose the internals of either application. This allows the implementation of the application to change over time without affecting the message definitions. The Patterns for e-business web site that is referenced above will be updated with B2B pattern information in the near future.

The solution described in this report uses the concepts and terminology described in the Patterns for e-business web site. It may be useful to refer to that web site for additional background information.

Executive summary

This scenario enhances the existing self-service web insurance operations. The goal was to provide a web site where the insurance company's customers and agents could use a browser to obtain insurance policy information, update their personal information, submit payments, and submit applications. This scenario allowed us the opportunity to describe the following procedures for AS/400 customers:

- Use the original INSCO environment, which included a firewall and Network Dispatcher, as the environment for this scenario.
- Set up and use the IBM HTTP Server for AS/400 licensed program.
- Set up and secure a virtual private network.
- Establish and implement a security policy and use various techniques to ensure the privacy of transactions: Secure Sockets Layer (SSL), basic authentication, WebSphere™ security for login processing, which is backed by a Lightweight Directory Access Protocol (LDAP) directory, which is in turn backed by a validation list for secure storage of passwords.
- Set up and use WebSphere Application Server 3.02 Advanced Edition for AS/400 licensed program offering (LPO). (WebSphere Application Server 3.02 Standard Edition for AS/400 PRPQ could also be used.) The web-facing application in this scenario implemented a combination of Java™ servlets and JavaServer Pages™ (JSP) technology.
- Use the WebSphere Connection Manager function to improve performance when accessing the database.
- Set up and use the IBM WebSphere Payment Manager for AS/400 licensed program with the CyberCash cassette to accept payments over the web.
- Use eXtensible Markup Language (XML) to format messages to the remote underwriters using standard document type definitions (DTDs) defined by the insurance industry.
- Use XML Lightweight Extractor (XLE) to convert policy information stored in the database to XML format. This approach will be used in lieu of DB2 XML Extenders for AS/400 that will be available at a later time.
- Use LotusXSL (eXtensible Stylesheet Language) for converting XML documents into HTML to view policy applications through a browser.
- Use LDAP to provide agent authorization to policy records.

- Access existing enterprise data residing in a DB2® Universal Database™ for AS/400 (DB2 UDB for AS/400) from a Java servlet.

The intent of our scenario implementation was to extend how the AS/400 could be used to implement an e-business solution. We found that we were able to extend this e-business scenario on the AS/400 and take advantage of AS/400 strengths such as reliability, scalability, security, and cost of ownership. Not only did we use the AS/400 for the core transactional web servers, but we also used the AS/400e™ server as the enterprise database, LDAP, and Payment Manager servers.

Key findings

The following list provides a glimpse into some of the key findings for this scenario:

- The AS/400 provided the server functionality we needed for our base environment. This included the key servers, IBM HTTP Server, WebSphere Application Server, LDAP server, and Payment Manager server. The approach we chose was to get the servers up and running with minimal configurations and then tune the configuration per our specific implementation. For more information on the setup and configuration, refer to the **“**INSCO application**”** section for more details.

- For the main INSCO implementation, we made use of WebSphere Application Server security to secure our servlets and JSP files. We also used the LDAP server to provide a way to grant agents access to specific customer policies. We also used the Payment Manager server to provide a means for secure payment of policies. For more information, refer to the “INSCO application” section for more details.

- To secure our web applications, we used WebSphere Application Server Version 3.02 security. When considering using the security function provided, take into consideration that your application will run slower and that the function is complex to configure. For more information, refer to the “INSCO application” section for more details.

- For the B2B implementation, we established a secure connection between the two businesses’ internal systems by establishing a VPN. Communication between the B2B applications involved the use of eXtensible Markup Language (XML). One application used XML Lightweight Extractor (XLE) to convert its database information into an XML message. The other used LotusXSL (eXtensible Stylesheet Language) to convert the request XML document into HTML. For more information, refer to the “B2B application” and “Network and Security” sections for more details.

- During the implementation of this scenario, we experienced several key debugging challenges. We were faced with debugging issues concerning LDAP, WebSphere Application Server, DB2 UDB for AS/400, and WebSphere Payment Manager. In the sections where these technologies are discussed, there are discovery sections that will give

you helpful hints. For more information, refer to the “INSCO application” and the “B2B application” sections for more details.

Scenario architecture

In this scenario, INSCO will continue to utilize the logical three-tier model of IBM's Application Framework for e-business. More information on Application Framework for e-business is available on the web at:

http://www-4.ibm.com/software/ebusiness/

For more information on the three-tier model used in the first scenario, refer to the first implementation report titled "Scaling Up e-business Implementations with WebSphere: AS/400 Edition." It is available on the web at:

http://www-4.ibm.com/software/ebusiness/scalingwebsphere

INSCO enhanced its business by integrating its internal system with a business partner system. This allowed INSCO to run its operation more efficiently and cost effectively. INSCO implemented a secure B2B connection by configuring a virtual private network (VPN) (see Figure 1). This connection, along with the supporting B2B applications, allowed the two businesses to automate the insurance application approval process.

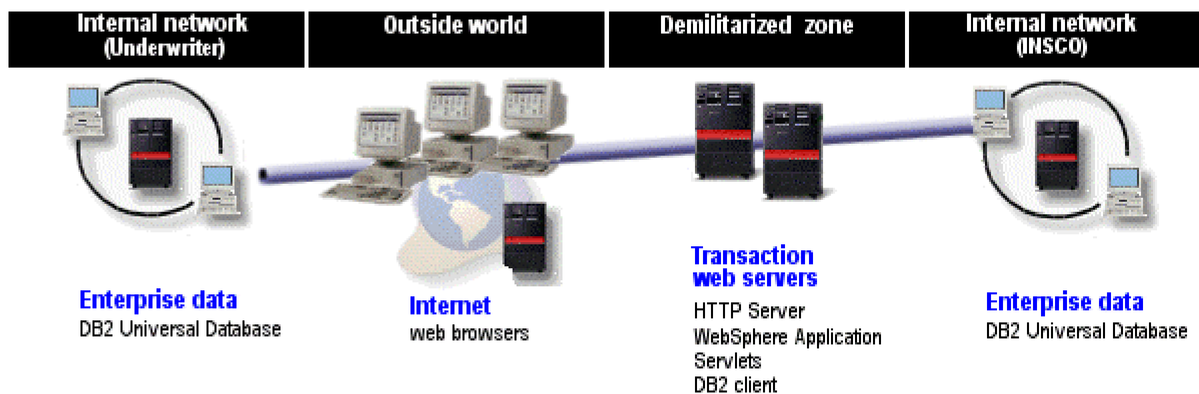


Figure 1. B2B model

Environment overview

This topic discusses the run-time environment, the system hardware, and the key products that were used for this B2B solution.

Run-time environment

The INSCO insurance company wanted to be able to provide high availability and scalability for its application, so it used dual web servers with a network dispatcher node to handle the expected

traffic. It also needed to protect its internal network, so it established a screened subnet security architecture using two security gateways (also known as firewalls) to create a demilitarized zone (DMZ) in which the web servers would run. The web servers communicated with the back-end systems through a domain firewall. This configuration was the same as the configuration used in scenario 1.

For this scenario, several servers were required for the successful implementation of this scenario. "At the heart" of the scenario are the HTTP Servers and the WebSphere Application Servers that reside on the systems in the DMZ, the INSCO intranet, and the underwriter intranet. These servers were responsible for handling the HTTP and HTTPS requests made by INSCO customers and for retrieving the requested data from the back-end enterprise data server. They were also used by the INSCO company to communicate with the business partner. The enterprise data server, DB2 UDB for AS/400, running in the INSCO internal network provided the data access and management services. A Domain Name System (DNS) server running in the outside world provided name resolution services so that customers could refer to the transactional web servers by name (www.insco.com). We provided secure connections for our customers by using the HTTPS protocol. This required that we establish a certificate authority and create server certificates for our transactional web servers. We established the certificate authority in the outside world and used the Digital Certificate Manager feature of Operating System/400® (OS/400®) to create and manage the server certificates.

Figure 2 reflects the run-time topology used in this scenario. With scenario 2, a node was added, which represents a business partner's application and data--the underwriter. Communication with this node needs to be secure; therefore, a VPN was established to allow the two nodes to communicate.

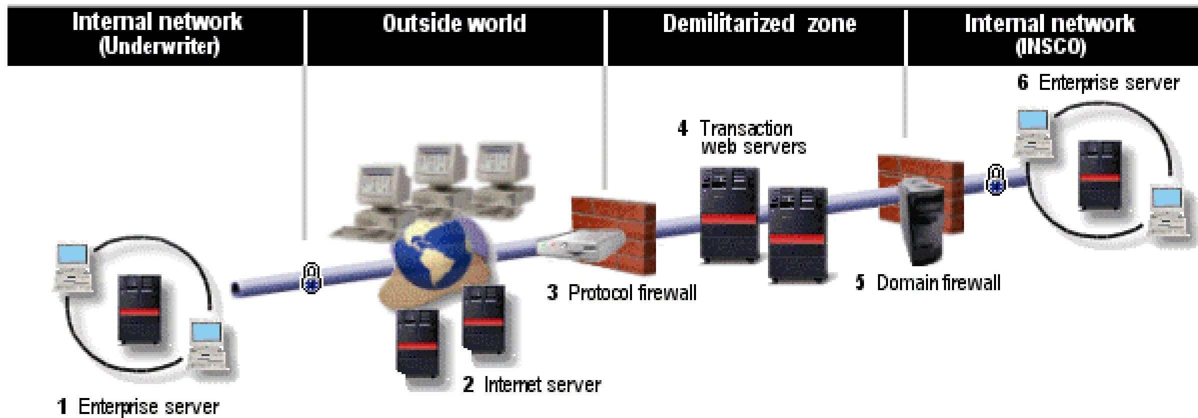


Figure 2: Run-time topology

The following list represents the details for each item depicted in Figure 2. Each system is followed by a list of the key products and features.

1. The underwriter enterprise server is an AS/400 at V4R4.
   a. DB2 UDB for AS/400

b. WebSphere Application Server 3.02 Advanced Edition for AS/400 LPO
c. IBM HTTP Server for AS/400 licensed program
d. Java Development Kit 1.1.7 licensed program

2. Outside world system is an AS/400 at V4R4
   a. Digital Certificate Manager feature of OS/400
   b. Domain Name System feature

3. Protocol firewall is implemented on an IBM 2212 Router
   a. N-ways Multiprotocol Access Services feature of the 2212 Router

4. Two transactional web servers are on two AS/400 systems at V4R4
   a. WebSphere Application Server 3.02 Advanced Edition for AS/400
   b. IBM HTTP Server for AS/400
   c. Java Development Kit 1.1.7

5. Domain firewall is installed on an Integrated Netfinity Server (Windows NT Version 4.0)
   a. IBM eNetwork Firewall for NT Version 4.1

6. INSCO enterprise server is an AS/400 at V4R4
   a. DB2 UDB for AS/400
   b. WebSphere Application Server 3.02 Advanced Edition for AS/400
   c. IBM HTTP Server for AS/400
   d. Java Development Kit 1.1.7
   e. WebSphere Payment Manager for AS/400 Version 2.1
   f. Directory Services for AS/400 feature of OS/400

System hardware

Two AS/400 systems were used in our scenario implementation: two 8-way systems. On the first 8-way system, two logical partitions were created. Two logical partitions were created on the first system, and three logical partitions were created on the second system.

> **Note:** The system hardware details are available in the "System hardware" section of the "Appendix."

Logical partitioning is the ability to make a single multiprocessing AS/400 system run as if it were two or more independent systems. Version 4 Release 4 Modification 0 (V4R4M0) of OS/400 introduced logical partitioning. For more on logical partitions, see the AS/400 Information Center available on the web at:

http://www.as400.ibm.com/infocenter

Key software products

The key software products used in this scenario were WebSphere Application Server Advanced Edition for AS/400, WebSphere Payment Manager for AS/400, Lightweight Directory Access Protocol (LDAP), and AS/400 virtual private networking (VPN). In addition, key elements used were LotusXSL (eXtensible Stylesheet Language), IBM XML4J parser, and XML Lightweight Extractor (XLE). The following sections describe these products and contain links to more information.

> **Note:** The AS/400 software product inventory for this scenario is available in the "System software requirements" section of the "Appendix."

About the WebSphere Application Server

IBM WebSphere Application Server is a robust deployment environment for e-business applications. The Standard Edition lets you use Java servlets, JSP technology, and XML to quickly transform static web sites into vital sources of dynamic web content. The Advanced Edition is a high-performance Enterprise JavaBeans™ (EJB) server for implementing EJB components that incorporate business logic. The Enterprise Edition integrates EJB and Common Object Request Broker Architecture (CORBA) components to build high-transaction, high-volume e-business applications.

Detailed information on IBM WebSphere Application Server for AS/400 (Advanced and Standard Editions) is available on the web at:

http://www.as400.ibm.com/products/websphere/

About the WebSphere Payment Manager

IBM WebSphere Payment Manager for AS/400 provides secure, electronic payment processing to Internet merchants. IBM WebSphere Payment Manager works with payment cassettes to support the SET™ Secure Electronic Transaction™ and CyberCash payment protocols. Payment Manager integrates with merchant software systems and provides cash register-like functionality to manage payment processing. Payment Manager receives payments, and processes those payments with banks and other financial institutions.

Detailed information on IBM WebSphere Payment Manager for AS/400 is available on the web at:

http://www-4.ibm.com/software/webservers/commerce/payment/

About Lightweight Directory Access Protocol (LDAP)

LDAP is a directory service protocol that runs over TCP/IP. The LDAP directory service follows a client/server model. One or more LDAP servers contain the directory data. An LDAP client connects to an LDAP server and makes a request. The server responds with a reply, or with a pointer (a referral) to another LDAP server. Because LDAP is a directory service, rather than a database, the information in an LDAP directory is usually descriptive, attribute-based information. LDAP users generally read the information in the directory much more often than they change it. Updates are typically simple, all-or-nothing changes. Common uses of LDAP directories include online telephone directories and e-mail directories. Other uses include security tasks such as authentication and authorization.

Detailed information on LDAP is available on the AS/400 Information Center on the web at:

About AS/400 virtual private networking (VPN)

A virtual private network (VPN) allows a company to securely extend its private intranet over the existing framework of a public network such as the Internet. With a VPN, a company can control network traffic while providing important security features such as authentication and data privacy.

AS/400 VPN is a function of Operations Navigator, the graphical user interface (GUI) for AS/400, that can be used to create a secure end-to-end path between any combination of host and gateway. VPNs use authentication methods, encryption algorithms, and other precautions to ensure that data sent between the two endpoints of its connection remains secure.

AS/400 VPN runs on the network layer of the TCP/IP layered communications stack model. Specifically, VPNs created using the AS/400 VPN GUI use the IP Security Architecture (IPSec) open framework. IPSec is unique in that it provides base security functions for the Internet, as well as flexible building blocks from which robust, secure virtual private networks can be constructed.

Detailed information on AS/400 VPN is available on the AS/400 Information Center on the web at:

About eXtensible Markup Language (XML)

The eXtensible Markup Language (XML) is a framework for defining application-specific data-markup languages. Markup languages, like HTML, use tags to delineate each piece of data found within an overall document. Unlike HTML, XML is focused on describing data and its structure independent of the way the data is presented to the user. This, plus the fact that XML is

a standard, platform-neutral technology, makes XML a good choice for applications that need to exchange information, especially those that cross enterprise boundaries.

Detailed information on XML is available on the web at:

http://www.ibm.com/developer/xml/

About XML Lightweight Extractor (XLE)

XML Lightweight Extractor (XLE) allows a user to annotate a given document type definition (DTD), then extract XML documents conforming to that DTD from underlying data sources. The template at the heart of all XML data interchange is a DTD, which defines the semantics of an XML document. Organizations like XML.org are coordinating vendor-neutral efforts to standardize DTDs for vertical markets.

In many applications, it is often necessary to generate XML documents from existing data sources so that the documents conform to certain given DTDs. XLE allows a user to complete this task in a simple and flexible way, without requiring the user to write detailed access queries (for example, SQL).

Detailed information on XLE is available on the web at:

http://www.alphaworks.ibm.com/tech/xle

About LotusXSL (eXtensible Stylesheet Language)

LotusXSL provides a mechanism for formatting and transforming XML, either at the browser or on the server. It allows the developer to take the abstract data semantics of an XML instance and transform it into a presentation language such as HTML or into another XML document type.

LotusXSL implements an XSL processor in Java that can be used from the command line, can be used in an applet or a servlet, or can be used as a module in other programs.

Detailed information on LotusXSL is available on the web at:

http://www.alphaworks.ibm.com/tech/LotusXSL

Application overview

This section introduces the INSCO and the Underwriter applications. It then describes the two key products needed to support the INSCO application: WebSphere Application Server security using LDAP and database. Next, the application overview describes the flow of the Underwriter and INSCO B2B applications. Finally, it describes the following key products needed to support the B2B application: XML, XLE, and VPN.

The INSCO company is extending the services it provides to include enhanced policy access, online payment, and online application submission:

- **Policy access.** Customers will be able to view policy information and update personal data. Agents will also have access to the policy information and personal data for the customers that they support. Account administrators will be able to view and update information about all customers.

- **Online payment.** Customers will be able to pay semiannual insurance premiums over the web by using a credit card and a secure electronic payment method, CyberCash.

- **Online application submission.** Agents will be able to submit new insurance applications over the web using a simple browser interface. Once the application is submitted and validated, a request is sent to an insurance underwriter.

When developing these new services, INSCO defined the following roles and responsibilities:

Customers. These are the people who buy services from the insurance company and keep an active account. These people traditionally deal directly with an insurance agent for any activity associated with their policies. With the new e-business services provided by this company, the customers will be able to use any browser to access the insurance company's web page, log on with a user ID and password associated with their account, and view information about any of their individual policies. They will also be able to update their individual account information to indicate change of address, for example. Another new e-business service that customers will have at their disposal is the ability to pay their policy premiums over the Internet using a credit card. Not all customers will take advantage of this service, but some will enjoy the convenience and paperwork reduction of doing it this way. Customers expect the insurance company to maintain the privacy of their account information by making sure that only those authorized to see their information (such as their agent) are able to see it. They also expect their account and policy information as well as credit card information to be protected as it flows over the Internet.

Agents. Agents are employees of the insurance company and conduct the day-to-day business of dealing with customers, filing claims, and soliciting new business. The new e-business services being introduced will make agents' lives easier by allowing them to be more mobile as well as more responsive to their customers' needs. With a simple browser, the agent will be able to access the policy information for the customers they support and update personal information for any INSCO customer. They will also be able to submit policy applications for any customer and create accounts for new customers. The agents expect to be able to conduct business whether they are in their office or working from a remote location. They expect to be able to access information about all their clients without requiring unique credentials for each client, and they expect to conduct their business securely.

Account administrators. Account administrators are employees of the company who have been assigned the responsibility of ensuring that day-to-day operations run smoothly. As such, they are given the privileges required to access and modify accounts and policies of any customer. Account administrators expect easy-to-use tools to do their job, as well as unfettered access to all policy information once appropriate credentials are supplied.

Security administrators. Security administrators are employees of the company who have the responsibility of developing and implementing the company's security policy. As such, they are able to create and modify user registries and authorization databases or directories. They can handle customer and agent requests to change or reset passwords or to temporarily set individual authorizations. Security is important to this company and its customers.

Underwriters. Underwriters are the employees of a company who are responsible for approving new policy applications. They review new applications, make an assessment on the insurability of the applicant, and determine the rate to be applied to this policy. Traditionally, this is largely a manual process. With the new e-business services being introduced, some of these processes can be automated. For example, new policy applications that adhere to certain criteria (existing customer in good standing, maximum face value of policy, standard contract) can be automatically approved or conditionally approved pending credit approval. This greatly reduces the workload for the underwriter. The underwriter expects to be able to establish the rules associated with automatic underwriting and modify or restrict those rules at any time. They expect to be notified immediately of all new applications that require manual approval as well as have access to a log of applications that are automatically approved.

Application design decisions

The following describes the design decisions that were made up front:

- The payment processing portion of the INSCO application was written as a separate servlet. This allows the INSCO company to change its Payment Manager APIs in the event that the Payment Manager changes or if INSCO decides to go with a different payment solution.

- For simplicity, we will only include the main driver on the auto insurance policy, which will be the customer.

- Underwriters may choose to implement their database as they want.

- We will use a data queue to generate the next available customer number and policy number to avoid the issuance of duplicate numbers.

- There will not be an update policy function. To update a policy, the current policy would need to be deleted and a new one created. To delete a policy, the agent must contact the account administrator who would delete the policy.

- Customers will only see and be able to update the following information: first name, last name, middle initial, address, zip code, e-mail address, home phone, and work phone.

- Agents will only be allowed to view those policies that they are responsible for. They will be allowed to change any customer's personal information, which includes first name, last name, middle initial, address, zip code, e-mail, home phone, and work phone. Agents will also be allowed to submit an application for any customer. These changes are based on the LDAP design we chose.

INSCO application

An INSCO user (customer or agent) will log on using a form-based authorization, which calls the InscoLoginServlet. The InscoLoginServlet uses WebSphere authentication to verify the user. Once the user has been verified, the user information will be stored as session data, and the user will be redirected to the Insco2Servlet. At this point, the Insco2Servlet becomes the controlling servlet. The Insco2Servlet will access an LDAP directory to provide the user type (that is, customer, agent, or account administrator), which will allow the servlet to load the proper JSP file. For a customer, the Customer Home Page will be loaded. For an agent or account administrator, the Select Customer Information page will be loaded.

The flow of a customer's interaction with the INSCO application is shown in Figure 3. The numbered transitions are described in the steps that follow.

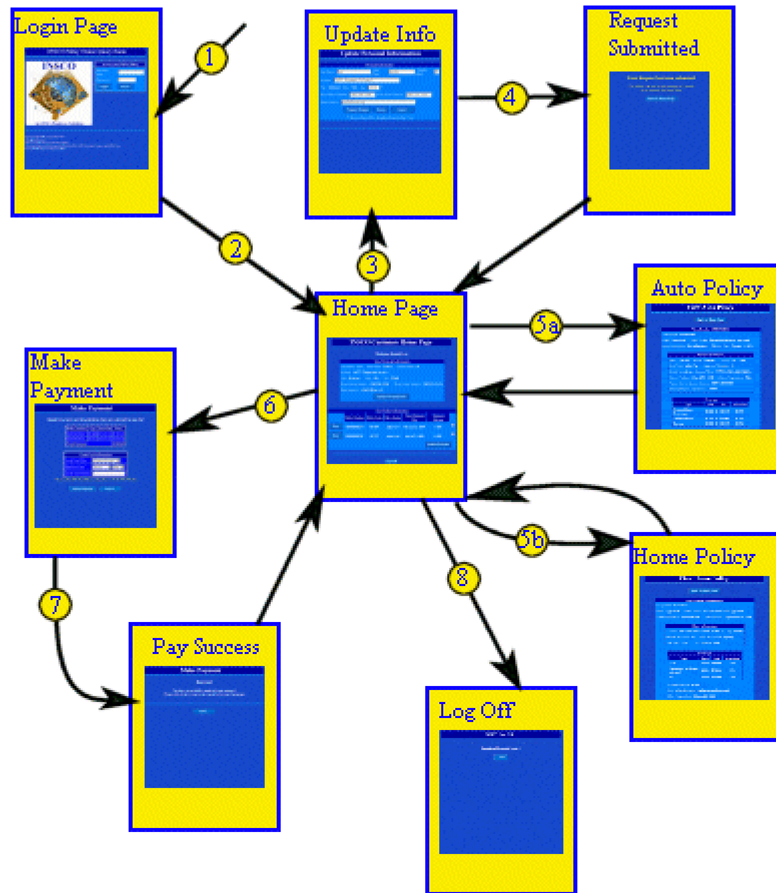


Figure 3. Customer flow

1. Using a web browser, customers access http://www.insco.com/inscologin.html to load the login page.

2. Customers enter their account number and password and click Login. At this point, customers are authenticated by IBM Websphere Application Server security using LDAP, and the Customer Home Page is loaded. From the Customer Home Page, they can update their personal information, view the details of any of their policies, submit a payment on one or more of their policies, or log off of the INSCO application.

3. From the Customer Home Page, customers can change their name, address, and phone number by clicking the Update Personal Info button. This sends them to the Update Personal Information page, which allows them to view and update their personal information. While updating their personal information, customers can use the Reset button to refresh the values of all fields to their initial values. Customers can also click the Cancel button to return to the Customer Home Page without making any changes.

4. To actually update their information, customers must click the Submit Changes button. This will send them to the Request Submitted page. On the Request Submitted page, they

are automatically redirected to the Customer Home Page after five seconds. They may also click the Back to Home Page button to return there as well.

5. From the Customer Home Page, customers can also view the details of a home or auto policy by clicking the View button corresponding to the policy they want to view. When customers finish viewing their policy information, they can click the Back to Home Page button to return to the Customer Home Page.

   a. If the selected policy is an auto policy, customers are sent to the View Auto Policy page detailing the selected policy's information.

   b. If the selected policy is a home policy, the customer is sent to the View Home Policy page detailing the selected policy's information.

6. Customers can also submit an online payment from the Customer Home Page. To submit a payment, they select the policies they would like to make a payment for and click the Submit Payment button. This will direct them to the Make Payment page. The Make Payment page provides a form for them to enter their credit card information and allows them to verify that the correct policies were selected and that the total payment amount is correct. If customers decide against making the payment, they can click Cancel and return to the Customer Home Page.

7. To make the payment, the customer must click the Make Payment button. After clicking the Make Payment button, the IBM WebSphere Payment Manager for AS/400 is used to process the customer's payment. Upon completion, the customer is sent to the Pay Success page. On the Pay Success page, the customer may click the Home button to return to the Customer Home Page or may wait five seconds to be automatically redirected.

8. When customers finish using the INSCO application, the customer should click Log Off from the Customer Home Page to leave the INSCO application securely. By clicking Log Off, no one else using the customer's browser will be able to view any of the customer's personal information from the INSCO web site.

The flow of an agent's interaction with the INSCO application is shown in Figure 4 with the numbered transitions described in the steps that follow.

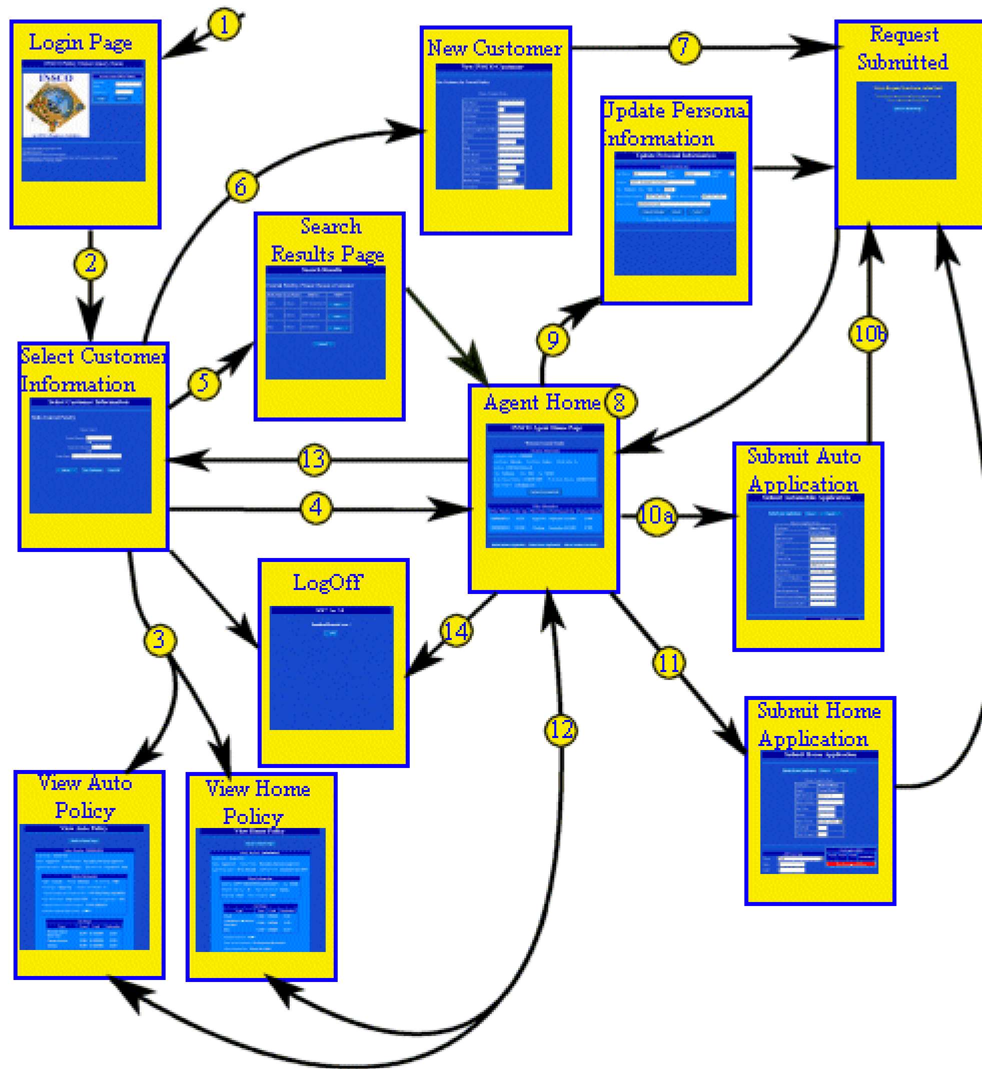


Figure 4. Agent and account administrator flow

1. Using a web browser within the INSCO intranet, agents access http://sys.bus.com/inscologin.html to load the initial page.

2. Agents enter their account number and password and click Login. At this point, the agent is authenticated by IBM Websphere Application Server security using LDAP and the Select Customer Information page is loaded. The Select Customer Information page provides several options for agents. Agents can search on a policy number, a customer number, or a customer's last name; can create a new customer; or can log off the INSCO application.

3. On the Select Customer Information page, agents can enter a policy number and click the Select button. LDAP is used to check agent authorization to the policy. If agents have authorization to access the policy, the View Auto Policy page or the View Home Policy

page is displayed with the policy's information. When agents have finished viewing the policy details, they can click the Back to Home Page button to go to the Agent Home Page. See item 8 below for more information on the Agent Home Page.

4. Agents also can search on a customer number from the Select Customer Information page. After entering a customer number and clicking the Select button, agents are sent directly to the Agent Home Page if the customer exists. The Agent Home Page contains the information for the customer whose number was entered by agents.

5. From the Select Customer Information page, agents can do a fuzzy search on a customer's last name. After entering all or part of a customer's last name, agents click the Select button and is sent to the Search Results page. The Search Results page provides a list of customers whose last name matches or starts with the name provided by the agent. If agents find the customer in the list that they would like to work with, they can click the View button corresponding to that customer. This will send agents to the Agent Home Page to work with that customer. If agents do not find the customer they are looking for, they can click Cancel and return to the Select Customer Information page.

6. Agents can create a new customer by clicking the Create New Customer button on the Select Customer Information page. This sends agents to the New Customer page containing a form to fill in all of the customer's personal information. If agents decide not to create a new customer account, they may click Cancel to return to the Select Customer Information page.

7. When all required fields are filled in, agents may click Add Customer to actually create the new customer account. After clicking Add Customer, agents are sent to the Request Submitted page, which confirms the account was added. Agents can return to the Agent Home Page by clicking Back to Home Page or wait to be redirected there after five seconds.

8. The Agent Home Page is similar to the Customer Home Page. It contains the same information that the customer sees, but with several major differences. First, the View buttons only appear for policies that agents are authorized to access. The check boxes for each policy and the Submit Payment button have been removed. Finally, agents can also submit auto and home applications or can return to the Select Customer Information page.

9. From the Agent Home Page, agents can update the customer's personal information by selecting the Update Personal Info button. This operation is the same as it is for customers. For more information on the flow of updating a customer's personal information, see items 3 and 4 of the customer flow.

10. From the Agent Home Page, agents can submit a new auto policy application by selecting Submit Auto Application.

   a. This will direct agents to the Submit Auto Application page, which allows agents to enter the information for the auto to be covered and to select the desired coverage. If agents decide not to submit the application, they can click the Cancel button to return to the Agent Home Page.

   b. After agents enter all of the specific policy information, they can select the Submit Auto Application button to submit the application. If the application submission completes successfully, agents are sent to the Request Submitted page. Agents can continue to the Agent Home Page by selecting the Back to Home Page button, or they are automatically redirected there after 5 seconds.

11. From the Agent Home Page, agents can submit a new home policy application by selecting Submit Home Application. The flow of submitting a home application is the same as the flow for submitting an auto application. See the previous item for more information.

12. By clicking one of the View buttons on the Agent Home Page, agents can view a policy. This operation flows the same as it does for customers from their respective home page. For more information, see item 5 of the customer flow for a detailed description.

13. Agents may return to the Select Customer Information page by clicking the Select Another Customer button on the Agent Home Page.

14. Agents can log off of the INSCO application from either the Agent Home Page or the Select Customer Information page by clicking Log Off. This securely logs agents off of the application and prevents others who are using an agent's browser from viewing any sensitive information that an agent may have viewed while working from the INSCO web site.

WebSphere security using LDAP

In this scenario, INSCO used WebSphere Application Server security to secure its web resources. INSCO implemented this security by using the Lightweight Third-Party Authentication (LTPA) framework. LTPA accesses the Lightweight Directory Access Protocol (LDAP) to authenticate users. LDAP is provided on the AS/400 as a part of the operating system and is known as Directory Services. The details of LTPA are described later in the INSCO application section.

INSCO used LDAP exclusively for authentication and authorization:

- **Authentication** provides verification of the user’s identity. This was accomplished by prompting the users for their user ID and password. These credentials were then passed from the WebSphere Application Server to the LDAP server for verification.

- **Authorization** allows users access to specific objects. Authorization was handled in the application by binding to the LDAP server with the user’s credentials and attempting to

access a certain policy. If the attempt was successful, the user has authority to that policy within the INSCO application.

The following diagram depicts the Directory Information Tree (DIT) that was defined to be used by WebSphere to authorize users.

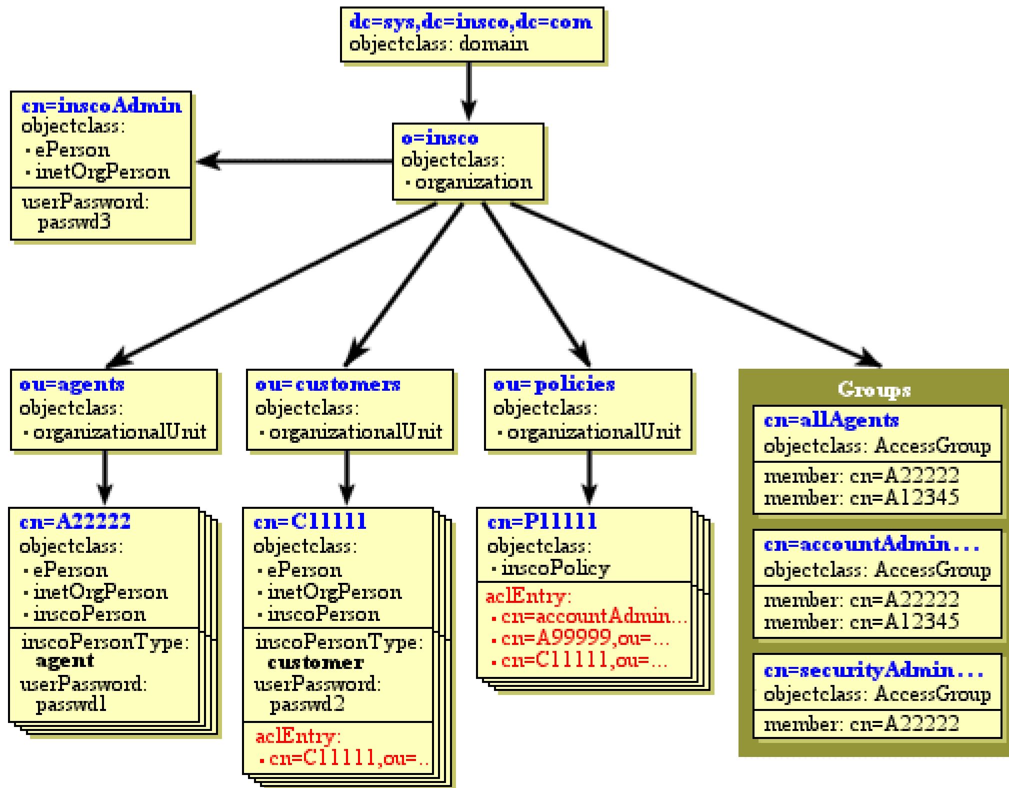


Figure 5. Directory Information Tree

Each object in a DIT has a distinguished name (DN) associated with it. A DN is composed of a sequence of relative distinguished names (RDNs) separated by commas. The sequence of RDNs makes up a DN that identifies the parents of a directory entry up to the root of a DIT. For example, in this scenario, the DN `cn=A0000001,ou=agents,o=insco,dc=sys,dc=insco,dc=com` represented the person with the common name (cn) A0000001 under the organizationalUnit (ou) agents in the organization (o) insco, which was under the domain (dc) sys.insco.com.

The INSCO DIT was broken up into three main subtrees: agents, customers, and policies. In addition, several groups were defined to make assigning authority easier. An account administrator, inscoAdmin, owned and could access all other objects in the DIT. INSCO used the following default objectclasses provided by LDAP: domain, organization, organizationalUnit, AccessGroup, inetOrgPerson, and ePerson. In addition, INSCO defined two new objectclasses, inscoPerson and inscoPolicy, and one new attribute, inscoPersonType. Each object type of the DIT is described in the following section.

o=insco and cn=inscoAdmin
The organization o=insco and the account cn=inscoAdmin were created and owned by the LDAP administrator. The o=insco entry contained all of the main organizationalUnits and groups used in this scenario. The account cn=inscoAdmin had an objectclass of inetOrgPerson and ePerson. This account was responsible for all entries below the o=insco entry and was the only person who had access to the cn=securityAdministrators group.

Customers
Each of the insurance company's customers had a corresponding account added to the ou=customers container. These entries had the objectclass inetOrgPerson, ePerson, and inscoPerson. The addition of inscoPerson was needed so INSCO could add an attribute to each entry specifying the type of account. For customers, the attribute inscoPersonType was set to `customer`. An aclEntry attribute was also added to each entry to allow customers to access their own entry. After customer authentication, the aclEntry allowed the application to check the inscoPersonType attribute to determine if the person was a customer or agent.

Policies
Every insurance policy had a corresponding entry under the ou=policies container. The insurance policies used the objectclass inscoPolicy, and had no attributes associated with them. These entries did not contain any policy information, but instead served as access control for the application. Associated with each policy was an aclEntry attribute for each person who had access to the policy. By default, an aclEntry was added for the customer who owned the policy, the agent who created the policy, and both the cn=securityAdministrators and cn=accountAdministrators groups.

Agents
Each insurance agent had a corresponding entry created under the ou=agents container. Agents had the same objectclasses as customers. The difference was that the inscoPersonType attribute for agents was set to `agent` unless the agent was an administrator. Each agent was a member of the group cn=allAgents. They could also be members of the administrator's groups, which would give them more access.

Account administrators
If agents were members of the cn=accountAdministrators group, they had access to update and view all policies in the insurance company. However, they could not create or delete agents or change who had access to a policy.

Security administrators
The security administrators were responsible for adding or deleting agents. They could also change who had access to a policy and add agents to the cn=accountAdministrators group.

ACLs
Access control lists (ACLs) were used to control who had access to certain entries in the LDAP server. After an agent logged on and tried to view a certain policy, the application

would bind to the LDAP server with the agent's credentials and try to access the policy entry. Agents would be allowed to view the actual policy if they had access to the policy entry.

Database

DB2 UDB for AS/400 provides reliable data management and allows INSCO to access the data from multiple platforms. Again, our environment consists of AS/400e servers in the middle tier (DMZ), with our data residing on an AS/400e server in the third tier (internal network). Figure 6 depicts the INSCO database schema.

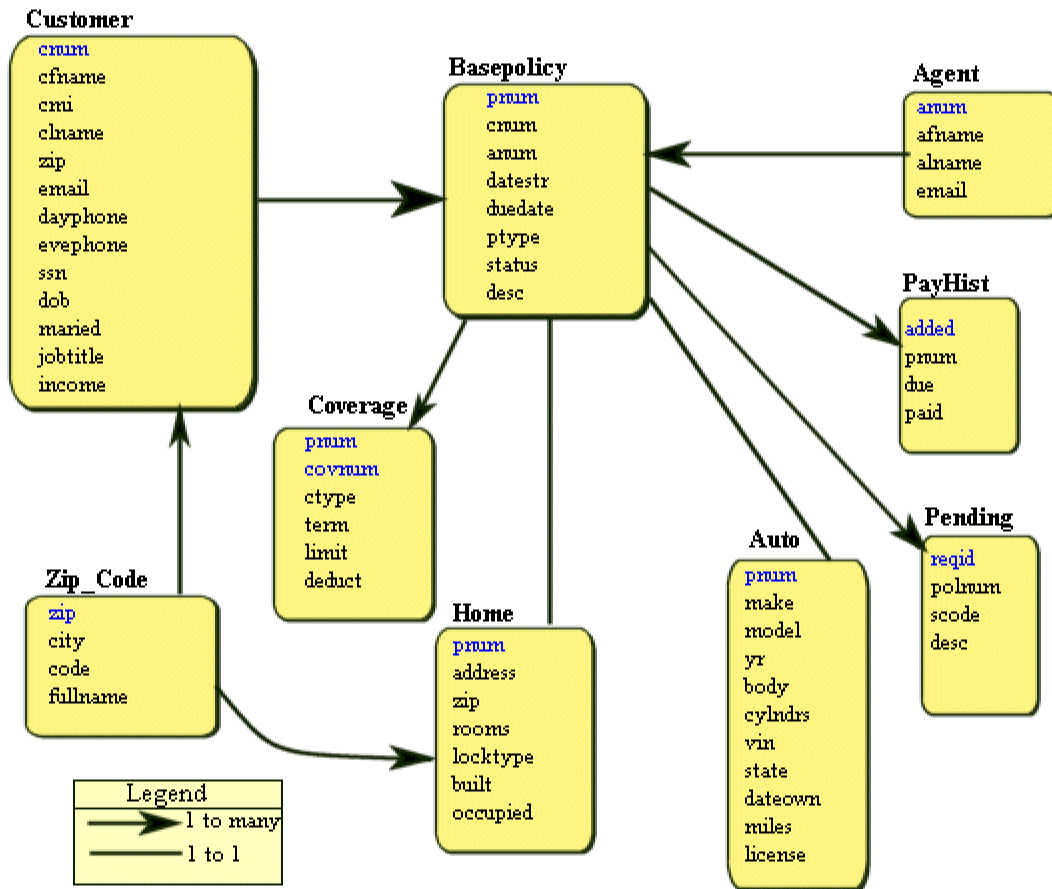


Figure 6. INSCO database schema

The following list shows the file contents for each table that is used to store the insurance company information.

Table name	File contents
Customer	Customer's personal data
Agent	Insurance agent information
Base policy	Base policy information
Coverage	Information on the policy coverage
Payment history	Payment history of a policy
Zip	Zip codes and state names
Home	Home policy information
Auto	Auto policy information
Pending	Pending policy applications

The arrows show the relationship between the tables. Note that the database was redesigned from the original INSCO database. We removed the ClientPW table and replaced it with an LDAP directory. We moved the passwords to the LDAP directory so we could use WebSphere authentication.

Note: The complete database layout can be found in the "Database details" section of the "Appendix."

Underwriter and INSCO B2B applications

The process of approving a policy application with the Underwriter and INSCO B2B applications is illustrated and described below.

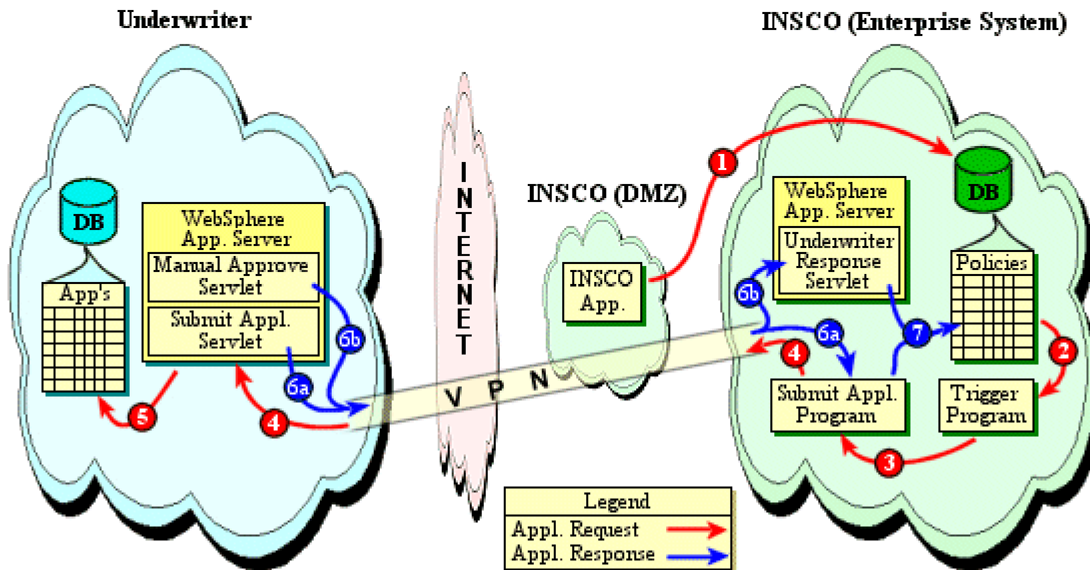


Figure 7. B2B flow

1. The INSCO application running on the DMZ web servers adds a **policy** application to the database.

2. Whenever a policy gets added to the database, a trigger program is called.

3. The trigger program invokes a Java application (SubmitApplication).

4. The INSCO SubmitApplication program, using XLE, gets the data from the database and submits the resulting XML document to the underwriter's SubmitApplicationServlet.

5. The SubmitApplicationServlet parses the XML data from INSCO, adds an entry in a database with the request identifier, and stores the XML data in a file on the server.

6a. The SubmitApplicationServlet then determines whether this application can be instantly approved or rejected. In this scenario, we base this decision solely on income. It then formulates an XML-encoded response and passes it back to the SubmitApplication program to indicate whether it was instantly approved or not.

6b. Applications that are not instantly approved or rejected must be handled manually by the underwriters. They can do so by running the ManualApproveServlet. This servlet queries the database for new applications and retrieves the XML data about the application from the file stored on the server. The servlet formats this data and presents the information to the underwriters by using LotusXSL style sheets. The underwriters can then approve or reject this application by pressing the appropriate button, and the ManualApproveServlet forwards the XML-encoded response to the UnderwriterResponseServlet on the INSCO server.

7. The UnderwriterResponseServlet or the SubmitApplicationServlet parses the received XML-encoded response from the underwriter and, based on the content, updates the database accordingly.

eXtensible Markup Language (XML)

We used XML for formatting communication between INSCO and the underwriter primarily because we wanted to use a flexible, well-defined protocol that was already defined for the industry. We found the XML standard we were looking for at Agency-Company Operations Research and Development (ACORD). They have defined XML data type definitions (DTDs) for all aspects of the insurance industry. DTDs define XML tags and their relationship with other tags found in an XML document or data stream. We found two separate and distinct insurance models defined at the ACORD site. One was strictly for life insurance. The other was more broad and encompassed auto insurance, home insurance, and many other types, called property and casualty. Because these two models were so different and to make things easier, we decided to pick only one to use. From these two models, we used the property and casualty model because it included both auto and home insurance and mapped better to our scenario.

These models were defined to include all the data that an insurance company would require to conduct business. From these models, a separate DTD is proposed and approved for each type of transaction that uses this defined data. For example, one DTD would be created to define a home-owners insurance application request to an underwriter. Not all of the available DTDs had been proposed at the writing of this report, but fortunately the following DTDs had been proposed or approved:

- Personal Auto Insurance Request
- Personal Auto Insurance Response
- Personal Home Insurance Request
- Personal Home Insurance Response

Given the status of these DTDs, we decided to limit this scenario to include only auto and home insurance. More information about the ACORD DTDs can be found at:

http://www.acord.org/standards/xml/Frame.htm

XML Lightweight Extractor (XLE)

To make the XML messages easier to create, we decided to use XLE, which is a utility provided by the IBM AlphaWorks team. XLE allows you to easily extract data out of any database and format the results into XML. This is done by creating a mapping file, which tells XLE which database tables and columns to map to which XML tags.

Virtual private networking (VPN) and IP packet security

There were several network security considerations in our scenario. We needed to provide a secure connection between the INSCO and underwriter systems for the automated approval application. We also needed to secure other network communications.

We needed to connect the INSCO and underwriter systems hosting the automated approval application. The applications communicate using the HTTP protocol. Two key security requirements for this communication are data integrity and confidentiality. Since both companies have established Internet connectivity, a VPN solution was chosen.

AS/400 VPN uses two IPSec protocols to protect data as it flows through the VPN tunnel: Authentication Header (AH) and Encapsulating Security Payload (ESP). The Internet Key Exchange (IKE) protocol, or key management, is another part of IPSec implementation. While IPSec encrypts your data, IKE supports automated negotiation of security associations and supports automated generation and refreshing of cryptographic keys.

For the automated approval application process, INSCO implemented a host-to-host dynamic VPN connection. There was no need to connect the INSCO and underwriter internal networks. Only the internal systems that were hosting the automated approval applications needed to be

connected. Use of a dynamic connection provided additional security through the use of the IKE protocol mentioned above.

IP packet security was deployed to secure network communications. IP packet filtering was implemented on the firewalls and AS/400 systems. IP packet filtering protects your system by filtering packets according to rules that you specify. You define the policies that determine the types of packets that are permitted or denied access to your system or network.

Network address translation (NAT) was also used to hide internal IP addresses of INSCO internal servers from public knowledge. Static NAT, which is a one-to-one mapping of IP addresses, was used. NAT was configured on the INSCO domain firewall. Static NAT rules were defined to map the internal addresses of the INSCO servers to public IP addresses.

INSCO application

The goal of the INSCO application was to allow the insurance company to extend its customer service capabilities for external customers and to provide service capabilities for its internal customers (that is, the insurance agents).

The INSCO application was deployed on the transactional web servers residing in the DMZ and in the intranet. The three web servers, with IBM HTTP Server and IBM WebSphere Application Server installed, were equipped with identical versions of the application. The two web servers used in the DMZ allowed the implementation to handle more customer requests with greater efficiency. The one web server used in the intranet allowed the agents to access the application internally.

INSCO server setup

For the INSCO application, the following servers had to be set up:

- WebSphere Application Server 3.02 Advanced Edition
- LDAP 2.1
- Payment Manager 2.1

The following sections describe how each of these servers was set up and configured.

Setting up the WebSphere Application Server

For our implementation, we used IBM WebSphere Application Server 3.02 Advanced Edition for AS/400. We used the Advanced Edition because WebSphere Application Server Standard Edition for AS/400 Version 3.02 was not available. Once the Standard Edition is made publicly available, it could be used in place of the Advanced Edition.

Installing WebSphere Application Server

The setup of IBM WebSphere Application Server 3.02 Advanced Edition was completed using the product documentation. No special considerations were made for our particular implementation. The general setup steps used in our implementation follow:

1. Check the prerequisite requirements for installing and running WebSphere Application Server.
2. Install WebSphere Application Server on the AS/400.
3. Start the WebSphere Application Server environment.

   On the AS/400 command line, enter STRSBS QEJB/QEJBSBS to start the QEJB subsystem and the default administrative server.

4. Verify that the WebSphere Application Server environment has started.
5. Install the WebSphere Administrative Console on a workstation that is located in the same domain as the WebSphere Application Server.

   The Administrative Console is used for configuring and managing WebSphere Application Server for AS/400. It is necessary to install the Console before using WebSphere Application Server.

6. Start the WebSphere Administrative Console on the workstation.
7. Create a new HTTP server configuration.
8. Configure IBM HTTP Server for AS/400 to support WebSphere Application Server.
9. Start an HTTP server instance.
10. Verify the installation.
11. Check the WebSphere Application Server Advanced Edition V3.02 PTF web page to verify that the latest Advanced Edition program temporary fixes (PTFs) are installed on the AS/400.

    Make sure that you install PTF SF99028 and PTF SF99029. The page is available on the web at:

    http://www.as400.ibm.com/products/websphere/services/service.htm#AE

For complete documentation on setting up WebSphere Advanced Edition on AS/400, refer to *IBM WebSphere Application Server Advanced Edition for AS/400* available on the web at:

http://://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

About WebSphere Application Server security

It was desirable for INSCO to provide a form-based authentication (inscologin.html). To provide this type of authentication, the WebSphere Application Server was set up to use the custom challenge mechanism. Custom challenge is useful when one wants the server to use an HTML form to retrieve the user ID and password. The use of an HTML form allows the application to retain the user ID and password for later use. For example, the INSCO servlet requires the user ID and password to bind to the LDAP directory for agent authorization to policies.

To implement the custom challenge mechanism, Lightweight Third-Party Authentication (LTPA) needed to be utilized. LTPA accesses the LDAP, and LDAP contains the user registry against which authentication is performed. In this scenario, the user registry was populated with the user ID and password from the existing database tables. The passwords were only stored in the LDAP directory. To allow users to change their passwords or to apply password policy rules, an application would need to be written. In this scenario, a password changing tool was not implemented; therefore, users would need to contact the security administrator to change their password.

Using LTPA, WebSphere allows users to authenticate once per session. In our scenario, the web servers were located in the DMZ, and the LTPA and LDAP servers were located in the INSCO intranet. A typical authentication flow and description follow:

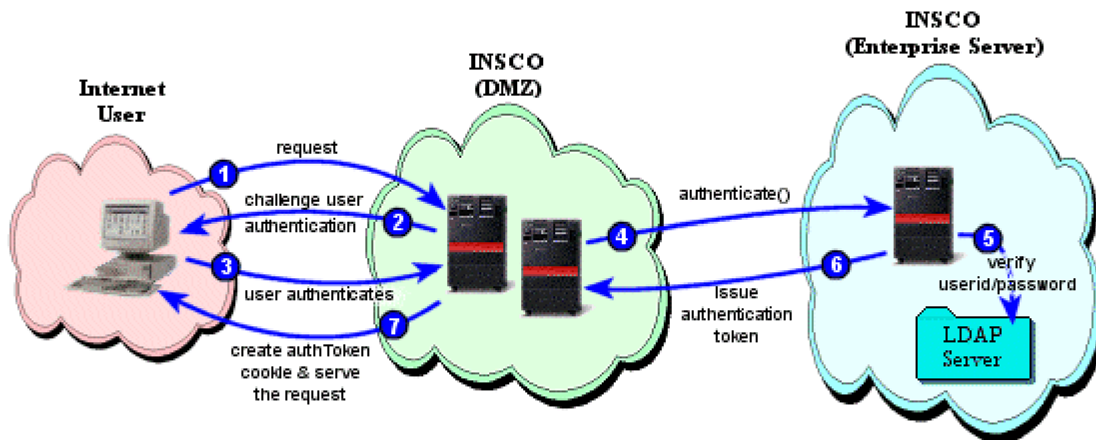


Figure 8. LTPA authentication flow

1. A user who has yet to be authenticated issues a request to access a secure web resource that is configured to use the Custom login challenge mechanism.
2. The web server challenges the user by redirecting the browser to a uniform resource locator (URL) configured as the login URL. To perform a form-based login, this URL points to an HTML file containing a form to request a user ID and password.
3. The user then responds to the challenge by providing the authentication information.
4. When the form is submitted, the web server contacts the security application, a servlet that performs a customized login, and provides the authentication information. If the user is authenticated, a token is issued to the web server.
5. When a request for authentication comes in from a web server, the LTPA server will use the LDAP user directory to authenticate the user.
6. If the user's authentication data is verified, the LTPA server issues an LTPA token to the client.
7. The web server receives the token issued to the user and serves the requested resource. If WebSphere Single Sign-On (SSO) is enabled, the web server will store the token in the user's browser as a cookie.

To maintain the authentication state across multiple web requests, the WebSphere SSO framework must be used. The use of SSO requires LTPA to be the authentication mechanism. WebSphere includes a CustomLoginServlet sample that extends the AbstractLoginServlet that can be modified to perform a customized login. For an overview of WebSphere Application Server security, refer to *IBM WebSphere Standard/Advanced 3.02 Security Overview* available on the web at:

http://www-4.ibm.com/software/webservers/appserv/security.pdf

Configuring WebSphere Application Server security

This topic provides an overview of the steps necessary for configuring WebSphere Application Server security.

> **Note:** Due to the complexity of configuring WebSphere Application Server security, the exact steps are provided in the "Appendix."

To configure WebSphere Application Server security:

1. Enable security by selecting **Specify Global Settings** in the WebSphere Administrative Console.

2. Enable the use of authorization services for the HTTP Server configuration.

3. Secure the WebSphere resources.

   Turning on security disables any Enterprise JavaBeans (EJBs) beans until security has been configured for these applications. In our scenario, we did not have any EJBs deployed under the WebSphere Application Server.

For complete documentation on setting up WebSphere Advanced Edition security on AS/400, refer to *IBM WebSphere Application Server Advanced Edition for AS/400* available on the web at:

http://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

WebSphere security discoveries

The following discoveries are useful when configuring custom authentication. Make sure that the following items are defined correctly:

1. Under the Global Security settings:
   a. Make sure you enable SSO.
   b. Make sure that your domain name is fully qualified. For example, if your machine name is `foo.mydomain.com`, then your domain name should be `mydomain.com`.
   c. Make sure that the login URL is fully qualified and that the host name is within the domain that you specified.
   d. If your domain is a normal top-level domain, `myserver.com`, make sure to set your realm and domain names to `.myserver.com`. If your domain is an extended domain, `myserver.ny.us`, make sure to set your realm and domain names to `myserver.ny.us`.
2. Make sure that the redirect URL is fully qualified in the HTML form used for authentication. This refers to the "jumpto" URL within the CustomLoginServlet.

3. All resources that are accessed by a secured resource must also be secured. This includes all servlets and all JSP files.
4. The following changes need to be made to the AbstractLoginServlet:

```
try {
  retCreds = authenticator.login(userid, password, true);
   Authenticator.setInvocationCredentials(retCreds);        // add this line
} catch(Exception e) {                    // catch all possible exceptions
   throw new ServletException();              }

// do whatever is required of postlogin
// move this line from its previous location (just after the setSSO code block) to here.

postLogin(req, res);

// apart from performing authentication, set up the SSO cookie so that
// WebSphere can use that instead of prompting for used and password
// or in the case of Custom challenge type will not redirect the
// user back to the logic form
if (sets) {
  try {
   setupSSO(userid, password, req, res);
  } catch (RemoteException e) {
    throw new ServletException("Error setting single sign-on");
  } catch (org.omg.SecurityLevel2.LoginFailed e) {
       throw new ServletException("Login Failed");
  }
}
```

Setting up the LDAP server

The LDAP server is located on the enterprise server and is accessed from the transactional web servers via the default LDAP port, 389. To set up the LDAP server, we had to update the configuration files, configure the server through Operations Navigator, and populate the LDAP server with initial data from the database.

Because we decided to add custom objects and attributes to the DIT, we needed to update the LDAP server's configuration files to reflect this. The following objects were added to the end of the /QIBM/UserData/Os400/DirSrv/UsrOC.txt file on the enterprise server:

```
objectclass   inscoPerson
       requires
              objectClass
       allows
              inscoPersonType

objectclass   inscoPolicy
       requires
              objectClass,
              cn
```

This adds two objects to the LDAP server. The inscoPerson object allows the inscoPersonType attribute. The inscoPolicy object requires the cn attribute and allows no other attributes.

We also added the following line to the /QIBM/UserData/Os400/DirSrv/UsrAT.txt file:

```
attribute   inscoPersonType     cis    inscoPersonType    15    normal
```

This line indicates that attribute inscoPersonType can be added to objects, and it can have a maximum of 15 characters. This attribute will be set to either customer, agent, securityAdmin, or accountAdmin, based on what type the person is. Once these changes were made, we were ready to create and configure our LDAP server.

To configure a new LDAP server:

1. Within Operations Navigator, expand **Network** **Servers** **TCP/IP,** right-click **Directory**, and select **Configure** to configure a new LDAP server. We used the following values in the configuration wizard:

Library	/QSYS.LIB/LDAPINSCO.LIB
Administrator Name	cn=administrator
Administrator Password	admin
Directory Suffix	dc=sys,dc=insco,dc=com

2. Create an index on the `cn` attribute (to improve the performance of searches).

   a. Right-click **Directory.**
   b. Select **Properties**.
   c. Click the **Performance** tab.
   d. Click **Indexing Rules...** and add `cn` to the list of indexed attributes (which is initially empty).

3. Populate the LDAP server with initial data.

   We wanted to create an LDAP entry for each agent, customer, and policy currently in the database, so we wrote a simple Java program that queried the database and generated the LDIF files for LDAP. LDIF (LDAP Directory Interchange Format) files are specially formatted text files that provide a simple way to import a lot of data into an LDAP server at once. Our Java program generated a separate LDIF file for agents, customers, and policies, and then generated another LDIF file to import the groups that we needed. We also needed to write LDIF files to create the DIT structure.

Below is an example of a policy entry:

```
dn: cn=P55557,ou=policies,o=insco,dc=sys,dc=insco,dc=com
cn: P55557
aclEntry: group:cn=accountAdministrators,o=insco,dc=sys,dc=insco,dc=com:
       object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: access-id:cn=C22223,ou=customers,o=insco,dc=sys,dc=insco,dc=com:
       object:a:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: access-id:cn=A11113,ou=agents,o=insco,dc=sys,dc=insco,dc=com:
```

```
        object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com
entryOwner: group:cn=securityAdministrators,o=insco,dc=sys,dc=insco,dc=com
objectclass: inscoPolicy
```

All the aclEntry lines must be on a single line, but they are broken up here for presentation purposes. Only entryOwners are permitted to change an entry's access control list (ACL) attributes, so we made the group cn=securityAdministrators and the entry cn=inscoAdmin entryOwners. We also added an aclEntry attribute to each policy for the group cn=accountAdministrators so that account administrators could access every policy, for the customer who owns the policy and for the agent who created the policy. Agent and customer entries were set up similarly.

Note: Examples of agent and customer entries are listed in the "Appendix."

After we created the LDIF files, we imported them into LDAP. First, we had the main LDAP administrator import the top-level objects of the DIT, under which all the INSCO objects exist, and import the cn=inscoAdmin entry. Then, we had the cn=inscoAdmin user import the rest of the DIT and all the agents, customers, policies, and groups. To import the LDIF files into the LDAP server, we issued the following LDAP utilities from QSH shell on the enterprise server:

```
ldapadd -D "cn=administrator" -w admin -f "/insco/adminDIT.ldif"
ldapadd -D "cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com" -w insco -f "/insco/ldap/inscoDIT.ldif"
ldapadd -D "cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com" -w insco -f "/insco/ldap/agents.ldif"
ldapadd -D "cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com" -w insco -f "/insco/ldap/customers.ldif"
ldapadd -D "cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com" -w insco -f "/insco/ldap/policies.ldif"
ldapmodify -D "cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com" -w insco -f "/insco/ldap/groups.ldif"
```

The group objects cn=securityAdministrators, cn=accountAdministrators, and cn=allAgents were created in the inscoDIT.ldif file, but they were populated in the groups.ldif file. The entire LDAP data population took about 6 hours to complete in our scenario, which had 1000 agents, 30,000 customers, and 60,000 policies. The reason this took so long was due to the heavy use of ACLs in the LDAP server.

For more information about setting up and using the LDAP server on the AS/400, see the AS/400 Directory Services web site at:

http://www.as400.ibm.com/ldap

LDAP discoveries

The following discoveries are useful when setting up an LDAP server:

- In an LDIF file, you cannot change the ownerPropagate or inheritOnCreate attribute of an entry without also changing the entryOwner attribute.

- In an LDIF file, you cannot change the aclPropagate attribute without also changing the aclEntry attribute.

- There are two ways to import data into an LDAP server. One is by importing through Operations Navigator, and one is by calling the LDAP utilities (such as ldapadd) from the QSH shell. We found that the LDIF file you use must contain different attributes based on which method you use. To import data from Operations Navigator, you are required to specify the ownerSource and aclSource attributes or you cannot change any of the entryOwner or ACL attributes. This is not valid when using the LDAP utilities, where you cannot specify ownerSource or aclSource, or the command will fail.

Setting up the Payment Manager

The process for a merchant to become fully functional with a Payment Manager begins with the merchant going to a financial institution to set up bank accounts and choosing what credit cards they would like to support. The merchant would then register with CyberCash. The financial institution would give CyberCash information about the new accounts. CyberCash gives merchants a CyberCash ID and a CyberCash merchant key for each account they have with CyberCash, and that information is used to configure the CyberCash cassette. The following web pages explain these steps in detail.

These web sites give a good overview of what is involved in registering with CyberCash:

https://amps.cybercash.com/
http://www.cybercash.com/cybercash/merchants/credit_start.html

This web site has a step-by-step guide of the steps a merchant would take. In our case, we do not have to download any software:

http://www.cybercash.com/cybercash/merchants/docs/html/get_started.html

This web site contains the *MCK Planning Guide*, which walks through the process on page 31 (Integrating Your Storefront with Cash Register Service). Again, we did not need the MCK or any CyberCash software because we have integrated all of that into the cassette.

http://www.cybercash.com/cybercash/merchants/docs/plan.pdf

Installing Payment Manager Framework and CyberCash cassette

The following describes how to install and set up the Payment Manager.
The Restore Licensed Program (RSTLICPGM) CL command is used to install the Payment Manager. This installs the Payment Manager Framework and the Test cassette.

RSTLICPGM LICPGM(5733PY2) DEV(OPT01)

In addition to installing the Framework, the cassette for CyberCash needed to be installed. The cassette is installed using Option 2.

RSTLICPGM LICPGM(5733PY2) DEV(OPT01) OPTION(2)

During installation, the QPYMADM user profile is created. This profile is used as the default Payment Manager Administrator for all instances. A password must be associated with this profile before it can be used. The Change User Profile (CHGUSRPRF) CL command is used to change the password. Once the password is changed, this profile can be used to initially log onto the Payment Manager user interface.

Creating a Payment Manager instance and adding the cassette

Now that the Framework and the cassette have been installed, a Payment Manager instance needs to be created. To create an instance, you must first ensure that the HTTP administration server is started and also that the WebSphere Application Server has been started. To start the administration server, use the Start TCP/IP Server (STRTCPSVR) CL command:

STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)

The WebSphere Application Server can be started with the Start Subsystem (STRSBS) command:

STRSBS SBSD(QEJB/QEJBSBS)

To create the Payment Manager instance, access the AS/400 Tasks web page at http://<<hostname>>:2001/ and select the Payment Manager icon.

Once the instance has been created, you must add the cassette to the Payment Manager instance. This is accomplished by selecting the instance from the drop-down box and selecting Work with Cassettes.

Configuring and administering Payment Manager

After all installations are complete, the Payment Manager can be started by using the Start Payment Manager (STRPYMMGR) CL command.

After we installed and configured the Payment Manager Framework, Payment Manager instance, and the CyberCash cassette, we took the following steps to achieve a fully functioning Payment Manager:

1. Define Payment Manager users

The Payment Manager uses the WebSphere operating system user registry, which is composed of Operating System/400 (OS/400) users. To define Payment Manager users, users must have a valid OS/400 user profile prior to being assigned roles. The Create User Profile (CRTUSRPRF) CL command was used to create the appropriate OS/400 user profiles for this scenario.

2. Start the Payment Manager user interface

   After the OS/400 user profiles were created, the Payment Manager user interface was started by using the web site http://<hostname>>/PaymetManager/ and logging in as the default Payment Manager administrator, QPYMADM.

3. Create Merchant/Authorize Payment cassette

   The first step in configuring the Payment Manager was to create a merchant and authorize that merchant to use the CyberCash cassette.

4. Assign user roles

   Having created a merchant and user profiles previously, the next step was to assign user roles. In this scenario a user was assigned as a merchant administrator, and we also assigned a user as a clerk. The clerk role was selected because we needed a user who could perform all payment processing options for the merchant (except credit and credit reversal), and the clerk fit this role.

5. Create an account and a brand

   The first task of the merchant administrator was to establish an account for the CyberCash cassette. An **account** is a relationship between the merchant and the financial institution that processes transactions for that merchant. **To create an account**, we first logged in as the merchant administrator. The account was created by selecting the Merchant Cassette Settings.

   After adding an account, a brand must be created for the account. A **brand** is a credit card type, such as VISA. **To create a brand**, we accessed the Merchant Cassette Settings options, selected the CyberCash cassette, selected the account just created, and selected Brands.

Now that the Payment Manager and merchant administration tasks are complete, approving orders, depositing payments, settling batches, issuing credits, and viewing daily batch totals may begin.

To exercise the CyberCash functions in the Payment Manager, we used an internal IBM tool. This tool allowed us to validate the payment processing application. The CyberCash Cash

Register could also be used in test mode during the setup and testing for a payment processing application.

Payment Manager sources

The following documents are available after installation of Payment Manager 2.1 and can be accessed from the Payment Manager Tasks web page, accessible from the AS/400 Tasks page at **http://*system-name*:2001** where the *system-name* is the TCP/IP host name of the AS/400. The link name in the navigation frame is **Documentation**.

- *IBM WebSphere Payment Manager Administrator's and User's Guide*, Version 2.1, provides information regarding installing, configuring, and maintaining the IBM WebSphere Payment Manager on AS/400. The "For More Information" topic contains additional, related web sites and documents.

- *IBM WebSphere Payment Manager Programmer's Guide and Reference for AS/400*, Version 2.1, provides details about the Payment and Administration API.

- Payment Manager Read Me (English only).

- *IBM WebSphere Payment Manager Cassette for SET Supplement for AS/400*, Version 2.1, provides information about using the SET protocol with the Payment Manager Framework to process electronic payments.

- SET Cassette Read Me (English only).

- *IBM WebSphere Payment Manager Cassette for CyberCash Supplement for AS/400*, Version 2.1, provides information about using the CyberCash protocol with the Payment Manager Framework to process electronic payments.

- CyberCash Cassette Read Me (English only).

- Client API Library (CAL).

Payment transactions

The INSCO application uses the Payment Manager Accept_Payment() method specifying auto approve and auto deposit. The Payment Manager indicates success or failure of the payment transaction. If the payment transaction fails in any way, an appropriate message is sent to customers indicating that their credit card payment was not accepted. Possible failures include unknown card number, invalid expiration, insufficient funds, Payment Manager ended, and Gateway not communicating.

To accept payments over the web, the insurance company must register itself as a merchant with a public payment gateway. This gateway will allow the insurance company to request approval of

credit card charges. It also allows the company to set up an account such that approved credit card payments can be deposited directly into this account.

Numerous payment gateways are available that support a variety of payment protocols. The insurance company uses the Cybercash protocol due to its general acceptance in the industry.

For information regarding financial institutions that have established payment gateways to handle payment processing needs by using the Payment Manager, refer to the web site:

INSCO application details

The INSCO application, designed for the purposes of this report, can improve customer service and reduce costs by allowing customers to view and update their personal information, view policy information, and pay their semiannual insurance premiums.

INSCO is now expanding its web application to include agent service capabilities. These services will allow agents to access these services from both the Internet and INSCO intranet. The services provided to agents will allow them to view policy information for the customers that they support, view and update customer's personal information, submit new insurance applications, and create an account for a new customer. An account administrator working for INSCO will be able to use these new services to work with all customers.

In many cases, web applications such as this can be easily implemented by extending a company's existing applications and data for the web. For our application, INSCO had existing enterprise data residing in a DB2 UDB for AS/400 database. We created three servlets that extend the HTTPServlet class. This class has specialized methods for handling HTML forms, which allow for easy transfer of data from the HTML form to a servlet for processing. Since our application relies on HTML forms to receive input from the user, this was the logical choice.

> **Note:** See the "Application source code" section of the "Appendix" for instructions on downloading the source code and JSP files for the INSCO application.

The INSCO application can be broken down into four sections:

- The **InscoLoginServlet** is responsible for authenticating the user.
- The **Insco2Servlet** is responsible for automating the customer and agent business processes for the insurance company.

- The **PaymentServlet** is responsible for retrieving credit card information from the customer and processing the payment with the Payment Manager.

- The **JavaBeans** component is used to encapsulate user, policy, and coverage data. These beans are used to share data between the servlet methods and the JSP files.

InscoLoginServlet

The insurance company's login page, inscologin.html, is actually an HTML form. It is served via the HTTP protocol. On this page, users are asked to enter their user ID (a customer number or an agent number) and password. These values will then be sent via HTTPS to the doPost() method in the InscoLoginServlet on the web server.

The InscoLoginServlet inherits directly from the AbstractLoginServlet and is a replica of the CustomLoginServlet. However, to support session tracking, a few lines were added to the postLogin() method. The CustomLoginServlet and the AbstractLoginServlet are provided in the servlets directory of the WebSphere Application Server.

Insco2Servlet

The Insco2Servlet allows customers and agents to access business processes via the web. The Insco2Servlet runs on the transactional web servers. This part of the INSCO application consists of one servlet, Insco2Servlet, and the following JSP files: Home.jsp, InscoError.jsp, InscoLogoff.jsp, InscoWarning.jsp, NewCustomer.jsp, RequestSubmit.jsp, SearchResults.jsp, SelectCustomerInformation.jsp, SubmitAutoApp.jsp, SubmitHomeApp.jsp, UpdateInfo.jsp, ViewAutoPolicy.jsp, and ViewHomePolicy.jsp. All of the JSP files were created using the JavaServer Pages 1.0 specification.

The Insco2Servlet contains the following methods: doPost(), doGet(), init(), logoff(), home(), selectCustomerInformation(), selectInfo(), newCustomer(), addCustomer(), newAutoApp(), newHomeApp(), addPolicy(), padNumber(), populateDataQueue(), getCustomerInfo(), checkAgentAccess(), updatePersonalInfo(), submitPersonalInfo(), viewPolicy(), and errorHandle().

PaymentServlet

The PaymentServlet allows customers to submit a secure electronic credit card payment on their insurance policies. The PaymentServlet runs on the transactional web servers alongside the Insco2Servlet. This part of the INSCO application consists of one servlet, PaymentServlet, and the following JSP files: PaymentCreditCard.jsp, PaymentSuccess.jsp, and PaymentError.jsp. It contains four methods: doPost(), doPayment(), doCreditCard(), and init().

JavaBeans component

The beans used by the Insco2Servlet and the PaymentServlet are the UserBean, PolicyBean, AutoBean, HomeBean, and CoverageBean.

The UserBean is used by the Insco2Servlet to store information regarding the current user. If a customer is the current user, the bean will be used to store that customer's personal information. If an agent is logged into the INSCO servlet, the UserBean is used to store the personal information of the customer with which the agent is working.

The PolicyBean is used by the Insco2Servlet to store the base information about a policy. The AutoBean and HomeBean inherit directly from the PolicyBean and are used to store policy information for auto and home insurance, respectively. The CoverageBean is used to store the coverage information for either an auto insurance policy or home insurance policy.

INSCO application flow

In both the Insco2Servlet and the PaymentServlet, the init() method reads a data file called login.properties. This data file contains information about the database it will be connecting to, the data source that it will be using, the LDAP server URL, the system where the data queues are located, the LDAP suffix, and the user ID and password needed to connect to the database and the data queue.

The second part of the init() method establishes a connection to the database using the data source implementation. The setup of the data source was performed on the WebSphere Application Server. Detailed information on connection pooling is available on the web at:

http://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

The doPost() method, from both servlets, retrieves the session values and then routes each request to the appropriate method within the servlet. The supporting method then handles the request and responds to the appropriate JSP file. The doPost() method is used due to security concerns.

The ErrorHandle.jsp page is used to handle any exceptions encountered by the servlets and provide a friendly message to the user on the error encountered.

The Log Off button is available from the Customer Home Page, Agent Home Page, or Select Customer Information. This option is available because it is essential to security that the user logs off to end the session. The servlet is invoked to call the Log Off page, InscoLogoff.jsp, which displays a personalized good-bye message to the user through the use of the session data. The session is then invalidated, which means the session's HttpSession object and its data values are removed from the WebSphere Application Server. If the user fails to issue a log off, the session remains active for 30 minutes from the last time the user sent a request to the web server.

The flow of the main operations of the INSCO application are discussed in the following sections: Login, Home, Update personal information, View policy information, Submit payment information, Select customer information, Create new customer, and Submit insurance application.

Login

The login page provides the means by which customers and agents log onto the INSCO web site. Figure 9 illustrates logging onto the INSCO application. Each numbered transition is described in the steps that follow.

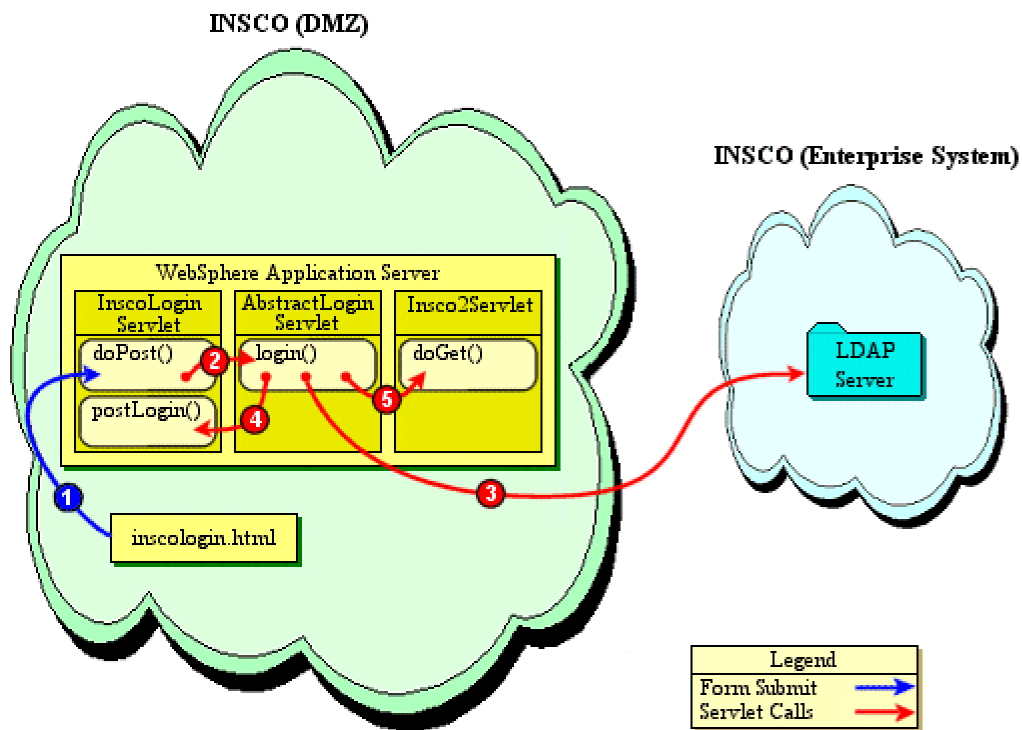


Figure 9. Insco login flow

1. The user either loads the inscologin.html page or is directed there by trying to load the Insco2Servlet. Once users have filled in their user ID and password and selected the Login button, the doPost() method is invoked in the InscoLoginServlet.

2. At this point, the doPost() method obtains the user ID, password, and redirect URL from the form. It then calls the login() method located in the AbstractLoginServlet.

3. The AbstractLoginServlet uses WebSphere Application Server's custom authentication and LTPA to authenticate the user against an LDAP directory via the SSOAuthenticator. The user's information is verified on the LDAP server residing in the INSCO intranet. If LDAP is unable to validate the login information, an exception is thrown and an error

page is returned.

4. Once the user has been authenticated, the login() method calls the postLogin() method located in the InscoLoginServlet. The postLogin() method sets the user ID and password as session values. Once the session values are set, control is returned to the login() method.

5. Finally, the login() method redirects the request to the Insco2Servlet's doGet() method via the redirectURL. A redirect request only routes to a doGet() method. The doGet() method retrieves the session values and then binds to the LDAP server to obtain the user type. It then sets the homebutton value to Home or Select Customer Information, based on user type, and calls the doPost() method.

Home

The main web page provided to both customers and agents is the home page. The home page provides the means by which customers navigate through the INSCO web site and agents work with a specific customer. Figure 10 illustrates the application flow when loading the home page. Each numbered transition is described in the steps that follow.

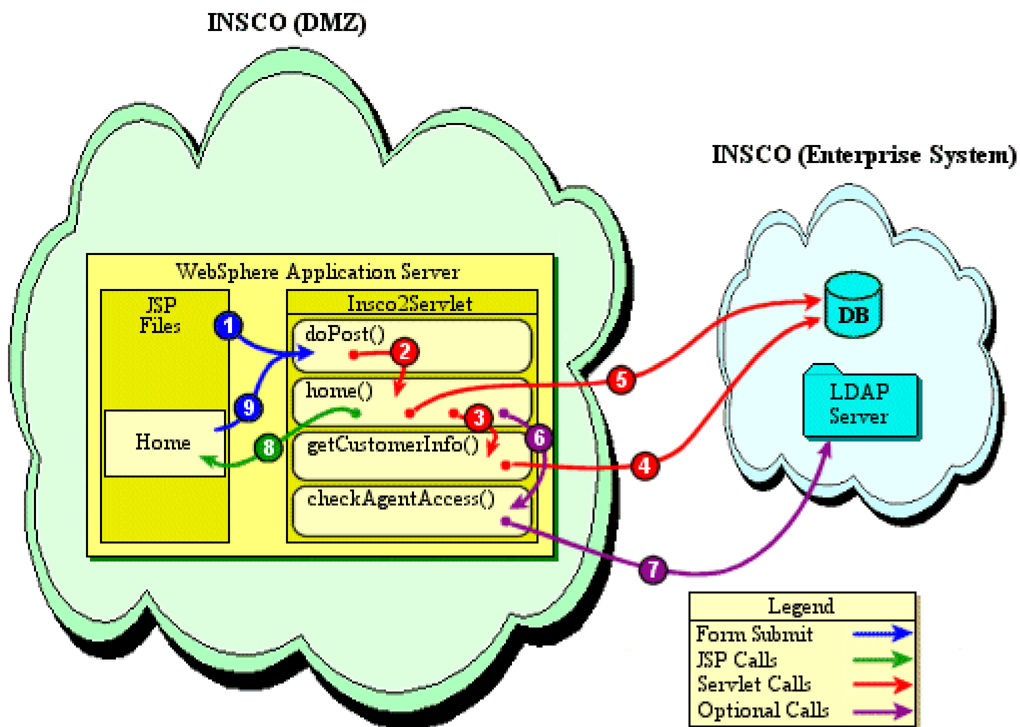


Figure 10. INSCO home flow

1. When the user clicks a button causing their respective home page to load, the servlet is invoked and the doPost() method is called.

2. The doPost() method calls the home() method.

3. The home() method then calls the getCustomerInfo() method to obtain the customer's personal information.

4. The getCustomerInfo() method obtains a data source object. A data source object is established by obtaining one of the existing database connections from the pool. Once the data source object is established, a query against the database is executed to return the customer's personal information based on the customer number. The customer's information includes first name, last name, middle initial, address, zip code, e-mail address, and home and work phone numbers. The data is retrieved from the database, parsed and stored in a UserBean object, and the data source object is returned to the pool. The customer's name is stored in the session data, and the UserBean object is returned to the home() method.

5. The home() method stores the UserBean object in the request object. It then obtains a data source object and performs a query against the database to return the policy information of all policies owned by the customer. Returned along with a policy number is the policy type, status, payment due date, and payment amount. The data is then stored in a vector of PolicyBean objects.

6. In the case of agents, the checkAgentAccess() method is called to determine which policies they are authorized to access.

7. The checkAgentAccess() method loops through the vector of policies performing an LDAP search on each policy. To perform an LDAP search, the servlet first sets the environment properties for LDAP access. Once the properties have been set, the servlet binds to the directory and performs the necessary search on each policy. If the agent has access, `Yes` is stored in the Access field of that policy's PolicyBean object (the default is `No`). After all the policies have been processed, control is returned to the home() method.

8. The vector of policy beans is stored in the UserBean object, and the request is forwarded to the Home.jsp page.

9. From the home page, a user has several options. All users can update personal information, log off of the application, and view policies. However, agents can only view policies they are authorized to access. A customer can also submit an online payment on one or more policies. An agent can submit home and auto applications and can return to the Select Customer Information page. When users select one of the options available to them, the servlet is invoked and the doPost() method is called to handle the user's selection.

Update personal information

The option to update a customer's personal information is provided to both customers and agents. The Update Personal Information page provides the means to change customer information. Figure 11 illustrates the application flow for updating personal information. Each numbered transition is described in the steps that follow.

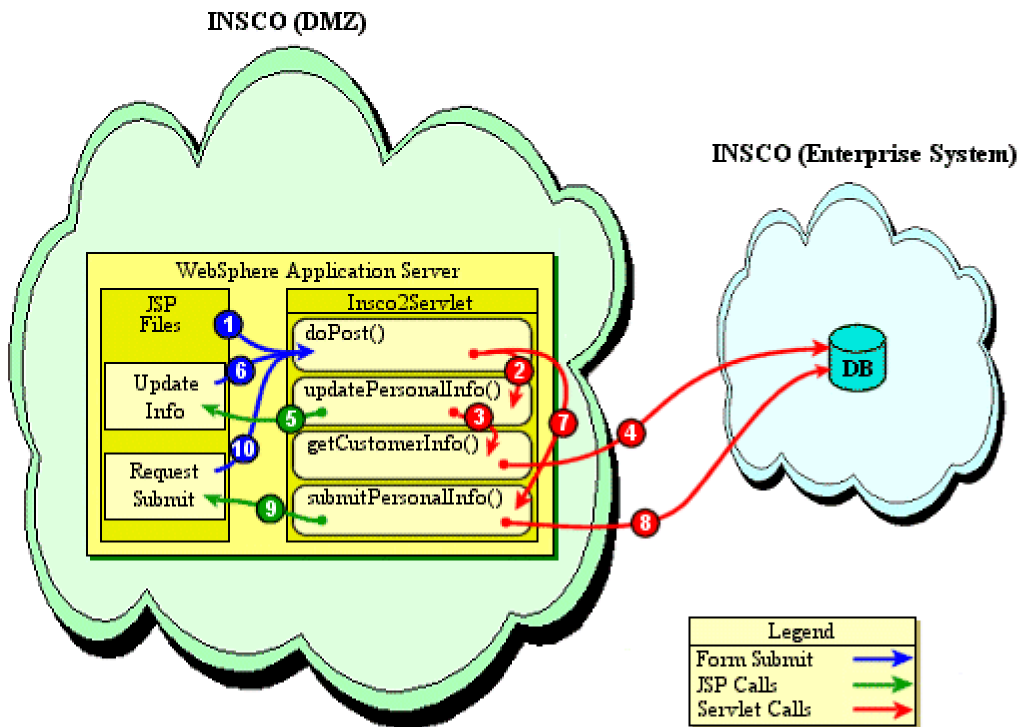


Figure 11. INSCO update personal information flow

1. When users select the Update Personal Information button from their respective home page, the servlet is invoked, and the doPost() method is called.

2. The doPost() method calls the updatePersonalInfo() method.

3. The updatePersonalInfo() method calls the getCustomerInfo() method to obtain the current information for the customer.

4. The getCustomerInfo() method obtains a data source object and queries the database for the customer's information. The data is stored in a UserBean object and returned to the updatePersonalInfo() method.

5. The updatePersonalInfo() method stores the UserBean object in the request object and calls the UpdatePersonalInformation.jsp.

6. The UpdatePersonalInformation.jsp contains the following editable fields: first name, last name, middle initial, address, zip code, e-mail address, home phone number, and work phone number.

   The user has the following choices: make changes on the form and submit them, reset the form to its initial values, or go back to the home page without making any changes. When the user clicks the Submit Changes button, the servlet is invoked and the doPost() method is called.

7. The doPost() method evaluates the homebutton parameter to the submitChanges value and calls the submitPersonalInfo() method.

8. The submitPersonalInfo() method obtains a data source object and updates the customer table.

9. Once the update completes successfully, the RequestSubmit.jsp is called. The RequestSubmit.jsp is used to prevent users from resubmitting the request to update or insert to the database if they press the Reload button on their web browser.

10. Users may click the Back to Home Page button to return to their respective home page, or they may wait five seconds and be redirected there.

View policy information

The option to view policy information is provided to both customers and agents. Customers are allowed to view all of their policies, but agents are only allowed to view a policy if they are authorized to access it. The View Auto (or Home) Policy pages allow the user to view the details of a specific policy. Figure12 illustrates the application flow when a user elects to view a policy's information. Each numbered transition is described in the steps that follow.

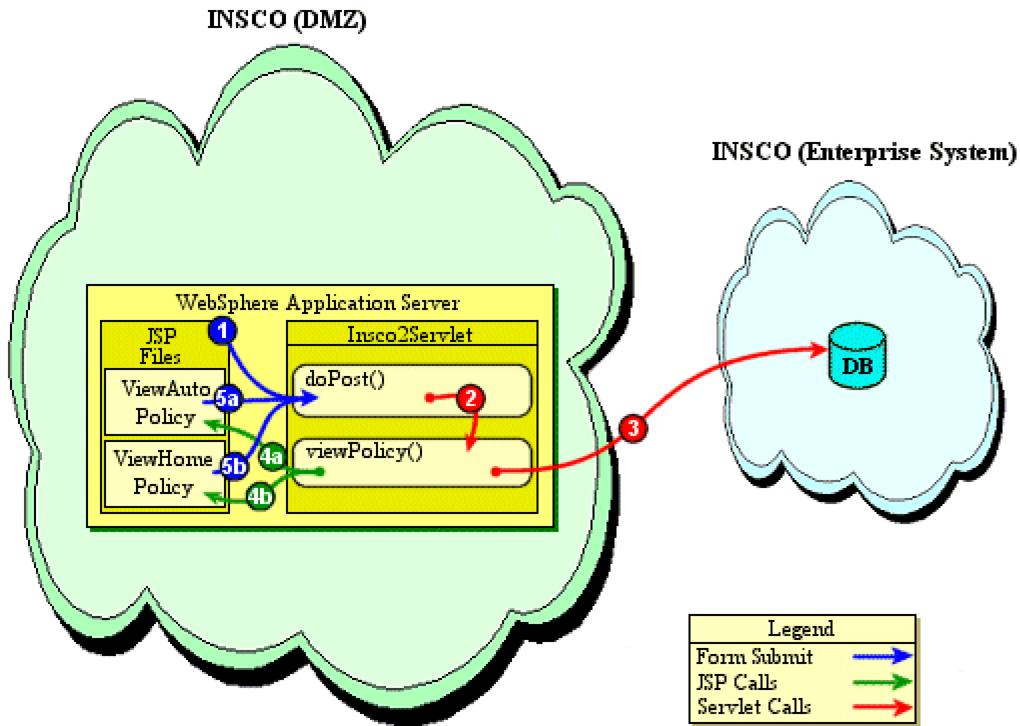


Figure 12. INSCO view policy information flow

1. When users select any View Policy button from their respective home page, the servlet is invoked, and the doPost() method is called.

2. The doPost() method calls the viewPolicy() method.

3. The viewPolicy() method obtains a data source object and then executes a series of queries to return the customer's policy information based on the policy number selected on the home page, the coverage information based on the policy number, and the date of the last payment based on the policy number. The data is retrieved from the database and stored in either a HomeBean or AutoBean object depending on the policy type.

4. At this point, one of two JSP files is loaded based on the type of policy viewed:
   a. If it is an auto policy, the ViewAutoPolicy.jsp page is loaded.
   b. If it is a home policy, the ViewHomePolicy.jsp page is loaded.

5. When users are done viewing the policy details, they click the Back to Home Page button, which invokes the servlet and calls doPost() to handle the request.
   a. From the View Auto Policy page, users are sent back to their respective home page.
   b. The same occurs from the View Home Policy page as in 5a.

Submit payment information

The option to submit payment information is provided to customers. The Submit Payment page allows a customer to pay the premiums on selected policies from the Home Page. Figure 13 illustrates the application flow for submitting payment information. Each numbered transition is described in the steps that follow.

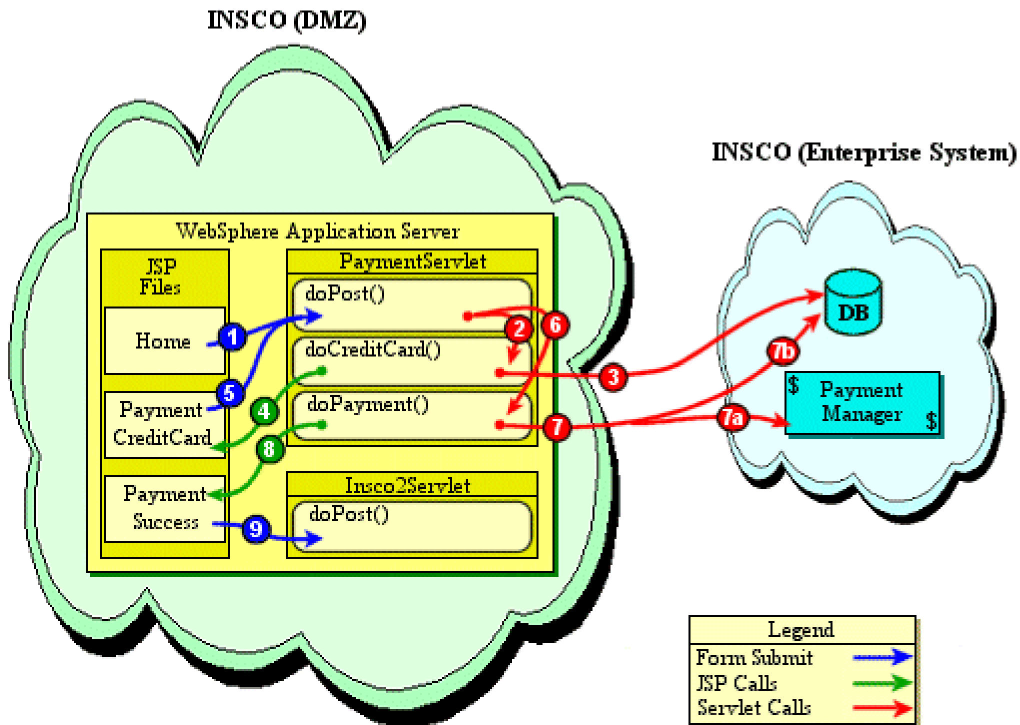


Figure 13. INSCO submit payment flow

1. When the user selects the Submit Payment button from the Customer Home Page, the PaymentServlet is invoked. The PaymentServlet will return control to the home() method once its processing is completed.

2. The doCreditCard() method gets called from the doPost() method if option parameter of the HTML form is set to the getCreditCardNum value. This method is called when the customer clicks Make Payment from the Customer Home Page.

3. The first thing the doCreditCard() method does is query the database for information about each policy. It then creates a vector of PolicyBean objects and stores this vector in a UserBean object. Next, it calls PaymentCreditCard.jsp and passes the newly created UserBean object into the request, so it is accessible from the JSP file.

4. The PaymentCreditCard.jsp displays all the policies to be paid for and a total payment to the browser. It also displays entries to be filled out by the customer about the credit card

being used to make the payment.

5. When the customer clicks the Make Payment button on PaymentCreditCard.jsp, the option parameter of the HTML form is set to the payPremium value and the form is submitted.

6. Next, the doPayment() method of the PaymentServlet is called.

7. The doPayment() method is the method that actually makes a connection to the Payment Manager and sends the payment information. The only Payment Manager API call used in the PaymentServlet is the AcceptPayment command. This command creates an order in the Payment Manager and has the option to automatically approve the order and/or to deposit the payment with a financial institution. In this case, we are automatically approving the order and depositing.

   a. The first thing the doPayment() method does is set up the parameters for the AcceptPayment command. This API call is documented in detail in the *Appendix*, but in general, the following needs to be passed: the payment amount, currency, order number, and whether or not the payment should be automatically approved and/or deposited. After the AcceptPayment command is called, the return code is tested to determine if the payment was a success.

   b. If the payment succeeded, another connection is made to the database to update the payment history table and the next due date.

8. Then, PaymentSuccess.jsp is called. If the payment fails, PaymentError.jsp is called, which tells customers to contact their agent.

9. PaymentSuccess.jsp flashes a success message to the customer and forwards the browser to the Customer Home Page after 5 seconds by submitting a form with the homebutton parameter set to Home.

Select customer information

The option to select customer information is provided to agents. The Select Customer Information page allows an agent to search on a policy number, a customer number, or a customer's last name. Figure 14 illustrates the application flow for selecting customer information. Each numbered transition is described in the steps that follow.

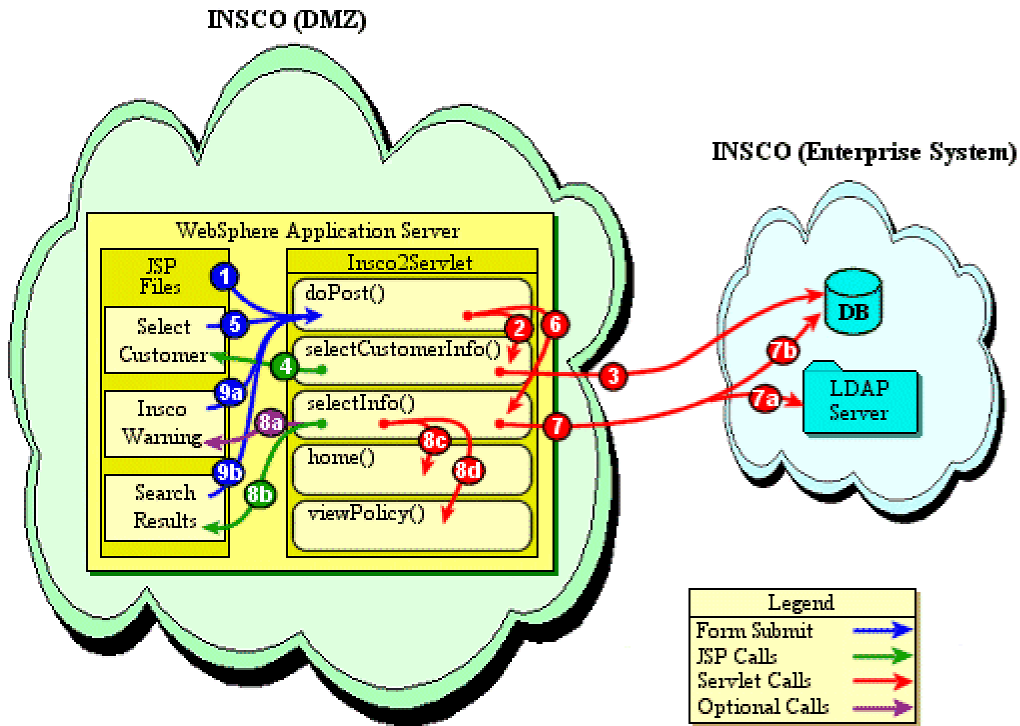


Figure 14. INSCO select customer information flow

1. Once the user has been authenticated and has been determined to be an agent, the doGet() method calls the doPost() method.

2. The doPost() method then calls the selectCustomerInformation() method.

3. The selectCustomerInformation() method obtains a data source object, and a query against the database is executed to return the agent's personal information based on the agent number. This information is stored in the session data. Then the selectCustomerInformation() method calls the SelectInformation.jsp page to welcome the agent.

4. The SelectInformation.jsp gives agents a choice to search for the following: a policy based on a policy number, a customer based on the customer number, or a customer based on the last name. The agent could also press the New Customer button (see Figure 15, INSCO create new customer flow), or log off.

5. The agent's choice is passed back to the servlet via the doPost() method.

6. The servlet's doPost() method routes the request to the selectInfo() method.

7. The selectInfo() method checks each of the possible parameters to determine what the agent was trying to search on, and takes the appropriate action.

a. When an agent selects to search on a policy, an LDAP request is opened and the policy in question is searched for. If the policy is not found, it could mean that the policy does not exist or that the agent does not have access to the policy. In either case, a message is displayed to the agent to explain this. Otherwise, if the record was found in LDAP, then the viewPolicy() method is called (see 8d in Figure 14).

b. When the agent selects to search by customer number, then the selectInfo() method requests a database connection to see if that customer exists in the database. If the customer does exist, then the customer's name and number are placed in the session, and the home() method is called (see 8c in Figure 14).

   When the agent chooses to search for customers by their last name, the selectInfo() method requests a database connection to see if any customers exist in the database with the last name that the agent supplied. For 7b in Figure 14, a count is taken on the number of customers that match.

8. If the count of the customers (from step 7) is greater than 20, then the selectInfo() method calls the InscoWarning.jsp (see 8a in Figure 14); otherwise, it calls SearchResults.jsp (see 8b in Figure 14).

   a. InscoWarning.jsp is displayed when the number of customers returned in the SQL result set is greater than 20.

   b. SearchResults.jsp is used to display all of the customers that were returned from the search.

   c. The home() method is called when the agent selects to search by a customer number that exists.

   d. The viewPolicy() method is called when the agent selects to search on a policy and has access to the policy.

9. If more than 20 customers are returned in the result, then the InscoWarning.jsp is displayed; otherwise, the SearchResults.jsp is displayed.

   a. A warning page is displayed to agents, telling them how many customers matched the search and asking them if they would like to continue. Agents have the choice of canceling the search or continuing. If they want to continue, then when they submit the HTML form, an appropriate parameter is passed to the servlet to call the SearchResults.jsp page.

   b. The SearchResults.jsp page contains a table listing all of the customers that match the search by last name. Next to each name in the table is a button to view that customer.

When the agent presses one of these buttons, then the servlet's doPost() method is invoked once again, and this time calls the home() method.

Create new customer

The option to add a new customer to the insurance company's database is provided to agents on the New Customer page. Figure 15 illustrates the application flow for creating a new customer. Each numbered transition is described in the steps that follow.

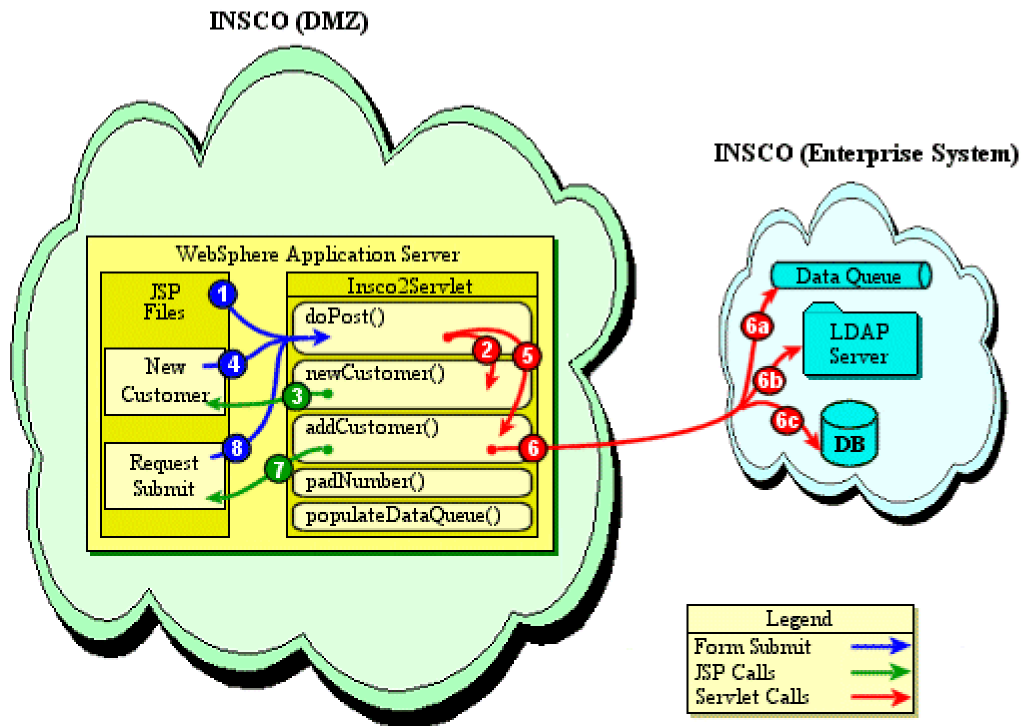


Figure 15. INSCO create new customer flow

1. When an agent selects the New Customer button from the Select Customer Information page, the servlet is invoked, and the doPost() method is called.

2. The doPost() method calls the newCustomer() method.

3. The newCustomer() method calls the NewCustomer.jsp page. The NewCustomer JSP file contains the following editable fields: first name, middle initial, last name, password, home address, zip code, e-mail address, phone numbers, social security number, date of birth, marital status, occupation, and household income. When the form is completed, the agent has the choice of pressing the Add New Customer button or the Cancel button. If the agent chooses to cancel the request, the agent is returned to the Select Customer Information page.

4. When the user presses the Add New Customer button, the form is submitted to Insco2Servlet's doPost() method.

5. The doPost() method determines that a new customer form has been submitted and calls the addCustomer() method to handle this request.

6. The addCustomer() method performs the following operations:

   a. It first obtains a new customer number through the use of an AS/400 data queue. The servlet uses the AS/400 Toolbox for Java access classes to get a reference to the data queue, and then tries to read from it. If the data queue does not exist, it is created. If it is empty, then a call is made to the populateDataQueue() method. The populateDataQueue() method makes a call to the database to get the largest customer number from the customer table, adds one to this number, and eventually returns this number to the calling method. But, before it returns, it pushes 20 new customer numbers on to the data queue. The numbers that are pushed on to the data queue must first be formatted so that they have the correct number of preceding zeros. The formatting is accomplished by making a call to the padNumber() method.

   b. An LDAP add operation is then performed to add the customer to the LDAP directory.

   c. If the customer is added successfully, the servlet requests a data source object and inserts the customer information into the database.

7. If the previous step completes successfully, the RequestSubmit.jsp is loaded informing agents that their information has been submitted.

8. The agent can then press the Back to Home button, or after 5 seconds, the RequestSubmit.jsp calls the doPost() method, which calls the home() method.

Submit insurance application

The option to submit either a home policy or auto policy insurance application is provided to agents. The Submit Home Application or Submit Auto Application page allows an agent to add a home or auto insurance application to the insurance company's database. Figure 16 illustrates the application flow for submitting an insurance application. Each numbered transition is described in the steps that follow.

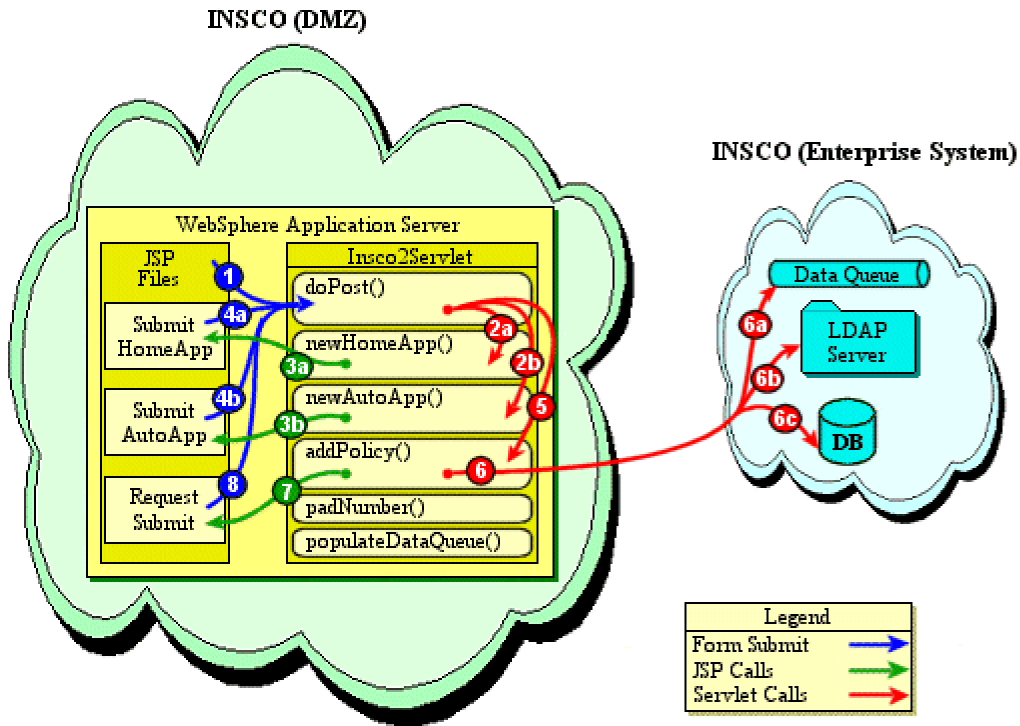


Figure 16. INSCO submit insurance application flow

1. When an agent selects the Submit Auto Application or Submit Home Application button from the Agent Home Page, the servlet is invoked and the doPost() method is called.

2. The doPost() method calls either:

   a. The newHomeApp() method.
   b. The newAutoApp() method.

3. The method called from the previous step displays the JSP page to submit the appropriate policy application form. This form contains all of the editable fields needed to submit an auto or home insurance application.

   a. The newHomeApp() method calls the SubmitHomeApp.jsp file.
   b. The newAutoApp() method calls the SubmitAutoApp.jsp file.

4. Both of the JSP pages work in the same manner. The primary difference between the files is the type of input that is requested from the user and the data that is displayed to the agent. When agents have completed the form, they have the choice to submit the application, add a coverage level, or cancel the operation. If the agent chooses to cancel the request, the agent is returned to the Agent Home Page. If the Add Coverage button is clicked, then the agent is taken back to step 2, and all of the policy information entered by the agent is passed along in the request to the servlet. The servlet then passes this

information back to the JSP page so that the agent does not have to re-enter any information.

a. The SubmitAutoApp.jsp
b. The SubmitHomeApp.jsp

After all desired coverages have been added to the policy, the agent clicks the Submit Auto Application button or the Submit Home Application button. The appropriate JSP file invokes the servlet and the doPost() method is called.

5. The doPost() method calls the addPolicy() method.

6. The addPolicy() method has several steps that it must go through to submit and add the policy to the INSCO environment:

   a. It obtains a new policy number through the use of a data queue.

   b. It then opens an LDAP request to add the policy to the LDAP directory along with the proper customer and agent access authorities.

   c. Once the policy has been added to the LDAP directory, the servlet obtains a data source object, and database inserts are performed into the appropriate tables. After the insert has completed on either the AUTO or HOME table, a database trigger is activated. This trigger submits the policy application to the underwriter for approval. The details of this trigger program are described later in the "INSCO B2B application" section.

7. Once the update has completed successfully, the RequestSubmit.jsp is loaded to give a visual confirmation to the agent.

8. The RequestSubmit.jsp redirects to the doPost() method, which calls the home() method.

INSCO servlet development discoveries

The following discoveries and recommendations are based on our experience coding the INSCO application:

- When using any Payment Manager API call, the API classes need to be imported into the servlet. The API classes, etillCal.zi and eTillxml4j209.jar, need to be copied from the enterprise server to both transactional web servers before compiling the servlet. These API classes also need to be added to the WebSphere Application Server classpath for the web application being used.

- The way beans were instantiated needed to be changed from ab = (ABean) Beans.instantiate(null, "ABean") as used in the scenario 1 servlet to ab = (ABean)

Beans.instantiate(getClass().getClassLoader(), "ABean"). The reason for this change is that if you use null for the first parameter, then WebSphere uses the system class loader and cannot find the ABean class.

- Migration from the Connection Manager to data source is recommended since the APIs for Connection Manager may not be available in releases beyond WebSphere Application Server 3.x Advanced Edition. Detailed information on migrating from Connection Manager is available on the web at:

  http://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

  From this page, select the **Servlets** tab, select **Migrating**, and then select **Connection Manager**.

- Create a data source to implement the new connection pooling. Detailed information on setting up a data source is available on the web at:

  http://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

  From this page, select the **Home** tab, select **Search**, enter `data source` and press **Enter**. Select **Creating a data source from the WebSphere Administrative Console**.

- In scenario 1, the servlet used the following code from com.sun.server.http.* to call JSP pages.

  ((com.sun.server.http.HttpServiceResponse) ares).callPage("/inscohome.jsp", areq);

  WebSphere Application Server 3.02 has deprecated this interface from WebSphere Application Server 2.0. However, to ease migration, they have re-implemented Sun's class for compatibility purposes. One solution is to recompile your source code with the WebSphere Application Server 3.02 version ibmwebas.jar file in your classpath. Another way to call a JSP page is to replace the callPage() call with the following:

  RequestDispatcher rd = getServletContext().getRequestDispatcher("/inscohome.jsp");
  rd.forward(areq, ares);

- The WebSphere servlet engine can handle most cases of session management. However, sessions can still become corrupted. The JavaServer Pages 1.0 specification addresses the possibility that sessions can break programatically. For example:

  1. WebSphere Application Server assigns a session to an instance variable in a servlet and then allows that assigned session to be modified by other threads that may run the servlet.
  2. A servlet spawns multiple threads and gives each thread access to the session.

The WebSphere documentation discusses sessions as well and is available on the web at:

http://www.as400.ibm.com/products/websphere/docs/as400v302/docs/index.html

From this page, click the **Servlets** tab and click **Developing** in the navigation bar. After the navigation bar expands, click **Servlet APIs**, and then click **Sessions**.

Here is a discussion from that section that applies to session management:

> In all cases, WebSphere Application Server defines the notion of a session transaction, which begins when the servlet calls HttpServletRequest.getSession() and ends with the completion of the servlet's service() method. By default, WebSphere Application Server locks sessions during the scope of these transactions to maintain session integrity. This means that one (and only one) instance of a servlet can access a session at a given time. In the case where several servlets are chained together to service an individual HTTP request, the session stays locked across each servlet in the chain until a response is finally sent back to the user.

- The following ports need to be opened on the domain firewall: port 389 for LDAP; ports 449, 8471, 8472, and 8476 for AS/400 Toolbox for Java, JDBC driver, and data queue usage; and port 446 for the AS/400 Developer Kit for Java.

- The CustomLoginServlet provided with the WebSphere Application Server only redirects to a doGet() method.

B2B applications

The Underwiter and INSCO B2B applications were deployed on the enterprise servers in the INSCO and the underwriter intranets. The two servers, both with IBM HTTP Server and IBM WebSphere Application Server installed, were equipped with each company's portion of the application. The applications communicated with each other over the VPN via XML messages using HTTP.

The goal of these applications is to allow the underwriter to approve any new policy applications submitted by the INSCO company. After an insurance application is added to the database residing in the INSCO intranet, the policy is submitted to the underwriter via an XML message. Once underwriters receive the insurance application, they make an assessment of the insurability of the applicant. Some policies are automatically approved while others must be approved manually. Once the underwriter approves or rejects the policy, an XML response is sent to INSCO informing it of the policy status. After receiving the response, INSCO updates the status of the policy in the database and sends an e-mail to the appropriate customer and agent informing them of the status change.

B2B server setup

For the B2B applications, WebSphere Application Server 3.02 Advanced Edition needed to be set up on both the enterprise server and the underwriter server. The setup was identical to the setup of the transactional web servers, except that security did not need to be set up on the two systems because communication between them was through a VPN. Data sources and web applications were set up similar to how they were set up on the transactional web servers.

Setting up e-mail

E-mail is probably the most widely used Internet application. It provides a convenient and inexpensive way to communicate across the world. This scenario introduced some automated features to INSCO. E-mail was chosen as a means to confirm transactions handled by the automated processes.

Looking at the run-time topology diagram (Figure 17), we see that in order for mail to be delivered from INSCO to the Internet, it must flow through the domain and protocol firewalls.

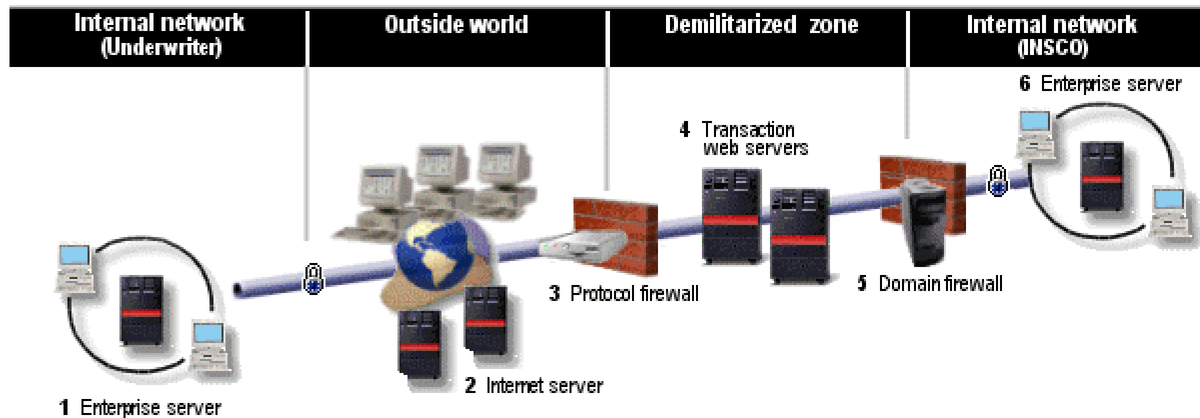


Figure 17: Run-time topology

The domain firewall is running a secure mail proxy server. The domain firewall's job is to send e-mail from hosts inside the firewall to hosts outside the firewall. Filter rules were added to the domain and protocol firewalls to permit mail traffic.

The Simple Mail Transfer Protocol (SMTP) provides both send and receive functions that allow you to send or receive e-mail. When we configured the AS/400 SMTP server for INSCO's internal network, two parameters needed to be set so that all outgoing mail was forwarded to the domain firewall.

These parameters can be easily set using Operations Navigator. From Operations Navigator, go to the **General** page of the **SMTP server properties**. We specified a mail router (INSCO firewall) and checked **Forward outgoing mail to router through firewall**.

The Domain Name System (DNS) server plays an important role in the delivery of e-mail. The local mail server determines an Internet address for each e-mail recipient and asks Domain Name System (DNS) servers to assist in determining these addresses. SMTP determines an address by first requesting the address of the mail exchanger (MX) for the recipient. A **mail exchanger** is a server program that is in charge of delivering e-mail to a set of hosts. Not all recipients have mail exchangers. The answer from the domain name server may indicate that there is no mail exchanger resource data for the requested recipient's host. If this happens, SMTP sends the query again to the domain name server. SMTP asks for the Internet address of the recipient's host so that the system can send the e-mail directly to that host.

A Post Office Protocol (POP) client was used to retrieve e-mail. POP is an electronic mail protocol with both client (sender/receiver) and server (storage) functions. POP allows mail for multiple users to be stored in a central location until a request for delivery is made by an electronic mail program. The AS/400 POP server was configured and used.

Additional information on setting up AS/400 e-mail can found at the following web site for the AS/400 Information Center.

http://www.as400.ibm.com/infocenter

Within the AS/400 Information Center, select **TCP/IP,** then **Sending and receiving e-mail**.

B2B application details

The B2B applications can be broken down into three main sections: the Underwriter B2B application, the INSCO B2B application, and the XML communication between the two applications.

XML communication

XML documents are sent in the body of an HTTP request to and from the INSCO and underwriter servers over the VPN. These XML documents follow the ACORD DTDs for personal-home and personal-auto insurance-application requests. These DTDs define the information and semantics that must be sent in a policy application request and response.

The request XML document will be created from the policy and customer information stored in the database on INSCO's enterprise server. INSCO used XML Lightweight Extractor (XLE) to map the data in the database to tags in the DTDs and to generate the request XML document. The request XML document is sent to the underwriter in the body of an HTTP request and then parsed with IBM's XML4J parser to extract the needed data. The underwriter also uses the LotusXSL product, shipped with WebSphere Application Server 3.02 Advanced Edition, to display the XML document. When the underwriter needs to send a response back to INSCO, it generates the response XML document, which is much shorter and simpler than the request, and sends it to INSCO in the body of an HTTP request. INSCO then uses the XML4J parser to extract the policies status and processes accordingly.

The four DTDs (PersAutoAddRq.dtd, PersAutoAddRs.dtd, PersHomeAddRq.dtd, and PersHomeAddRs.dtd) were copied to both the enterprise server and the underwriter server because they were needed to validate the XML documents. To make the communication more robust, we validated each XML document with the appropriate DTD after it was generated and after it was received. XML validation means that the XML document is checked to make sure it follows the rules and semantics set up in the DTD.

For more information on the ACORD DTDs, refer to the ACORD web site:

http://www.acord.org

Note: See the "Appendix" for an example of each XML document we used.

INSCO B2B application

Figure 18 depicts the INSCO B2B application, which consists of the following components: the POLICYTRIG trigger program, the autoPolicy.dtdsa and homePolicy.dtdsa XLE mapping files, the SubmitApplication Java program, and the UnderwriterResponseServlet servlet.

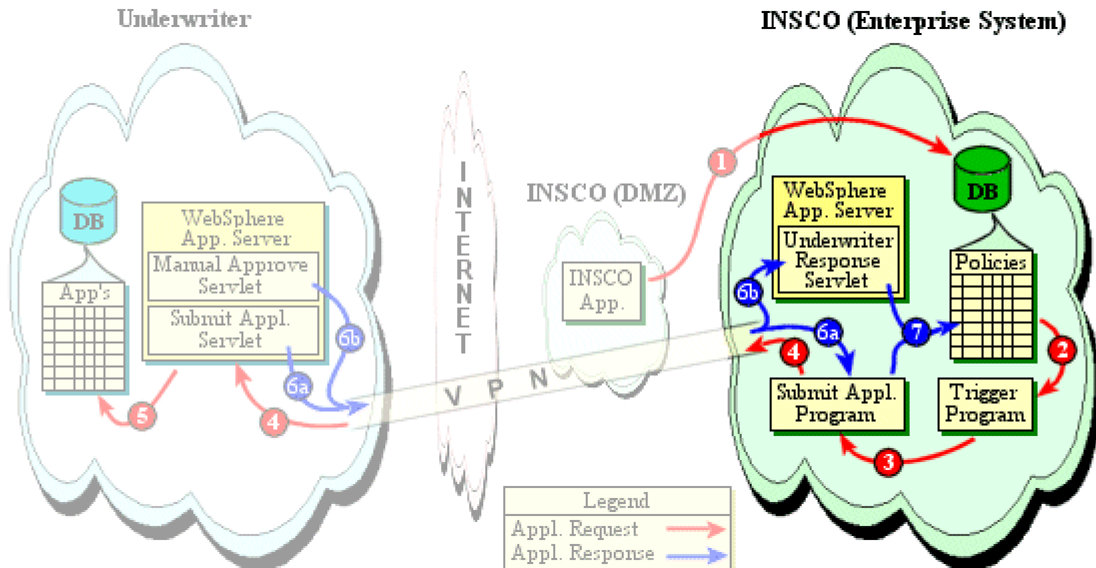


Figure 18. INSCO B2B application

Trigger program

A trigger is a program that is attached to a database file. The trigger executes when a specified event occurs to a specified table.

Detailed information on triggering automatic events is available on the AS/400 Information Center on the web at:

http://www.as400.ibm.com/infocenter

In this scenario, triggers were placed on both the HOME and AUTO tables to occur after an insert operation takes place. The CL trigger program, POLICYTRIG, submits the Java Program, SubmitApplication, to batch.

Note: See the Payment Manager 2.1 API section of the "Appendix" for the POLICYTRIG code snippet.

Trigger program discoveries

Two factors determined why the SubmitApplication program, which was called from the CL trigger program, needed to be submitted to batch.

First, if the SubmitApplication program was not submitted to batch, there was a chance that a deadlock could exist. A deadlock could be created when the SubmitApplication program needed to obtain a lock on the file that activated the trigger. This file was also being used by the insert request, which would be waiting for the trigger to complete before it would release its lock. Eventually, this would cause the SubmitApplication program's update request to time out, and it would not complete successfully. To alleviate this deadlock condition, the SubmitApplication program was submitted to batch. This allowed the program to run in its own process and subsystem while also allowing the trigger to complete. Once the trigger completed, the file lock was released and the SubmitApplication program was able to complete.

Second, if the SubmitApplication program was not submitted to batch, the insert request would wait until the trigger had completed its processing. This would cause a longer delay to the user before the request submitted page would be displayed. Since submitting a policy was already complex and costly for a web application, the best solution was to have the trigger program submit the SubmitApplication program to batch.

SubmitApplication

The SubmitApplication program receives new policy applications and sends them to the underwriter for approval. Then, if the policy is instantly approved or rejected, the program updates the status and sends e-mail to the appropriate people. The SubmitApplication program is called from the trigger program whenever a new policy gets added to the database. It contains the following methods: main(), submitApplications(), sendEmail(), addToPending(), updateDatabase(), validateXmlString(), sendRequest(), generateRequestId(), log(), init(), and readSMTP().

The main() method, which gets called when the SubmitApplication program is invoked, simply calls the submitApplications() method. The init() method is called from the submitApplications() method; it initializes a log file for error reporting and reads a properties file for database connection information. The log() method takes a string as a parameter and writes that string to the log file that is set up in the init() method. The generateRequestId() method creates a new unique request identifier based on the policy number and current time so that requests can be tracked easily.

The submitApplications() method first queries the database for new policies. For each new policy, it calls the XLE extract() method to get the request XML document. It then calls the validateXmlString() method to ensure that XLE generated a valid XML document. Next, it tries

to call sendRequest() to send the request XML document to the underwriter. If sendRequest() fails, it will retry a preset number of times. In our case we set the number of retries to 2.

The sendRequest() method opens up an HTTP connection to the underwriter server. It then prints the request XML document into the body of the HTTP request and sends it. Next, the sendRequest() method checks the header of the HTTP response for errors, setting a return code appropriately if any exist. If there are no errors, it reads the response XML document from the body of the response and calls the validateXmlString() method to ensure the underwriter sent a valid response. Next, it uses the XML4J parser to extract data from the response XML document and verifies that the request identifier matches the response identifier. Finally, the updateDatabase() and sendEmail() methods get called, and the addToPending() method gets called if the status of the policy is pending, meaning that the underwriter was not able to instantly approve or reject the policy.

The updateDatabase() method changes the status of the policy in the database. The addToPending() method inserts a record into the PENDING table of the database with the request identifier as the primary key. The sendEmail() method first queries the database to determine the customer and agent's e-mail addresses; then it opens up a socket to the SMTP server to send a status update e-mail to the customer and agent. The sendEmail() method uses the readSMTP() method to clear the input stream while writing to the SMTP server.

XLE mapping files

INSCO had to create a separate XLE mapping file (also called a DTDSA file, which stands for data type definition with source annotation) for each policy type, home and auto, because each type has its own DTD. These mapping files are similar to DTDs except that they have additional semantics to describe where each tag will get its data. This allows XLE to assemble an XML document from a set of relational database tables.

XLE was a new product during the implementation of this scenario. Some limitations with the XLE documentation made it difficult to map the INSCO database to the ACORD DTDs. The problem was that XLE was designed to map each XML tag to one specific column of one specific table. But in the ACORD DTD, they reuse tags to describe similar data. For example, they have a Name tag which can be used to describe a company name, an agent's name, or a customer's name, each of which is stored in a different table in our scenario. Dates are extremely troublesome because they reuse the *YEAR*, *MONTH*, and *DAY*. To work around most of the problems, some slight changes needed to be made to the ACORD DTD.

The list below summarizes the changes we made to the ACORD DTD:

- Changed the Date tags in the DTDs from multiple subtags to single value tags.
  Old format:

```
<DateOfBirth>
      <YEAR>1999</YEAR>
      <MONTH>09</MONTH>
      <DAY>14</DAY>
</DateOfBirth>
```

New format:

```
<DateOfBirth>1999-09-14</DateOfBirth>
```

This affected the *DateOfBirth*, *EffectiveDate*, and *ExpirationDate* tags. This change was required because of the XLE limitation and because of the way the database returned dates.

- Changed the *HouseholdIncome* and *DeductibleAmount* tags from multiple subtags to single value tags.

  Old format:

  ```
  <HouseholdIncome>
        <Amount>50000</Amount>
  </HouseholdIncome>
  ```

  New format:

  ```
  <HouseholdIncome>50000</HouseholdIncome>
  ```

  This change was needed because the *Amount* tag was already being used as a *CurrentTermAmount* subtag.

- Only used the *Addr1* subtag in the *CustomerAddress* tag and *Addr2* subtag in the *PostalAddress* tag. This was not a change to the ACORD DTD, but a restriction we had to follow because of the XLE limitation.

UnderwriterResponseServlet

The UnderwriterResponseServlet is similar to the SubmitApplication program. It has almost the same code except that it is a servlet instead of a Java application, and it processes the response but does not send a request to the underwriter. The UnderwriterResponseServlet gets called from the underwriter's ManualApproveServlet whenever a policy application needs to be manually approved or rejected. It contains the following methods: doPost(), sendEmail(), readSMTP(), updateDatabase(), validateXmlString(), and init().

As with all servlets, the init() method gets called when the servlet first gets invoked within WebSphere Application Server. It reads database connection properties from a properties file and sets up a data source object for making connections to the database.

The doPost() method is invoked each time the UnderwriterResponseServlet gets called. It reads the response XML document from the HTTP request body and makes sure it is a valid XML document by calling the validateXmlString() method. It then parses out the status information from the XML document with the XML4J parser, similar to the sendRequest() method of the SubmitApplication program. Next, it queries the PENDING table of the database to verify that the response matches a pending request. Finally, if there are no errors, the doPost() method calls both the updateDatabase() and sendEmail() methods.

The updateDatabase(), sendEmail(), and readSMTP() methods are all identical to the methods in the SubmitApplication program.

Underwriter B2B application

Figure 19 depicts the underwriter B2B application, which consists of the SubmitApplicationServlet, the ManualApproveServlet, and the appview.xsl LotusXSL file.

Figure 19. Underwriter B2B application

SubmitApplicationServlet

The SubmitApplicationServlet processes the application request from INSCO's SubmitApplication program and sends a response XML document back to INSCO stating whether the policy was approved, rejected, or pending. It contains the following methods: doPost(), init(), parseXmlDocument(), and validateXmlString().

The init() method is the same as it is in the other servlets in that it sets up the data source object for connecting to the database. The doPost() method does all the work in this servlet. First, the doPost() method reads the request XML document from the HTTP request body and makes sure it is valid by calling the validateXmlString() method. Then, it calls parseXmlDocument(), which uses the XML4J parser to extract certain information from the request XML document, such as policy number and income. Next, the doPost() method saves the request XML document as an XML file in a directory on the AS/400. It will later be read by the ManualApproveServlet for viewing by the underwriter.

The doPost() method then determines whether the policy can be instantly approved or rejected. To keep things simple, we decided to base this decision solely on the income of the customer. If a customer's income is greater than $50,000, we instantly approve the policy; if it is less than $20,000, the policy requires manual approval, and the status is sent back to INSCO as pending. Based on this decision, the doPost() method formulates a response XML document and passes it

back to the SubmitApplication program in the request body. Finally, the doPost() method inserts a record into a database table with the request identifier and status so that the underwriter can easily determine which policy applications have been approved, rejected, or are pending.

ManualApproveServlet

The ManualApproveServlet allows the underwriter to view all the pending, approved, or rejected policy applications; to view specific policy details; and to approve or reject policies that have a pending status. The methods used in this servlet are doPost(), sendResponse(), viewApp(), and init().

The init() method sets up the data source object for connecting to the database. The doPost() method first checks the option parameter of the HTML form. If it is set to the view value, the viewApp() method is called; if it is set to the respond value, the sendResponse() method is called. Otherwise, the doPost() method displays some or all of the policy applications from the database depending on what the option parameter is set to. The underwriter can choose what types (approved, rejected, or pending) of policy applications to display or can choose to display all policy applications. The doPost() method then displays a table of policy applications to the underwriter with data such as status and customer name and a View button for each application. When the View button is clicked, the option parameter is set to the view value and the doPost() method calls the viewApp() method.

The viewApp() method uses LotusXSL style sheets to display the policy application. First, the viewApp() method must read the request XML document that was saved into a directory by SubmitApplicationServlet. It then reads the appview.xsl style sheet for use by the LotusXSL processor, which processes the request XML document and prints the results to the servlet output stream (that is, to the browser).

The sendResponse() method is called when the option parameter is set to the respond value, which is done in the appview.xsl style sheet when the underwriter clicks Approve or Reject. It first reads the status from the HTML form parameters and then formulates the response XML document to send back to INSCO. Then it opens up an HTTP connection to INSCO via the VPN. The response XML document is sent in the body of the HTTP request to INSCO's UnderwriterResponseServlet. If there are no errors in the connection, the APPS table of the database is updated to reflect the status change of the policy application. Finally, a status message is sent to the browser informing the underwriter that the operation was successful or that it failed.

XSL style sheet

The appview.xsl style sheet tells the LotusXSL processor how to convert the request XML document into HTML to be sent to the browser. The style sheet works by telling the processor what HTML to print out for each tag of the XML document. In the appview.xsl style sheet, all the data of the policy application is formatted into HTML so that the underwriter can easily view the application from a browser and approve or reject the policy. Approve and Reject buttons are

also displayed on this page. When either button is clicked, the appropriate parameters, such as the status and the reason for approval or rejection, are passed to the ManualApproveServlet and the sendRequest() method is called.

For more information about the LotusXSL, refer to:

http://www.alphaworks.ibm.com/tech/LotusXSL

Network and security

As businesses are extended into the e-business world through user-to-business and B2B configurations, designing and deploying a security plan becomes increasingly important.

The first task in designing your security plan is to consult your security policy to understand what you need to protect and what level of access you need to provide. As you formulate your plan, you may find your security and access needs working against each other. Greater access needs may mean less security and more security may mean less access. This is where your security policy will determine which wins out--security or function.

Any time you are dealing with a network, especially the Internet, the best approach to take is a layered approach. That is, you implement as many layers of security as you can and as many as your security policy dictates. For example, rather than relying solely on a firewall to protect your network, you implement all the packet filtering your Internet Service Provider (ISP) provides, put additional packet filtering in your router, thoroughly test and keep your firewall configuration up to date, and implement a restrictive security policy on your host AS/400. The layered security approach that INSCO implemented is described below.

INSCO security requirements

The INSCO business plan called for the company to expand its capabilities in the following areas:

- Customer self-service via the Internet
- Agent access via the Internet
- Automated application approval

Let's analyze each of the enhancements in further detail to better understand the security requirements. What do we need to protect? We need to ensure that the communication between the end users (customers and agents) and the web servers is secure. The data that customers and agents are accessing needs to be secured. Customers can see only their policy information, and the agents can see only policy information for customers that they are responsible for. Payment processing requires secure communication. The application collecting payment information needs to communicate securely with the Payment Manager server.

Now let's figure out what level of access we need to provide. The customer and agent access capabilities are provided by custom applications running on servers in INSCO's demilitarized zone (DMZ) network. These applications need to access servers running on an internal system in INSCO's internal network.

Providing an automated approval application required cooperation with the underwriter company. Applications running on servers in the INSCO and underwriter networks communicate with one another to handle the application approval process. Communications

between these applications must be secure. The access provided also needs to be restricted to only what is needed for the applications.

Security technologies

With a basic understanding of what we need to protect and what level of access we need, we can investigate available technologies. There are several technologies available: firewalls, SSL over HTTPS, VPN, and IP packet filtering. You can also incorporate security into your application utilizing security architectures such as the WebSphere distributed security model.

Firewalls have been and remain the anchor point for network security, but a growing number of alternatives continue to become available. Firewalls are neither cheap nor easy to install and maintain, so a market has been created for security devices that provide cost-effective solutions and solutions that are easy to deploy and manage. A number of security functions on servers and routers, as well as the increasing number of security appliances, have created the ability to build more layered security solutions, without adding to the complexity of installation and maintenance.

The SSL protocol consists of two separate protocols, the record protocol and the handshake protocol. The handshake protocol is encapsulated within the record protocol. The **SSL handshake** is used to establish an SSL session on the TCP/IP connection between a client and a server application. The SSL handshake usually occurs immediately after the TCP connection is established. During the handshake, the client and server agree on the encryption algorithms and the encryption keys that they will use for that session. In all SSL handshakes, the client will authenticate and verify the identity of the server. The server can optionally authenticate and verify the identity of the client. After the SSL handshake has successfully completed, information exchanged between the client and server is encrypted using the negotiated keys. An important advantage of SSL is its ability to negotiate unique encryption keys for each SSL session between a client and a server even if they have not previously communicated with each other.

During the SSL handshake, the client and server exchange digital certificates. **Digital certificates** provide identifying information that enables the client and server to identify each other. Digital certificates are issued by trusted third-parties called certificate authorities. An SSL client must trust the certificate authority that issued the server's certificate in order for the SSL handshake to complete successfully.

The SSL protocol engine provides a set of published application programming interfaces (APIs) that are used by socket applications. It uses the services of a cryptographic service provider to perform all cryptographic services.

A **virtual private network** (VPN) is an extension of an enterprise's private intranet across a public network such as the Internet, which creates a secure private connection essentially through a private tunnel. VPNs securely convey information across the Internet that connects remote users, branch offices, and business partners into an extended corporate network. ISPs offer cost-effective access to the Internet (via direct lines or local telephone

numbers), enabling companies to eliminate their current, expensive leased lines, long-distance calls, and toll-free telephone numbers.

VPN implementations come in two forms, endpoint (host) and gateway. An **endpoint** solution is when the VPN implementation exists at the application's initial entry into the TCP/IP domain. In this case, the data never travels on an IP segment without encryption. In a **gateway** implementation, some IP segments are not secured, and another node provides encryption for other segments in the path.

Integrated VPN solutions provide better security than firewall or gateway encryption solutions because they provide encryption support between the network endpoints, thus thwarting both external and internal hackers.

IP packet filtering is a technology that is inserted at a low level in the IP stack. IP, or Internet Protocol, runs on every host and is responsible for routing packets to their destination. As packets are about to be sent or received, the packet filter decides whether the operation should be performed or whether the packet should be discarded. It does this by comparing the packet against a set of rules that say what packets are permitted. Packet filters are a good way to selectively allow some traffic into a subnetwork. They are also a good way of protecting the higher communications layers and applications from unwanted traffic. Because good traffic goes through unchanged, the protection is completely transparent to users and applications. Packet filters look at the first few bytes of each packet, called the packet header. The packet header describes the connection protocol and application protocol that are being used. Using this information the packet filter determines whether it should allow the packet through or discard it.

WebSphere Application Server 3.02 Advanced Edition provides a unified security model that can be used to secure web pages, servlets, and enterprise beans. It also provides an HTTP Single Sign-On solution and improved LDAP directory support.

INSCO security design

INSCO implemented network configuration for its customer and agent access functions according to the screened subnet architecture. This configuration introduces a third network known as the demilitarized zone (DMZ). Two security gateways are used to create the three networks. Details on the configuration can be found in scenario 1, "Scaling Up e-business Implementations with WebSphere - AS/400 Edition." This report is available on the web at:

http://www-4.ibm.com/software/ebusiness/scalingwebsphere.html

For the screened subnet architecture implementation, INSCO deployed a protocol firewall and a domain firewall. Both firewalls perform IP packet filtering. The domain firewall also performs network address translation (NAT). NAT is used to "hide" the private addresses of the INSCO internal network servers. AS/400 IP packet filtering is also deployed on the servers running in the DMZ. This adds yet another layer of defense. Further details on the INSCO screened subnet architecture implementation can be found in scenario 1.

The HTTPS protocol secures the communication between end-users and the INSCO transactional web servers. We are currently using the HTTP protocol for communications between the INSCO transactional web servers and the Payment Manger server. We plan to switch to the HTTPS protocol as soon as the Payment Manager 2.1 receives export approvals.

For the automated approval application process, INSCO implemented a host-to-host VPN. There is no need to connect the INSCO and underwriter internal networks. Only the internal systems hosting the automated approval application need to be connected.

Other security design considerations

We have stepped through the INSCO security implementation fairly quickly. There are many more security considerations that today's e-business companies need to address.

As stated earlier, the best approach is a layered approach:

When looking for an ISP, we chose one that obviously took our security concerns seriously. They have a special team for dealing with and tracking security breaches as well as denial of service attacks. Not all ISPs have this level of concern. We wanted to make sure we would get the support we need should our system ever be compromised. We also made sure the ISP had implemented the latest denial-of-service techniques that are appropriate for an ISP. Our ISP also allowed us to put in place some basic packet filtering rules, which was yet another layer of security.

We make sure that we review and test our firewall configurations regularly. This is especially important after adding a new application to our web server or altering our firewall configuration. We have a separate test group from those making the configuration changes.

We have taken an exclusionary approach to securing our AS/400e servers. The following are the choices we made:

- Users only have authority to run the applications they need to run. *PUBLIC authority on most application libraries is *EXCLUDE to prevent general access. In addition, *PUBLIC authority on the '/' (root) directory has been changed to *RX.

- We loaded only the products we intended to use on this system. No extraneous OS/400 options, licensed programs, or third-party applications were loaded on a system unless we needed that function. The reason for this is to minimize the number of interfaces that must be secured (and the potential tools with which the system could be attacked).

- QSECURITY is set to 50, which provides the greatest system integrity capabilities.

- We used the Security Configuration Wizard (available through Operations Navigator) to determine the best settings for the security-related system values, given our configuration.

The wizard also provided help in scheduling security monitoring tools and setting system auditing values.

- The only TCP/IP servers started were those associated with the TCP/IP applications we were using, for example, HTTP. Port restrictions are used to ensure no other port can be exploited.
- Reports are run regularly (once a month) to ensure that object-level security has not changed and that the only user profiles on the system are ones that actually need access to the system, are IBM-supplied profiles, or application profiles.

The scenario network

Before diving into setup and configuration details, we need to have an understanding of the scenario network implementation. The following figure illustrates the desired network configuration:

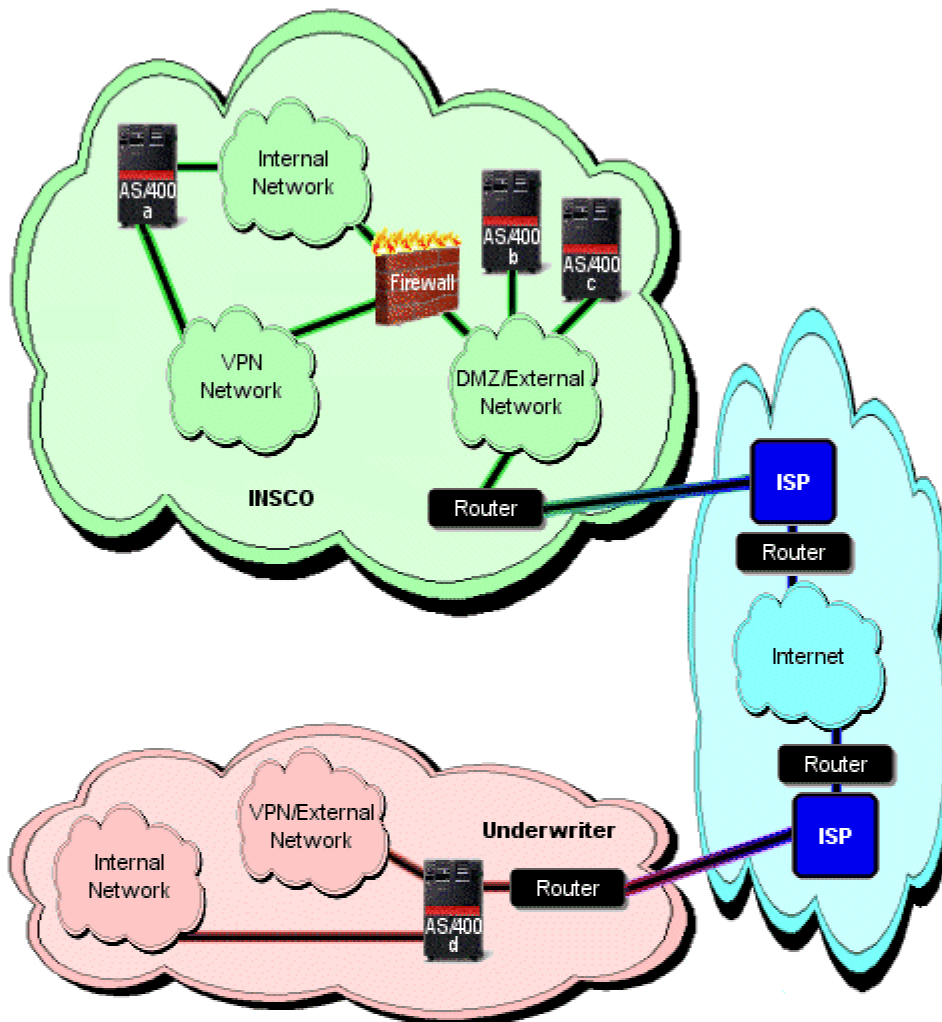


Figure 20. Desired scenario network configuration

Due to restrictions in our testing lab, our actual implementation ended up as follows:

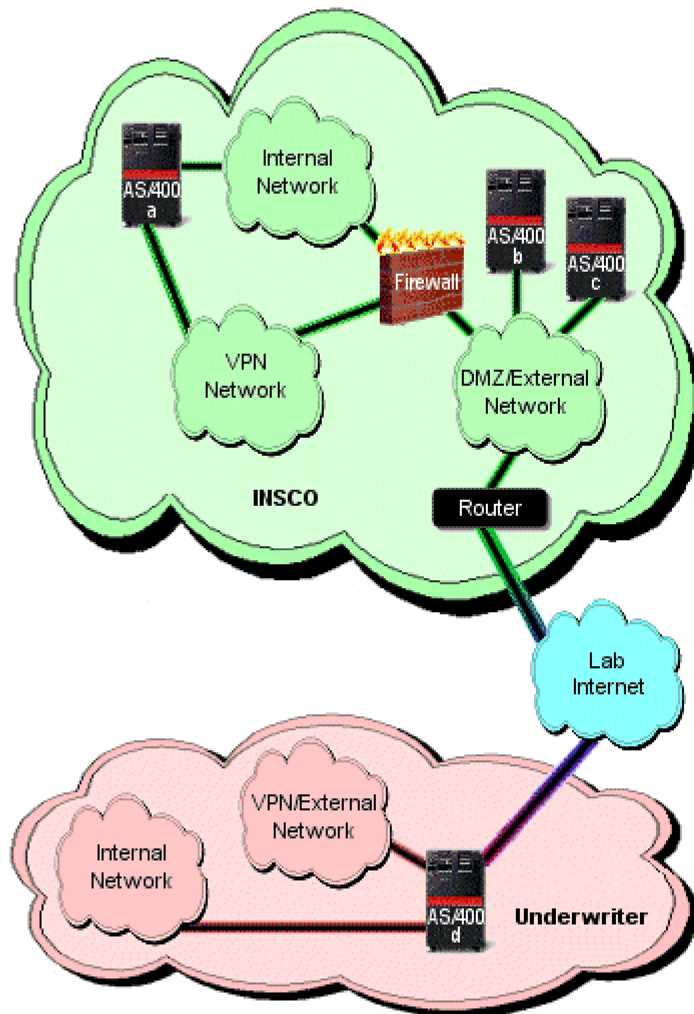


Figure 21. Actual scenario network configuration

Configuring IP packet filtering and NAT

First, let's look at the changes to the INSCO firewall configuration. The INSCO firewall implementation is discussed in our previously published report, titled *Scaling Up e-business Implementations with WebSphere - AS/400 Edition.* This report is available on the web at:

http://www-4.ibm.com/software/ebusiness/scalingwebsphere.html

Security enhancements made to the INSCO application required updates to the INSCO firewall configuration. WebSphere security was implemented using an LDAP server. The LDAP server resides *behind* the firewall in the INSCO internal network. Filter rules needed to be added to permit INSCO transactional web server access to the LDAP server. Also, the INSCO application

now supports online payments. The Payment Manager server resides *behind* the firewall in the INSCO internal network. Filter rules needed to be added to permit INSCO transactional web server access to the Payment Manager server.

Note: Detailed information on the specific rules can be found in the “Appendix.”

We also used Network Address Translation (NAT) to hide two internal interfaces from the public. One interface is used to access the database and LDAP servers. The other interface is used to access the Payment Manager.

Note: Specific information on the NAT configuration can be found in the “Appendix.”

Setting up the VPN

Before setting up your VPN, you need to make sure you have established a network route to and from the endpoints of the VPN. This can be challenging because you may need to momentarily open up your private network to verify the connection by using a utility like Ping. This will require coordination and cooperation between the owners of the endpoint systems and the ISPs involved.

A VPN was established to support the automated approval application process. There are several types of VPNs. For this implementation a host-to-host VPN was established. The **VPN New Connection Wizard** was used to create the VPN configurations. This wizard makes it easy to create a VPN connection.

The wizard creates a dynamic key group with a policy setting of filter rule for remote ports and protocol. The only protocol that was needed was TCP port 80 (HTTP). We found it easiest to simply modify the dynamic key connection properties to restrict the traffic. We first changed the dynamic key group settings from filter rule to connection for local ports, remote ports, and protocol. We then modified the dynamic key connection services properties to restrict the traffic. We specified a local port of 80, a remote port of any port, and a protocol of TCP.

We then added filter rules to enable the VPNs. Two filter rules were added to allow the Internet Key Exchange (IKE) negotiations. Another filter rule was added to serve as the IPSec anchor rule for the VPN. Following are the options selected (indicated by check marks) while using the Wizard to create the VPNs.

Note: Detailed VPN configuration information can be found in the “Appendix.”

INSCO VPN configuration

Host to host connection group name: UnderwriterVPN

Remote key server

- Authentication: Pre-shared key
- Remote hosts to connect to:

  199.0.0.44

Data Policy

- Highest security/lowest performance

Key Policy

- Balanced security/performance

Local key server:

- Identifier type: Version 4 IP address
- IP Address: 199.0.0.65

Underwriter VPN configuration

Host to host connection group name: InscoUnderwriterVPN

Remote key server

- Authentication: Pre-shared key
- Remote hosts to connect to:

  199.1.1.65

Data Policy

- Highest security/lowest performance

Key Policy

- Balanced security/performance

Local key server:

- Identifier type: Version 4 IP address
- IP Address: 199.0.0.44

We found the redbook *AS/400 Internet Security: Implementing AS/400 Virtual Private Networks*, SG24-5404-00, helpful in constructing the VPN connection. It can be found at:

http://www.redbooks.com

Additional information on virtual private networking can found at the AS/400 Information Center:

http://www.as400.ibm.com/infocenter

Within the AS/400 Information Center, select **Internet and Secure Networks** and then select **Virtual private networking**.

Appendix

The "Appendix" includes reference information and the more complex setup procedures that pertain to this scenario.

Reference information

The following subtopics discuss the software and hardware used, where to find the source code, views of some application screens, database details and data files used, and examples of LDIF, XML, and the Payment Manager API.

System hardware

Details concerning hardware content of the two AS/400e servers used for this scenario are described in the following tables. **AS4SYS1a** and **AS4SYS1b** are the two AS/400e servers located in the DMZ. **AS4SYS2a** is the AS/400 located in the INSCO intranet. **AS4SYS2b** is the AS/400 located in the underwriter intranet. **AS4SYS2c** is the AS/400 located in the scenario internet.

AS/400e 1
9406 server 740

Partition	**AS4SYS1a**	**AS4SYS1b (primary)**
Processors	4	4
Release	V4R4M0	V4R4M0
System	ASP 472 GB	ASP 728 GB
Main storage	19.8 GB	19.4 GB

AS/400e 2
9406 server 730

Partition	**AS4SYS2a**	**AS4SYS2b**	**AS4SYS2c (primary)**
Processors	4	2	2
Release	V4R4M0	V4R4M0	V4R4M0
System	ASP 267 GB	ASP 155 GB	ASP 120 GB
Main storage	8.9 GB	2 GB	1.7 GB

System software requirements

The software inventory for each AS/400e server is described in the following tables:

AS4SYS1a, AS4SYS1b:

DB2 UDB for AS/400	
Java Developer Kit 1.1.7	5769-JV1
AS/400 Developer Kit for Java	5769-JV1
AS/400 Toolbox for Java	5769-JC1
IBM HTTP Server for AS/400	5769-DG1
Client Encryption 128-bit	5769-CE3
WebSphere Application Server for AS/400 (version 3.02)	5733-WA2
Crypto Access Provider 128-bit for AS/400	5769-AC3
Cryptographic Service Provider	5769-SS1

AS4SYS2a:

DB2 UDB for AS/400	
Java Developer Kit 1.1.7	5769-JV1
WebSphere Application Server for AS/400 (version 3.02)	5733-WA2
IBM HTTP Server for AS/400	5769-DG1
WebSphere Payment Manager for AS/400 (version 2.1)	5733-PY2
LDAP 2.1	
Crypto Access Provider 1228-bit for AS/400	5769-AC3
Cryptographic Service Provider	5769-SS1
Digital Certificate Manager	5769-SS1
Domain Name System	5769-SS1
AS/400 Integration for NT	5769-SS1
Host Servers	5769-SS1

AS4SYS2b:

DB2 UDB for AS/400	
WebSphere Application Server for AS/400 (version 3.02)	5733-WA2
IBM HTTP Server for AS/400	5769-DG1
Java Developer Kit 1.1.7	5769-JV1

AS4SYS2c:

Crypto Access Provider 128-bit for AS/400	5769-AC3
Cryptographic Service Provider	5769-SS1
Digital Certificate Manager	5769-SS1
Domain Name System	5769-SS1

Application source code

Along with this report, a zip file is available for download. This zip file contains the HTML file, images, properties file, Java source files, and JSP files necessary to re-create this scenario.

Once you have downloaded this file, unzip it to a location of your choice, and then refer to the readme.txt file for instructions on deployment.

Application screen views

The following images are the full size views of what the INSCO application looks like to the user. For the order of appearance of the screen views for the customer and agent, see Figure 3 (Customer flow diagram) and Figure 4 (Agent and account administrator flow diagram), respectively.

Figure 23. INSCO Login page

Select Customer Information

Hello Jane Smith

Please Select:

Policy Number:

OR

Customer Number:

OR

Last Name:

Select New Customer Log Off

Figure 24. Select Customer Information page

Search Results

Jane Smith, Please Choose a Customer

First Name	Last Name	Address	VIEW
Elroy	Doe	68319 Ingraham Drive W	View
Jane	Doe	123 Broadway	View
Jenny	Doe	323 Oak St.	View
John	Doe	345 Any St.	View
Elnora	Doebler	16548 Baraboo Parkway S	View
Brett	Doepke	97043 Loxley Circle SE	View
Lisandra	Doering	50835 Bulls Gap Street E	View
Allegra	Doerr	72863 Francisco Circle E	View
Louis	Doescher	90221 Oostburg Parkway SW	View
Basilia	Doetsch	94094 Collettsville Circle NW	View

Cancel

Figure 25. Search Results page

INSCO Agent Home Page

Welcome Jane Smith

Customer Information

Customer Number: **C0030037**

Last Name: **Doe** First Name: **John** Middle Initial: **W**

Address: **345 Any St.**

City: **Beverly Hills** State: **CA** Zip: **90212**

Home Phone Number: **(281)213-3414** Work Phone Number: **(281)993-0594**

Email Address: **johndoe@some-isp.com**

[Update Personal Info.]

Policy Information

	Policy Number	Policy Type	Policy Status	Next Payment Due	Payment Amount
View	P0000060086	AUTO	New	November 23, 2000	$ 300
View	P0000060087	HOME	New	November 23, 2000	$ 400

[Submit Home Application] [Submit Auto Application] [Select Another Customer]

[Log Off]

For further assistance, please email us the description of your problem.

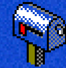


Figure 26. INSCO Agent Home Page

INSCO Customer Home Page

Welcome John Doe

Your Personal Information

Last Name: **Doe** First Name: **John** Middle Initial: **W**

Address: **345 Any St.**

City: **Beverly Hills** State: **CA** Zip: **90212**

Home Phone Number: **(281)213-3414** Work Phone Number: **(281)993-0594**

Email Address: **johndoe@some-isp.com**

Update Personal Info.

Your Policy Information

	Policy Number	Policy Type	Policy Status	Next Payment Due	Payment Amount	
View	P0000060086	AUTO	New	November 23, 2000	$ 300	☐
View	P0000060087	HOME	New	November 23, 2000	$ 400	☐

Submit Payment

Log Off

For further assistance, please email us the description of your problem.

Figure 27. INSCO Customer Home Page

Update Personal Information

Personal Information

Last Name: Doe First Name: John Middle Initial: W

Address: 345 Any St.

City: **Beverly Hills** State: **CA** Zip: 90212 *

Home Phone Number: (281)213-3414 Work Phone Number: (281)993-0594

Email Address: johndoe@some-isp.com

Submit Changes **Reset** **Cancel**

* City and State will be determined from the Zip Code

Figure 28. Update Personal Information page

View Home Policy

Back to Home Page

Policy Number: P0000060087

Policyholder: **John Doe**

Status: **New** Status Details:

Agent Responsible: **Jane Smith** Effective Date: **May 23, 2000**

Home Information

Address: **345 Any St.** Zip: **90212**

Number of Rooms: **7** Type of Doorlock: **Spring**

Year Built: **1999** Year Occupied: **2000**

Coverage

Type	Term	Limit	Deductible
Fire	$ 100	$ 100000	$ 2000
Flood	$ 200	$ 100000	$ 1000
Theft	$ 100	$ 50000	$ 1000

Payment Amount: **$ 400**

Date of Last Payment: **No Payments Recorded**

Next Payment Due: **November 23, 2000**

Back to Home Page

Figure 29. View Home Policy page

View Auto Policy

Back to Home Page

Policy Number: P0000060086

Policyholder: **John Doe**

Status: **New** Status Details:

Agent Responsible: **Jane Smith** Effective Date: **May 23, 2000**

Vehicle Information

Make: **Ford** Model: **Focus** Model Year: **2000**

Body Type: **Four-Door Hard Top** Number of Cylinders: **4**

Vehicle Identification Number(VIN): **A123B4567CD890EF1**

Date of Purchase: **May 23, 2000** State of Registration: **MN**

Primary Driver's License Number: **A123456789012**

Estimated Annual Miles Driven: **40000**

Coverage

Type	Term	Limit	Deductible
Comprehensive	$ 200	$ 100000	$ 1000
Collision	$ 100	$ 80000	$ 1000

Payment Amount: **$ 300**

Date of Last Payment: **No Payments Recorded**

Next Payment Due: **November 23, 2000**

Back to Home Page

Figure 30. View Auto Policy page

Submit Automobile Application

Submit Auto Application | Reset | Cancel

Please Complete Form:

Customer:	**John Doe**
Agent:	**Jane Smith**
Effective Date: YYYY-MM-DD	2000-5-23
Make:	
Model:	
Year of Car:	
Date Purchased: YYYY-MM-DD	2000-5-23
Body Type:	2 Door Hard Top
Number of Cylinders:	4
VIN:	
State Registered in:	
Estimated Annual Mileage:	
Drivers Licence Number:	

Add Coverage

Type: Personal Injury Protection

Term: $

Limit: $

Deductible: $

Add Coverage

Coverages added

Type	Term	Limit	Deductible
---- No Coverages Added ----			

Submit Auto Application | Reset | Cancel

Figure 31. Submit Automobile Application page

Submit Home Application

Submit Home Application | Reset | Cancel

Please Complete Form:

Customer:	**John Doe**
Agent:	**Jane Smith**
Effective Date: YYYY-MM-DD	2000-5-23
House Address:	
Zip Code: 55555	
Rooms:	1
Type of Lock:	Double Cylinder
Year Built:	
Year Occupied:	

Add Coverage

Type: Theft

Term: $

Limit: $

Deductible: $

Add Coverage

Coverages added

Type	Term	Limit	Deductible
---- No Coverages Added ----			

Submit Home Application | Reset | Cancel

Figure 32. Submit Home Application page

New INSCO Customer

New Customer for Jane Smith

Please Complete Form:

First Name:	
Middle Initial:	
Last Name:	
Customer's Password:	
Password (again to Verify):	
Address:	
Zip: 55555	
Email:	
Home Phone: (111) 111-1111	
Work Phone: (111) 111-1111	
Social Security Number: 111223333	
Date Of Birth: YYYY-MM-DD	
Marital Status:	Married
Occupation:	
Household Income: 10000	

Add Customer | Reset | Cancel

Figure 33. New INSCO Customer page

Make Payment

John Doe, here are the policies that you selected to pay for:

Policy Number	Type	Due Date	Price
P0000060086	AUTO	11-23-00	$ 300.00
P0000060087	HOME	11-23-00	$ 400.00
Total Payment:			**$ 700.00**

Credit Card Information		
Credit Card Type:	American Express	
Expiration Date:	January	2000
Credit Card Number:	*	

* Note: Don't enter any dashes or spaces (ie. 1111222233334444)

Make Payment | Cancel

Figure 34. Make Payment page

Make Payment

Success!

You have successfully completed your payment.
Please click *Home* or wait to be transferred to your home page.

Home

Figure 35. Make Payment - Success page

Your Request has been submitted

You will be re-directed to the home page in 5 seconds.
Or you can press the button below.

Back to Home Page

Figure 36. Request Submitted Page

INSCO Log Off

Good Bye John Doe !

Login

Figure 37. Log Off page

Database details

Definitions for each of the data files used in the INSCO database are listed below. Each data file is stored in its own separate table. The tables contain the following fields:

Key	Any special requirements associated with a field. This indicates that a constraint or an index needs to be applied to the data file to enforce the rule. In the case of a check constraint, a description of the check constraint is given after the table. Possible values used in the table are Primary key (P), Foreign key (F), Unique (U), Check (C), and Index (Inx).
Field Name	The name of the field in the data file
Alias Name	Another way to reference the field. Normally, it is a bit longer and more descriptive than the Field Name.
Data Type	The type of data that is contained in the field: character, integer, date, and so forth.
Length/Alloc	Length refers to how long the field is. Alloc is how much space is initially allocated to that field.
Allow Null	A "Y" indicates that this field can contain a null value.
Digits, DecPos	Used when the Data Type field is listed as DECIMAL. Digits describes the length of the field, and DecPos is the number of decimal positions allowed. For example 8,2 means the field could accept a value like 123456.78
Description/Example	A brief description of what the field contains, and possibly an example to show the format of the data to be contained.

Data files

CUSTOMER--File contains information for each customer of INSCO

Key	Field Name	Alias Name	Data Type	Length /Alloc	Allow Null	Description/Example
PC	CNUM	CUSTOMER_NUMBER	CHAR	8		Customer identification number - C0000001
	CFNAME	FIRST_NAME	VARCHAR	20/10		First name
	CMI	MIDDLE_INITIAL	CHAR	1	Y	Middle initial
Inx	CLNAME	LAST_NAME	VARCHAR	25/10		Last name
	ADDRESS	HOME_ADDRESS	VARCHAR	40/20		Address
F	ZIP	ZIP_CODE	CHAR	9		Zip code
	EMAIL	E_MAIL_ADDRESS	VARCHAR	255/30	Y	E-mail address
	DAYPHONE	WORK_PHONE	CHAR	14	Y	Day time work number (xxx) xxx-xxxx
	EVEPHONE	HOME_PHONE	CHAR	14	Y	Evening telephone number - (xxx) xxx-xxxx
	SSN	SOCIAL_SECURITY_N UMBER	CHAR	9		Social security number (000000001)
	DOB	DATE_OF_BIRTH	DATE			Date of birth
C	MARIED	MARITAL_STATUS	CHAR	1		Marital status M - Married S - Single D - Divorced W - Widowed
	JOBTITLE	OCCUPATION	VARCHAR	50/15	Y	Occupation of insured
	INCOME	HOUSEHOLD_INCOME	INTEGER			The household income of the customer

Check constraints:

- A constraint is needed to make sure the MARITAL_STATUS field contains only the four choices given.
```
CONSTRAINT CUSTOMER_MARIED_CHK CHECK
(UPPER (MARITAL_STATUS) IN ('S','M','D','W'))
```
- A constraint is needed to make sure the CUSTOMER_NUMBER fits the required format defined in the table above.
```
CONSTRAINT CUSTOMER_CNUM_CHECK CHECK
(CUSTOMER_NUMBER BETWEEN 'C0000000' AND 'C9999999')
```

ZIP_CODE--File contains zip codes and state names

Key	Field Name	Alias Name	Data Type	Length /Alloc	Description/Example
PC	ZIP	ZIPCODE	CHAR	9	Zip code
	CITY	CITY_NAME	VARCHAR	40/20	City name
	CODE	STATE_CODE	CHAR	2	Abbreviated state name
	FULLNAME	FULL_STATE_NAME	VARCHAR	40/20	Full state name

Check constraints:

- A constraint is needed to make sure the ZIPCODE field contains only numeric characters
```
CONSTRAINT ZIP_CODE_ZIP_CHK CHECK (TRIM ( ZIP )
BETWEEN '000000000' AND '999999999'
```

BASEPOLICY--File contains policy information common to both Auto and Home

Key	Field Name	Alias Name	Data Type	Length /Alloc	Allow Null	Description/Example
PC	PNUM	POLICY_NUMBER	CHAR	11		Policy number P1234567890
F	CNUM	CUSTOMER_NUMBER	CHAR	8		Customer identification number - C0000001
F	ANUM	AGENT_NUMBER	CHAR	8		Agent identification number - A0000001
	DATESTR	EFFECTIVE_DATE	DATE			Date the policy takes effect
	DUEDATE	NEXT_DUE_DATE	DATE			Date that the next bill is due
C	PTYPE	POLICY_TYPE	CHAR	4		Type of policy purchased - Home or Auto.
C	STATUS	STATUS_CODE	CHAR	1		Status of policy A - Approved R - Rejected N - New P - Pending C - Closed
	DESC	DESCRIPTION	VARCHAR	255/10	Y	Description of status that was given. Policy was rejected because applicant has a poor credit history.

Check constraints:

- A constraint is needed to make sure the MARITAL_STATUS field contains only one of the two choices given
```
CONSTRAINT BASE_TYPE_CHK CHECK
(UPPER ( POLICY_TYPE ) IN ('HOME', 'AUTO'))
```
- A constraint is needed to make sure the CUSTOMER_NUMBER fits the required format
```
CONSTRAINT CUSTOMER_CNUM_CHECK CHECK
(CUSTOMER_NUMBER BETWEEN 'C0000000' AND 'C9999999')
```
- A constraint is needed to make sure the AGENT_NUMBER fits the required format
```
CONSTRAINT AGENT_CNUM_CHECK CHECK  (AGENT_NUMBER BETWEEN 'A0000000' AND
'A9999999')
```
- A constraint is needed to make sure the POLICY_NUMBER fits the required format
```
CONSTRAINT BASE_PNUM_CHK CHECK
(POLICY_NUMBER BETWEEN 'P0000000000' AND 'P9999999999')
```
- A constraint is needed to make sure the MARITAL_STATUS field contains only one of the two choices given
```
CONSTRAINT STATUS_STAT_CHK CHECK (UPPER (STATUS_CODE) IN ('A','R','N','P','C'))
```

AGENT--File contains agent information

Key	Field Name	Alias Name	Data Type	Length /Alloc	Allow Null	Description/Example
PC	ANUM	AGENT_NUMBER	CHAR	8		Agent identification number in the form of: A0000001
	AFNAME	FIRST_NAME	VARCHAR	20/10		First name of agent
	ALNAME	LAST_NAME	VARCHAR	25/10		Last name of agent
	EMAIL	E_MAIL_ADDRESS	VARCHAR	255/30	Y	E-mail address

Check constraints:

- A constraint is needed to make sure the AGENT_NUMBER fits the required format
```
CONSTRAINT AGENT_ANUM_CHK CHECK (AGENT_NUMBER BETWEEN 'A0000000' AND 'A9999999')
```

HOME--File contains information for a home insurance policy

Key	Field Name	Alias Name	Data Type	Length /Alloc	Description/Example
UF	PNUM	POLICY_NUMBER	CHAR	11	Policy number - P1234567890
	ADDRESS	HOUSE_ADDRESS	VARCHAR	40/20	Address of home covered
F	ZIP	ZIP_CODE	CHAR	9	Zip code of home covered
	ROOMS	NUMBER_OF_ROOMS	INTEGER		Number of rooms in house
	LOCKTYPE	LOCK_CODE	CHAR	6	Type of door lock DBLCL - Double cylinder DEADB - Dead bolt SPRING - Spring OT - Other
C	BUILT	YEAR_BUILT	CHAR	4	Year house was built
C	OCCUPIED	YEAR_OCCUPIED	CHAR	4	Year house was occupied

Notes:

- The types of door locks and their codes were taken from page 49 of the ACORD_Code_Lists_for_PC.doc.
- A check constraint was not placed on LOCK_CODE because this field could be expanded to accept more types of locks than the ones that we limited ourselves to in this scenario.

Check constraints:

- A constraint is needed to make sure the YEAR_BUILT fits the required format
```
CONSTRAINT HOUSE_BUILT_CHK CHECK (YEAR_BUILT  BETWEEN '0000' AND '9999')
```
- A constraint is needed to make sure the YEAR_OCCUPIED fits the required format
```
CONSTRAINT HOME_OCCUPIED_CHK CHECK (YEAR_OCCUPIED BETWEEN '0000' AND '9999')
```

AUTO--File contains information for an auto insurance policy

Key	Field Name	Alias Name	Data Type	Length /Alloc	Description/Example
UF	PNUM	POLICY_NUMBER	CHAR	11	Policy number - P1234567890
	MAKE	MANUFACTURER	VARCHAR	20/10	Vehicle's make - Saturn
C	MODEL	MODEL	VARCHAR	20/10	Vehicle's model - SL2
	YR	MODEL_YEAR	CHAR	4	Vehicle's year - 1997
	BODY	BODY_TYPE	CHAR	6	Vehicle's body type 2DRHT - 2-door hard top 4DRHT - 4-door hard top 4W - 4-door wagon 4WD - 4-wheel drive CONVT - Convertible VANMV - Minivan TRUCKT - Truck MC - Motorcycle
	CYLNDRS	CYLINDERS	INTEGER		Number of cylinders - 4
	VIN	IDENTIFICATION_NUM	CHAR	30	Vehicle identification number - 39483038593098457277
	STATE	REGISTERED_STATE	CHAR	2	State vehicle is registered in - MN
	DATEOWN	PURCHASE_DATE	DATE		Date vehicle was purchased
	MILES	ANNUAL_MILES	INTEGER		Estimated annual mileage
	LICENSE	DRIVER_LICENSE	CHAR	13	Customer's drivers license number - T123456789012

Notes:

- Coverage types were taken from pages 99 - 101of ACORD_Code_Lists_for_PC.doc. Many additional codes could be used, but we decided to limit ourselves to the types listed.
- A check constraint was not placed on BODY_TYPE because this field could be expanded to accept many more types than the ones that we limited ourselves to in this scenario.

Check constraints:

- A constraint is needed to make sure the MODEL_YEAR fits the required format

```
CONSTRAINT AUTO_YEAR_CHK CHECK (MODEL_YEAR BETWEEN '0000' AND '9999')
```

COVERAGE -- File contains information on coverage for each policy

Key	Field Name	Alias Name	Data Type	Length	Description/Example
PF	PNUM	POLICY_NUMBER	CHAR	11	Policy number - P1234567890
P	COVNUM	COVERAGE_NUMBER	INT		Coverage number
	CTYPE	COVERAGE_TYPE	CHAR	5	Coverage type AUPIP - Auto personal injury protection ATOWG - Auto towing ACOMP - Auto comprehensive ACOLL - Auto collision THEFT - Theft VMM - Vandalism and malicious mischief FLOOD - Flood FIRE - Fire
	TERM	TERM_AMOUNT	INTEGER		Term amount - Amount that needs to be paid each term (every 6 months)
	LIMIT	LIMIT_AMOUNT	INTEGER		Limit amount - The maximum amount that will be paid on a claim
	DEDUCT	DEDUCTIBLE	INTEGER		Deductible amount

Notes:

- Coverage types were taken from pages 22 - 40 of ACORD_Code_Lists_for_PC.doc. Many additional codes could be used, but we decided to limit ourselves to the types listed.
- A check constraint was not placed on COVERAGE_TYPE because this field could be expanded to accept many more types than the ones that we limited ourselves to in this scenario.

PAYHIST -- File contains the payment history for each policy

Key	Field Name	Alias Name	Data Type	Length	Digits, DecPos	Description/Example
P	ADDED	ADDED	TIMESTAMP			Timestamp when record was added
F	PNUM	POLICY_NUMBER	CHAR	11		Policy number - P1234567890
	DUE	AMOUNT_DUE	DECIMAL		8, 2	Amount that is due
	PAID	AMOUNT_PAID	DECIMAL		8, 2	Amount that customer paid

PENDING -- File contains the pending policy applications

Key	Field Name	Alias Name	Data Type	Length	Description/Example
P	REQID	REQUEST_ID	CHAR	50	Unique request ID for this policy application
F	POLNUM	POLICY_NUMBER	CHAR	20	Policy number - P1234567890
	SCODE	STATUS_CODE	CHAR	1	Status of the policy - P
	DESC	DESCRIPTION	VARCHAR	255/20	Status description

The following table resides on the underwriter's system and contains a history of all policy applications submitted to the underwriter.

APPS -- File contains the underwriter's policy applications

Key	Field Name	Alias Name	Data Type	Length	Description/Example
P	REQID	REQUEST_ID	VARCHAR	50	Unique request ID for this policy application.
	TIMESTAMP	TIMESTAMP	TIMESTAMP		Timestamp when this application was added
	POLTYPE	POLICY_TYPE	CHAR	4	Type of policy - HOME or AUTO
	COMPANY	COMPANY	VARCHAR	100/20	Company the policy application came from
	ACCTNUM	ACCOUNT_NUM	INTEGER		Company's account number with the underwriter
	CUSTNAME	CUSTOMER_NAME	VARCHAR	100/20	Customer's name who owns the policy
	STATUS	STATUS	CHAR	1	Status of the policy - P

Examples: LDIF import files

The following LDIF examples were used to create the DIT structure and populate the LDAP server with data that was migrated from the database.
adminDIT.ldif

This LDIF file sets up the top-level objects in the DIT.

```
dn: dc=sys, dc=insco, dc=com
dc: sys
objectclass: domain

dn: o=insco, dc=sys, dc=insco, dc=com
o: insco
inheritOnCreate: FALSE
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rsc
entryOwner: access-id:cn=administrator
objectclass: organization
```

```
dn: cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
cn: inscoAdmin
sn: inscoAdmin
userPassword: insco
entryOwner: access-id:cn=administrator
inheritOnCreate: FALSE
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rsc
objectclass: inetOrgPerson
objectclass: ePerson
```

inscoDIT.ldif

This LDIF file sets up the groups and the agent, customer, and policy containers.

```
dn: cn=securityAdministrators, o=insco, dc=sys, dc=insco, dc=com
cn: securityAdministrators
member: cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=securityAdministrators, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
inheritOnCreate: FALSE
objectclass: AccessGroup

dn: cn=accountAdministrators, o=insco, dc=sys, dc=insco, dc=com
cn: accountAdministrators
member: cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=securityAdministrators, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
inheritOnCreate: FALSE
objectclass: AccessGroup

dn: cn=allAgents, o=insco, dc=sys, dc=insco, dc=com
cn: allAgents
member: cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=securityAdministrators, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=accountAdministrators, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
inheritOnCreate: FALSE
objectclass: AccessGroup

dn: ou=policies, o=insco, dc=sys, dc=insco, dc=com
ou: policies
inheritOnCreate: FALSE
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
```

```
aclEntry: group:cn=allAgents, o=insco, dc=sys, dc=insco, dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rwsc
objectclass: organizationalUnit

dn: ou=agents, o=insco, dc=sys, dc=insco, dc=com
ou: agents
inheritOnCreate: FALSE
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=securityAdministrators, o=insco, dc=sys, dc=insco, dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rwsc
objectclass: organizationalUnit

dn: ou=customers, o=insco, dc=sys, dc=insco, dc=com
ou: customers
inheritOnCreate: FALSE
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com
aclEntry: access-id:cn=inscoAdmin, o=insco, dc=sys, dc=insco, dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: group:cn=allAgents, o=insco, dc=sys, dc=insco, dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rwsc
objectclass: organizationalUnit
```

agents.ldif

This LDIF file adds all the agents to the LDAP directory. Only one agent is shown here.

```
dn: cn=A0000001, ou=agents, o=insco, dc=sys, dc=insco, dc=com
cn: A0000001
sn: A0000001
userPassword: insco01
inscoPersonType: agent
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com
entryOwner: access-id:cn=A0000001,ou=agents,o=insco,dc=sys,dc=insco,dc=com
objectclass: inetOrgPerson
objectclass: ePerson
objectclass: inscoPerson
```

customers.ldif

This LDIF file adds all the customers to the LDAP directory. Only one customer is shown here.

```
dn: cn=C0000001, ou=customers, o=insco, dc=sys, dc=insco, dc=com
cn: C0000001
sn: C0000001
userPassword: insco01
inscoPersonType: customer
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com
aclEntry: access-id:cn=C0000001,ou=customers,o=insco,dc=sys,dc=insco,dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rsc
objectclass: inetOrgPerson
```

```
objectclass: ePerson
objectclass: inscoPerson
```

policies.ldif

This LDIF file adds all the policies to the LDAP directory. Only one policy is shown here.

```
dn: cn=P55557,ou=policies,o=insco,dc=sys,dc=insco,dc=com
cn: P55557
aclEntry: group:cn=accountAdministrators,o=insco,dc=sys,dc=insco,dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: access-id:cn=C22223,ou=customers,o=insco,dc=sys,dc=insco,dc=com:
      object:a:normal:rwsc:sensitive:rwsc:critical:rwsc
aclEntry: access-id:cn=A11113,ou=agents,o=insco,dc=sys,dc=insco,dc=com:
      object:ad:normal:rwsc:sensitive:rwsc:critical:rwsc
entryOwner: access-id:cn=administrator
entryOwner: access-id:cn=inscoAdmin,o=insco,dc=sys,dc=insco,dc=com
entryOwner: group:cn=securityAdministrators,o=insco,dc=sys,dc=insco,dc=com
objectclass: inscoPolicy
```

groups.ldif

This LDIF file adds all the groups to the LDAP directory. Only the cn=accountAdministrators group is shown here.

```
dn: cn=accountAdministrators, o=insco, dc=sys, dc=insco, dc=com
member: cn=A0000500, ou=agents, o=insco, dc=sys, dc=insco, dc=com
member: cn=A0000501, ou=agents, o=insco, dc=sys, dc=insco, dc=com
member: cn=A0000502, ou=agents, o=insco, dc=sys, dc=insco, dc=com
```

Examples: XML coded messages

The following code is an example of a home policy application request:

```
<?xml version="1.0"?>
<!DOCTYPE PersHomeAddRQ SYSTEM "/insco/PersHomeAddRq.dtd">
<PersHomeAddRQ>
  <RqUID>INSCO-P0000000002-954363583374</RqUID>
  <Account>
    <AccountNumber>10000342</AccountNumber>
    <Name>
      <NameType>Corporation/Commercial</NameType>
      <CommercialName>
        <CompanyName>INSCO Insurance Company</CompanyName>
      </CommercialName>
    </Name>
  </Account>
  <Customer>
    <CustomerName>
      <NameType>Person</NameType>
      <PersonName>
        <LastName>JOHNSON</LastName>
        <FirstName>JAMES</FirstName>
```

```
      </PersonName>
    </CustomerName>
    <CustomerAddress>
      <Addr1>1234 Any Street SW</Addr1>
      <City>Some City</City>
      <StateProv>XZ</StateProv>
      <PostalCode>00001</PostalCode>
    </CustomerAddress>
    <CustomerContact>
      <ContactType>CUSTCONTACT</ContactType>
      <PersonContact>
        <DayPhone>(xxx)xxx-xxxx</DayPhone>
        <EvePhone>(xxx)xxx-xxxx</EvePhone>
        <EMailAddr>anyaddress@isp.com</EMailAddr>
      </PersonContact>
    </CustomerContact>
    <CustType>Retail</CustType>
  </Customer>
  <PersonalPolicy>
    <PolicyNumber>P0000000002</PolicyNumber>
    <LOBCode>HOME</LOBCode>
    <EffectiveDate>1999-09-04</EffectiveDate>
    <ExpirationDate>2000-03-04</ExpirationDate>
    <PersonalApplicationInfo>
      <InsuredInfo>
        <DateOfBirth>YYYY-MM-DD</DateOfBirth>
        <MaritalStatus>D</MaritalStatus>
        <OccupationDescription>Teacher</OccupationDescription>
        <SocialSecurityNumber>000000123</SocialSecurityNumber>
      </InsuredInfo>
      <HouseholdIncome>23199</HouseholdIncome>
    </PersonalApplicationInfo>
  </PersonalPolicy>
  <PersonalHomeLineOfBusiness>
    <LOBCode>HOME</LOBCode>
    <PersonalDwell>
      <DoorLockCode>DBLCL</DoorLockCode>
      <NumberOfRooms>2</NumberOfRooms>
      <YearBuilt>1931</YearBuilt>
      <YearOfOccupancy>1972</YearOfOccupancy>
    </PersonalDwell>
    <Coverage>
      <CoverageCode>THEFT</CoverageCode>
      <Limit>
        <LimitAmount>50000</LimitAmount>
      </Limit>
      <Deductible>
        <DeductibleAmount>50</DeductibleAmount>
      </Deductible>
      <CurrentTermAmount>
        <Amount>100</Amount>
      </CurrentTermAmount>
    </Coverage>
    <Coverage>
      <CoverageCode>Flood</CoverageCode>
      <Limit>
        <LimitAmount>50000</LimitAmount>
      </Limit>
      <Deductible>
        <DeductibleAmount>50</DeductibleAmount>
      </Deductible>
```

```
      <CurrentTermAmount>
        <Amount>100</Amount>
      </CurrentTermAmount>
    </Coverage>
    <Coverage>
      <CoverageCode>FIRE</CoverageCode>
      <Limit>
        <LimitAmount>50000</LimitAmount>
      </Limit>
      <Deductible>
        <DeductibleAmount>50</DeductibleAmount>
      </Deductible>
      <CurrentTermAmount>
        <Amount>100</Amount>
      </CurrentTermAmount>
    </Coverage>
    <Location>
      <PostalAddress>
        <Addr2>5678 Any Street SW</Addr2>
        <City>Some City</City>
        <StateProv>XZ</StateProv>
        <PostalCode>00002</PostalCode>
      </PostalAddress>
    </Location>
  </PersonalHomeLineOfBusiness>
  <Producer>
    <Name>
      <NameType>Person</NameType>
      <PersonName>
        <LastName>DOE</LastName>
        <FirstName>JOHN</FirstName>
      </PersonName>
    </Name>
  </Producer>
</PersHomeAddRQ>
```

The following is an example of an auto policy application response:

```
<?xml version="1.0" ?>
<!DOCTYPE PersAutoAddRS SYSTEM "/insco/PersAutoAddRs.dtd">
<PersAutoAddRS>
  <RqUID>INSCO-P0000000096-954172189002</RqUID>
  <Status>
    <StatusCode>A</StatusCode>
    <Description>Approved.</Description>
  </Status>
</PersAutoAddRS>
```

Payment Manager 2.1 API

The PaymentServlet will use only one API call from the Payment Manager product, AcceptPayment. This command creates an Order object in the Payment Manager. If successful, it will be placed in the Ordered state; if the command fails, it will not be created. You can optionally have the Payment Manager automatically approve the order and/or have funds automatically deposited. For our scenario we will be automatically approving and depositing when we call this command.

This table shows the **required keywords** for the AcceptPayment command.

Keyword	Value
AMOUNT	Amount of payment.
CURRENCY	Currency type of payment (that is, “840” (US Dollar)).
ETAPIVERSION	“3” (indicates PaymentManager Version 2.1).
MERCHANTNUMBER	PaymentManager merchant to place payment with.
OPERATION	“AcceptPayment” (indicates API command to call).
ORDERNUMBER	Unique order number.
PAYMENTTYPE	Specifies payment protocol used (that is, “Test” for test cassette or “CyberCash” for CyberCash cassette)

This table shows the **optional keywords** for the AcceptPayment command.

Keyword	Value
AMOUNTEXP10	The number of decimal places to shift payment amount.
APPROVEFLAG	Whether payment should be automatically approved. 0 - Transaction should not be approved. 1 - Transaction should be approved.
DTDPATH	Path to locally stored DTD.
ORDERURL	URL containing order details.
PAYMENTAMOUNT	Amount to be approved automatically.
PAYMENTNUMBER	Unique payment number.
DEPOSITFLAG	Whether payment should be automatically deposited. 0 - Funds should not be automatically deposited. 1 - Funds should be automatically deposited.
BATCHNUMBER	The batch number under which this payment will be processed.

This example code snippet calls the AcceptPayment command:

```
//setup the info to send to Payment Manager
Hashtable pairs = new Hashtable();
pairs.put(KEY_PAYMENTTYPE, “Test”);
pairs.put(KEY_MERCHANTNUMBER, “853”);
pairs.put(KEY_ORDERNUMBER, “123456789”);
pairs.put(KEY_PAYMENTNUMBER, “123456789”);
pairs.put(KEY_AMOUNT, new Integer(500));
pairs.put(KEY_PAYMENTAMOUNT, new Integer(500));
pairs.put(KEY_CURRENCY, “840”);
pairs.put(KEY_AMOUNTEXP10, “-2”);
pairs.put(KEY_APPROVEFLAG, “1”);
pairs.put(KEY_DEPOSITFLAG, “1”);
pairs.put("$PAN", “1111222233334444”);
pairs.put("$BRAND", “Visa”);
pairs.put("$EXPIRY", “092001);
pairs.put("$ACCOUNTNUMBER", “1”);
String pServerHost = "pmgr.insco.com";
int pServerPort = 80;
```

```
//call accept payment from the Payment Manager
int primaryRC = 0;
int secondaryRC = 0;
try {
  server = new PaymentServerClient(null, pServerHost, pServerPort);
  PaymentServerResponse psResp =
        server.issueCommand(OP_ACCEPTPAYMENT, pairs, DbUser, DbPassword);
  primaryRC = psResp.getPrimaryRC();
  secondaryRC = psResp.getSecondaryRC();
} catch (PaymentServerAuthorizationException ex) {
  // handle error
} catch (PaymentServerCommunicationException ex) {
  // handle error
} catch (PaymentServerClientException ex) {
  // handle error
} finally {
  if (server !=null) try {server.close(); } catch (IOException e) {}
}
```

Setting up AS/400 WebSphere applications for B2B integration

The following subtopics describe how to configure WebSphere Application Server security, HTTP Server configuration, and how to configure the network and its associated resources.

Configuring WebSphere Application Server security

To enable security for the administrative server, follow these steps:

1. Add users that need to be authorized to an LDAP directory.

2. From the **Tasks** view in the WebSphere Administrative Console, expand the **Security** tree.

3. Select the **Specify Global Settings** item and click the **Start** icon.

4. Fill in the fields for each tabbed page as appropriate for your WebSphere Application Server environment. General information concerning the fields on these pages can be found in the **Security Properties help** page.

    a. On the **General** tab, check the **Enable Security** check box. You may also want to change the Security Cache Timeout value. We changed ours to 30 seconds.

    b. On the **Application Defaults** tab, set your **Realm Name** to your domain name (see item 1d of the “WebSphere security discoveries” section of the “INSCO application” topic). Select your **Challenge Type** to be **Custom**. In the **Login URL** and **Relogin URL** fields, specify the full path to your HTML file, which contains a form to request a user ID and password; that is:

http://www.insco.com/inscologin.html

c. On the **Authentication Mechanism** tab, specify the Authentication Mechanism to be **LTPA**. Generate keys for LTPA by pressing the **Generate Keys** button. Enter a password to be used in creating the keys. Also, enable **SSO** by entering a **Shared Name**, which can be anything you want. The Shared Name will be used as the name of the cookie that gets created. Also, enter the name of your Domain (see item 1d of the "WebSphere security discoveries" section of the "INSCO application" topic).

d. On the **User Registry** tab, fill in the fields corresponding to your LDAP setup. The following list represents the values that we used for the fields:

   i. Security Server ID: cn=administrator
   ii. Security Server Password: password for cn=administrator
   iii. Directory Type select the advanced button
      1) Change the User Filter from uid to cn
      2) Change the User ID Map from uid to cn
   iv. Host: sys.insco.com
   v. Base Distinguished Name: dc=sys,dc=insco,dc=com
   vi. Bind Distinguished Name: cn=administrator
   vii. Bind Password: password for cn=administrator

e. Click the **Finished** button.

5. Restart the administrative server.

Configuring HTTP Server for authorization services

To enable the use of authorization services for the HTTP Server configuration, follow these steps:

1. Open the properties file on the AS/400 using the following command on an AS/400 command line: EDTF STMF('instance.root/properties/bootstrap.properties').

   Note: Instance.root will most likely be '/QIBM/UserData/WebASAdv/default'.

2. In the bootstrap.properties file, locate the property `ose.security.enabled`.
3. Change the line to read `ose.security.enabled=true`.
4. Save the updated file.
5. Restart your HTTP Server instance.

Securing web resources

The following steps will enable you to secure web resources using the WebSphere Administrative Console:

1. From the **Tasks** view, expand the **Configuration** tree.

2. Configure a web application to hold the servlets and JSP files that you want to secure.

3. Expand the **Configuration** tree again.

4. Select the **Configure an Enterprise Application** option and click the **Start** icon.

    a. Specify a name for your enterprise application; then press the **Next** button.
    b. Expand the **Web Applications** tree, select the web application that you configured in step 2, click the **Add** button, click the **Next** button, and then click the **Finished** button.

5. Expand the **Security** tree.

6. Select the **Configure Application Security** option and click the **Start** icon.

    a. Expand the **Enterprise Applications** tree.
    b. Select the enterprise application configured in step 4 and click the **Next** button.
    c. Set your **Realm Name** to your domain name (see item 1d of the "WebSphere security discoveries" section of the "INSCO application" topic), and select your **Challenge Type** to be **Custom**. In the **Login URL** and **Relogin URL** fields, specify the full path to your HTML file, which contains a form to request user ID and password; that is:

        http://www.insco.com/inscologin.html

        and then click the **Finished** button.

7. Select the **Work With Method Groups** option and click the **Start** icon.

8. Select the **Configure Resource Security** option and click the **Start** icon.

    a. Expand the **Virtual Hosts** tree.
    b. Expand the **default_host** tree.
    c. Select the uniform resource identifier (URI) for the servlet or JSP file that you want to secure and press the **Next** button. You will only be able to select one URI at a time, so you may have to perform this step multiple times. Answer `Yes` to the **Use Default Method Groups** dialog box, and then press the **Finished** button.

9. Select the **Assign Permissions** option and click the **Start** icon.

a. To associate a user registry entry with a permission, select the permission (such as Application_Name-ReadMethods) and click **Add**. To disassociate a user registry entry, select the entry and click **Remove**.
b. Select either **All Authenticated** or **Selection**. Selection will give you the option of selecting groups of users or specific users.

Configuring the network

There are four distinct networks in this scenario: the INSCO intranet, the INSCO DMZ, the scenario internet, and the underwriter intranet. The following figures illustrate each network in detail.

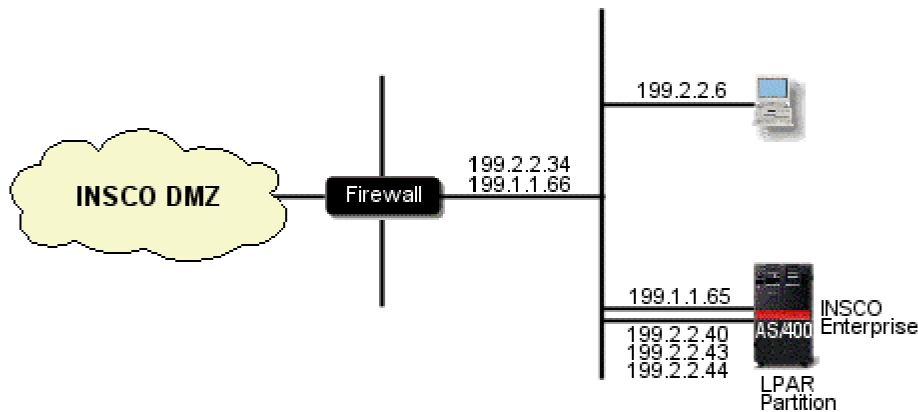


Figure 38. INSCO intranet

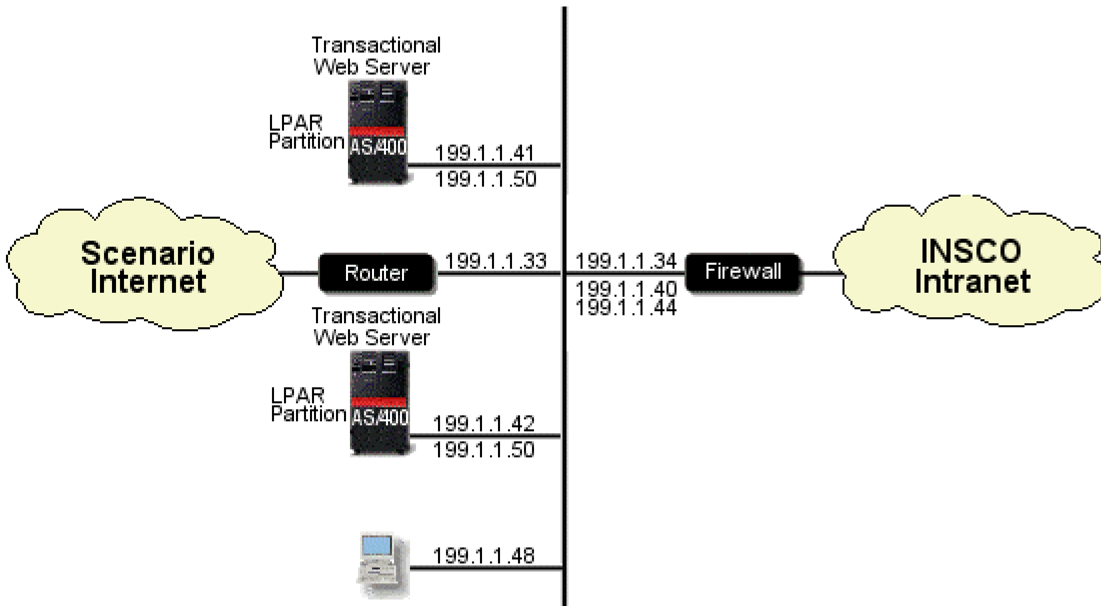


Figure 39. INSCO DMZ

The IP addresses, 199.1.1.40 and 199.1.1.44, were static mapped (using NAT) to INSCO internal addresses.

The IP address, 199.1.1.50, is a virtual IP address. It was created and activated on both transactional web server systems.

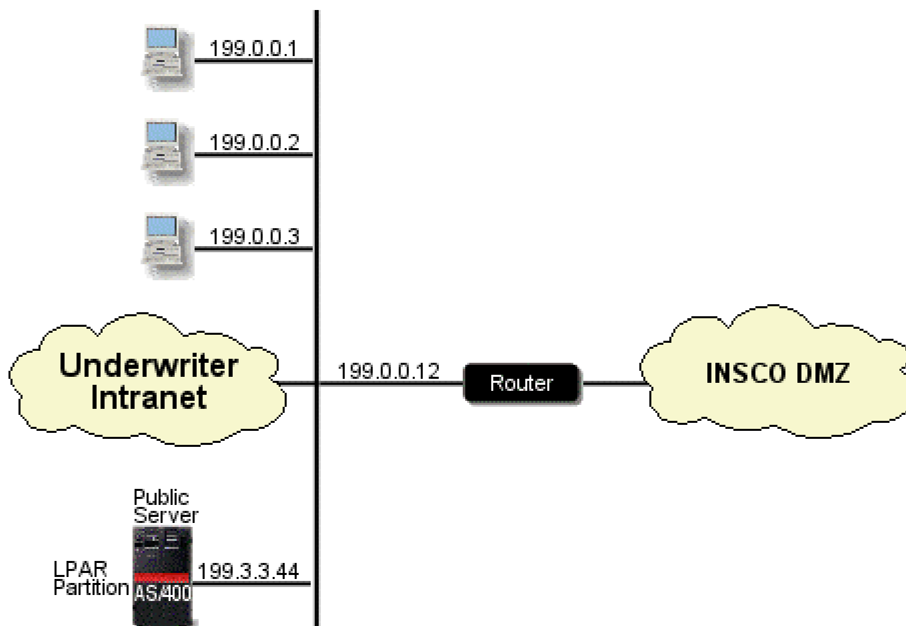


Figure 40. Scenario internet

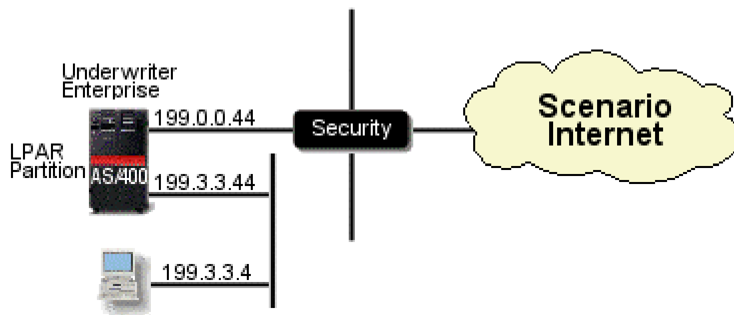


Figure 41. Underwriter intranet

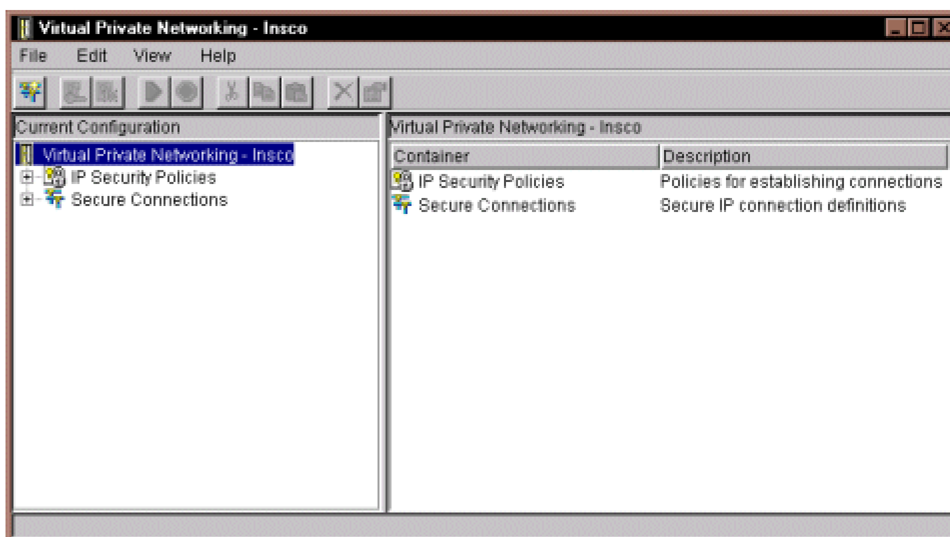
Configuring network security

This topic shows the VPN creation procedures, the VPN configuration details, and the VPN IP filter rules for the INSCO and the underwriter networks.

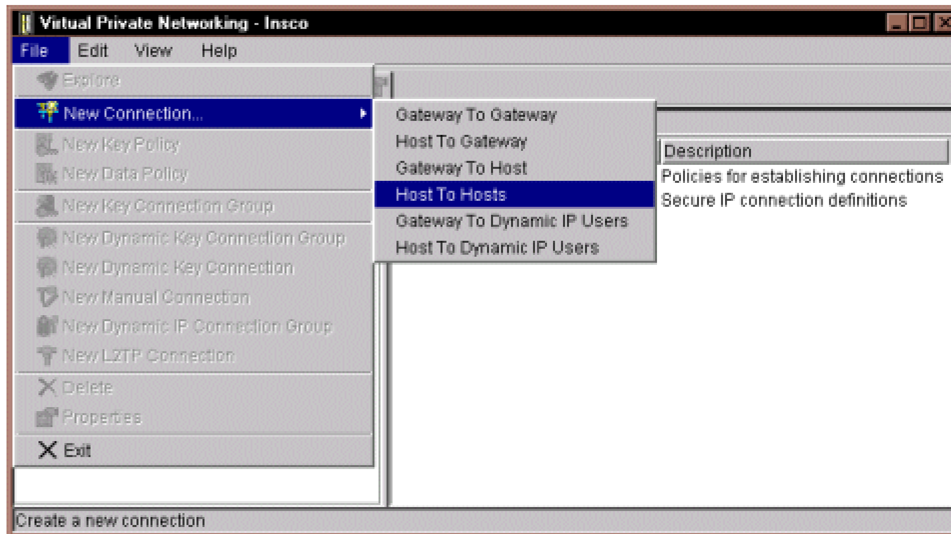
VPN creation steps (INSCO)

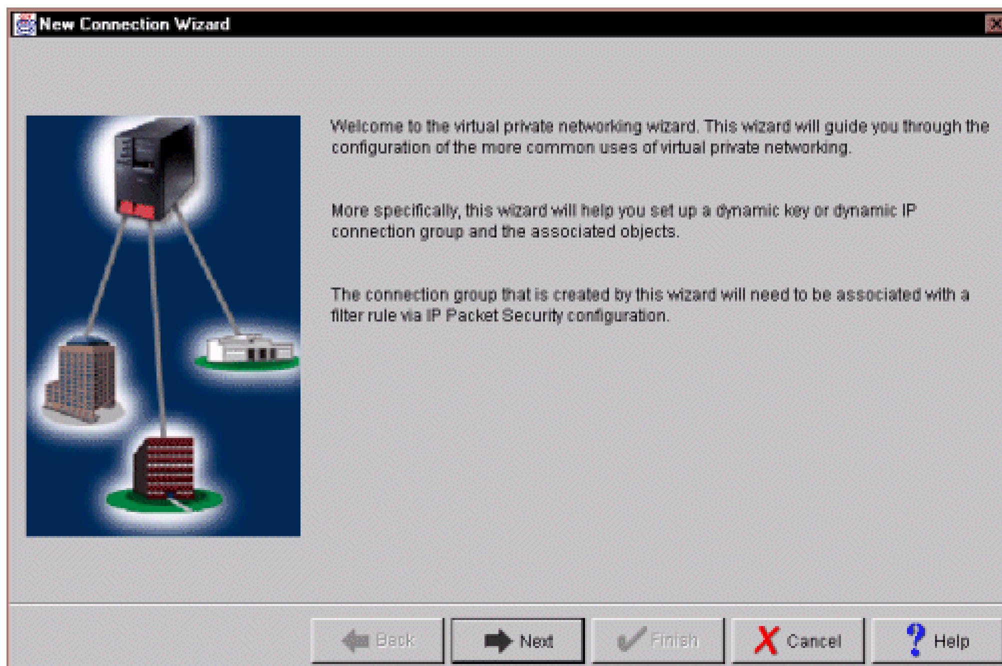
We created the INSCO VPN by using the New Connection Wizard - INSCO.

Starting at Operations Navigator, expand the **system** (INSCO) **Network** **IP Security**. Select **Virtual Private Networking** and then open **Configuration**.

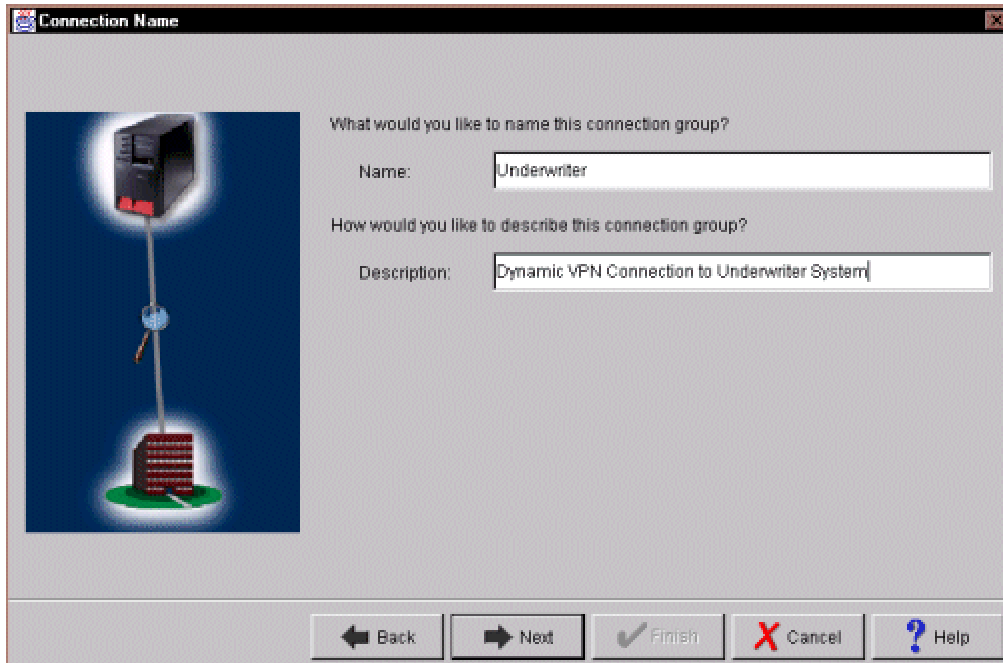


Select **File**, **New Connection**, and **Hosts to Hosts** to start the Connection Wizard.

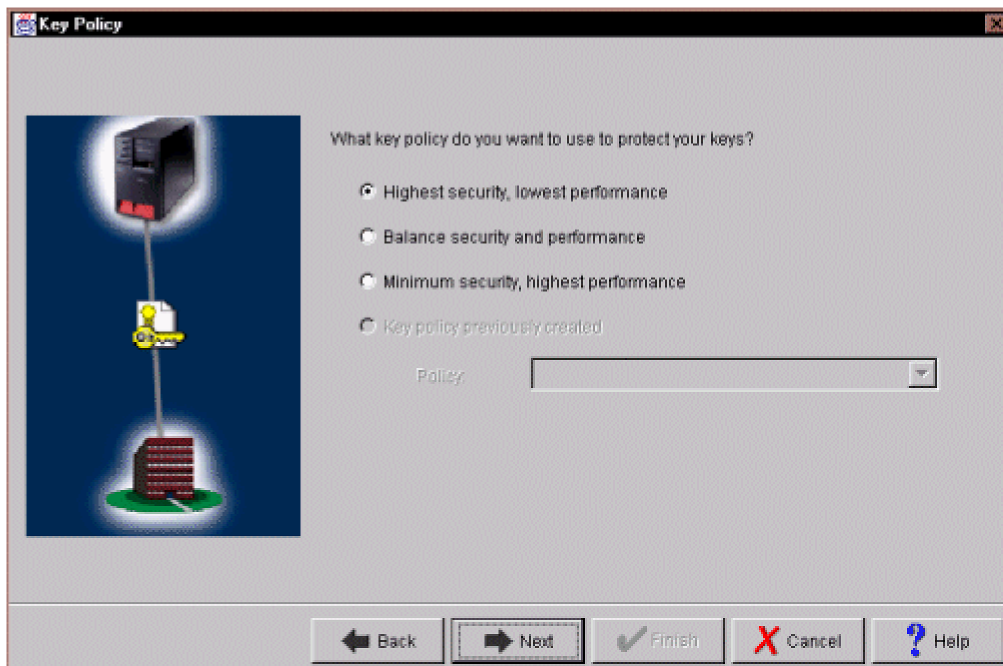




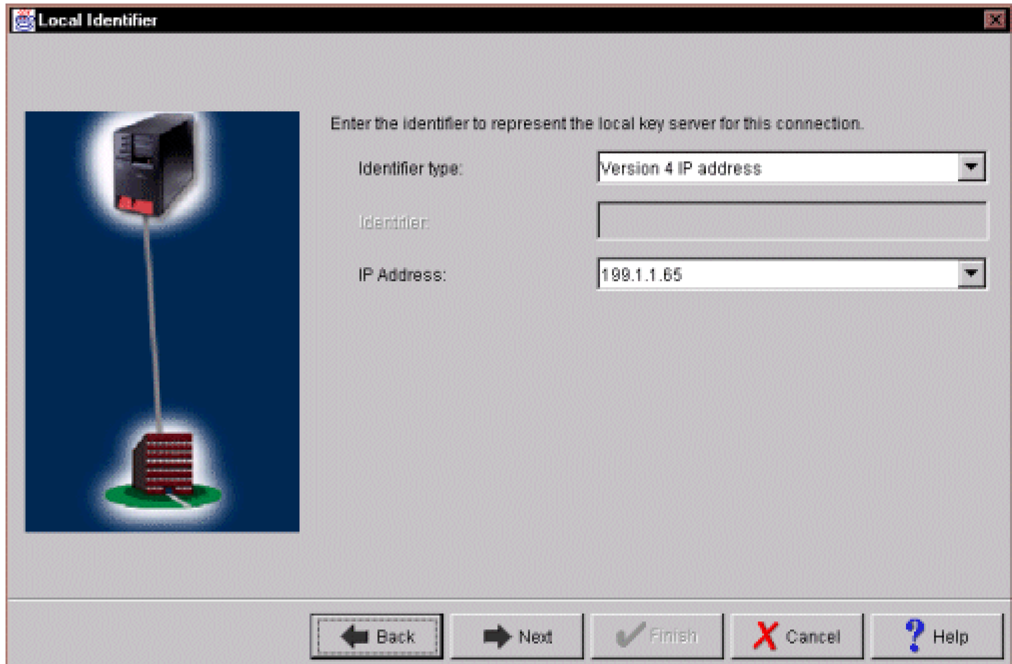
Specify a connection name and description.

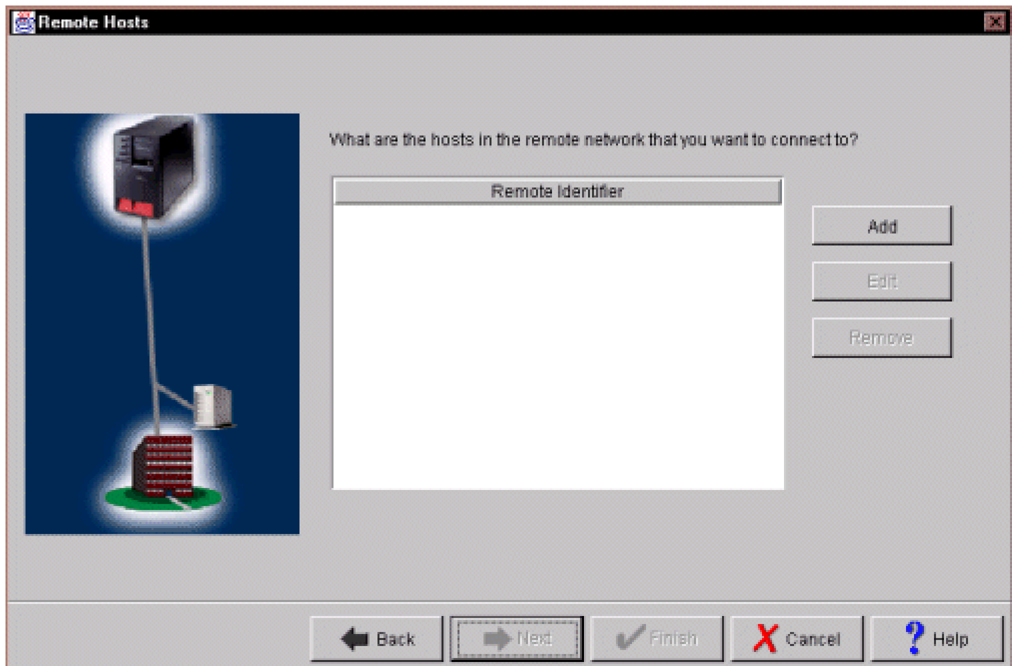


We chose the highest security, lowest performance selection for the key policy.

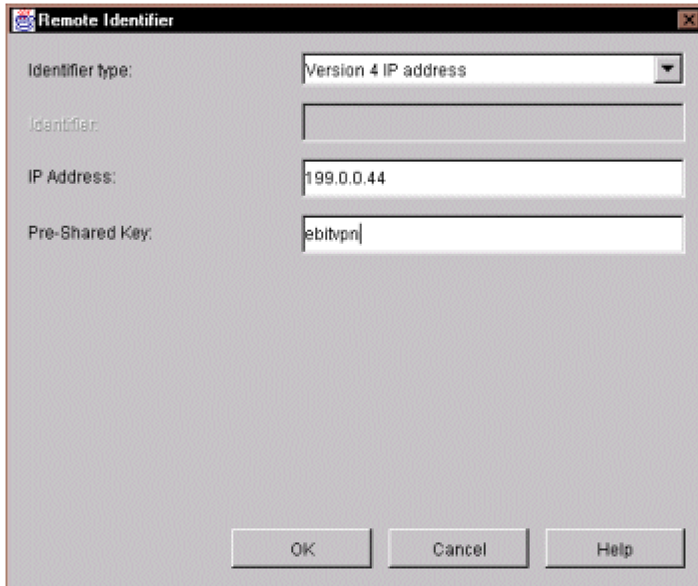


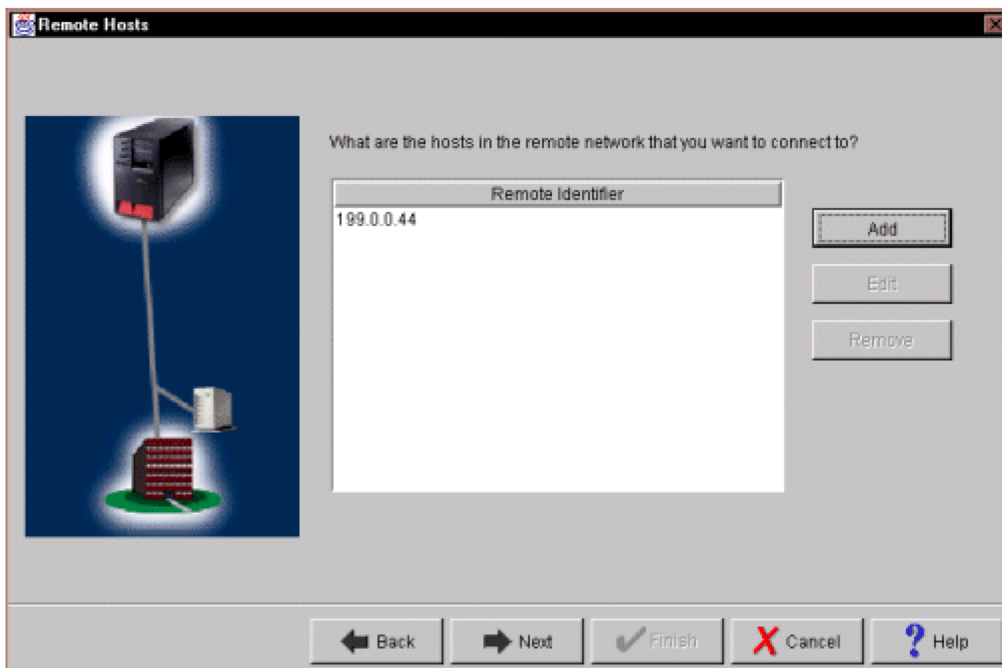
We specified the local identifier information. For a host-to-host connection, the VPN servers and data endpoints have the same IP addresses.



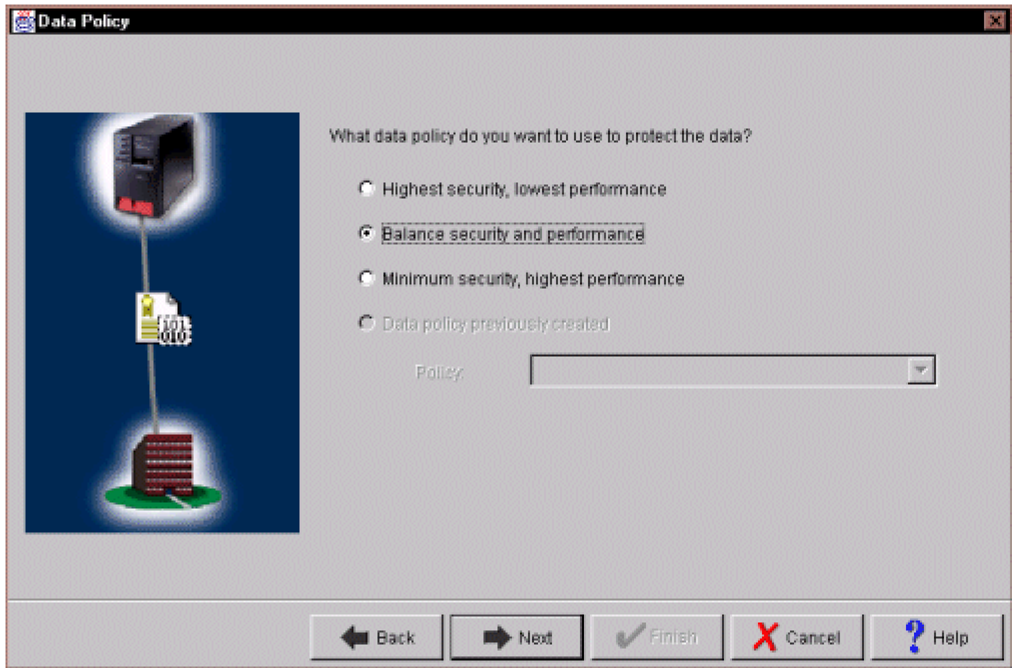


Next, we specified the remote VPN server and authentication information. Here we are using a pre-shared key.

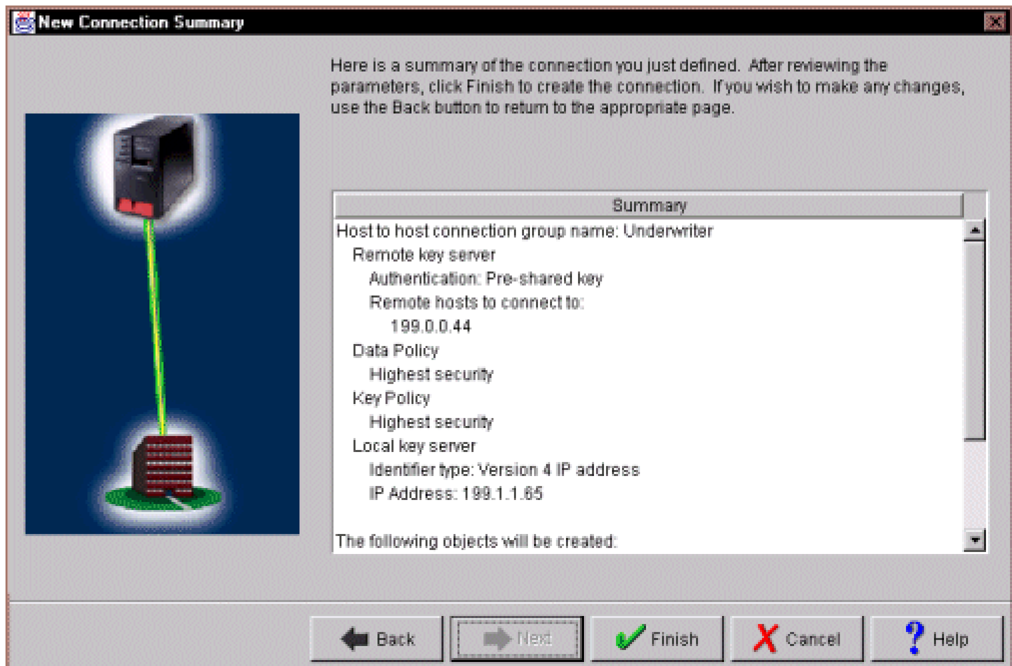




We chose to balance security and performance for our data policy.



Next, click the Finish button to create the VPN connection.



VPN configuration details (INSCO)

Key Policy

Name	UnderwriterHS
Initiator Negotiation	Aggressive Mode
Key Protection Transforms	
Authentication Method	Pre-shared Key
Pre-Shared Key Value	ebitvpn
Hash Algorithm	SHA
Encryption Algorithm	3DES-CBC
Diffie-Hellman Group	Default 768-bit MODP
Key Management	
Maximum Key Lifetime (minutes)	1440
Maximum Size Limit (kilobytes)	No size limit

Data Policy

Name	UnderwriterBS
Use Diffie-Hellman Perfect Forward Secrecy	No
Diffie-Hellman Group	
Authentication Method	Pre-shared Key
Data Protection Proposals	ESP
Encapsulation Mode	Transport
Protocol	ESP
Authentication Algorithms	HMAC-MD5
Encryption Algorithms	DES-CBC
Key Expiration	
Expire after (minutes)	60
Expire at size limit (kilobytes)	No size limit

Key Connection Group

Name	Underwriter
Remote Key Server	Yes
Identifier Type	Version 4 IP Address
IP Address	199.0.0.44
Key Policy	UnderwriterHS
Local Key Server	
Identifier Type	Version 4 IP Address
IP Address	199.1.1.65

Dynamic Key Group

Name	Underwriter
System Role	Both systems are hosts
Initiation	Either system can initiate this connection
Policy	Underwriter
Data Management Security Policy	UnderwriterBS
Connection Lifetime	Never expires
Local Address	Single value from connection
Local Ports	Connection
Remote Address	Single value from connection
Remote Ports	Connection
Protocol	Connection

Dynamic Key Connection

Name	Underwriter:L1
Remote Key Server	
Key Connection Group	Underwriter
Identifier	199.0.0.44
Start when TCP/IP is started?	Start Automatically
Local Address	199.1.1.65
Remote Address	199.0.0.44
Services	
Local Port	80
Remote Port	Any Port
Protocol	TCP

VPN IP filter rules (INSCO)

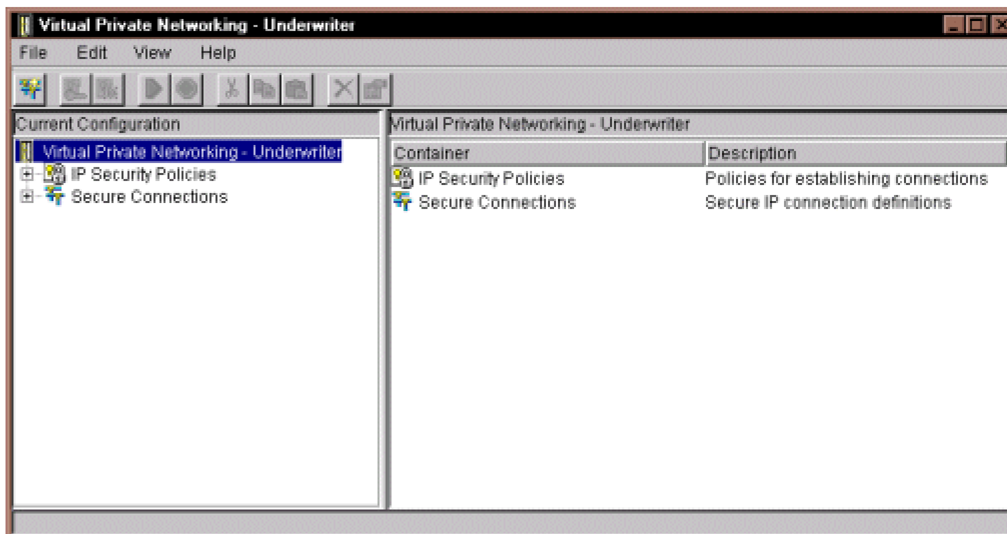
The IP filter rules for the INSCO VPN connection to the underwriter system follow.

Filter Set	Action	Direction	Source	Service	Destination	Fragment	Journaling
INSCOVPN	PERMIT	OUTBOUND	199.0.0.44	UDP	199.1.1.65	*	OFF
INSCOVPN	PERMIT	INBOUND	199.1.1.65	UDP	199.0.0.44	*	OFF
INSCOVPN	IPSEC	OUTBOUND	199.0.0.44	*	199.1.1.65	*	OFF

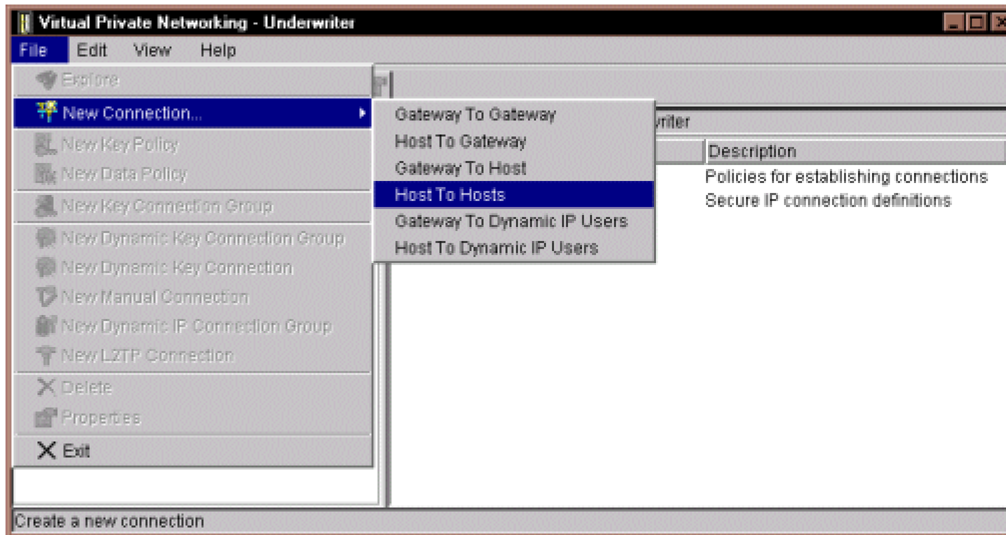
VPN creation steps (underwriter)

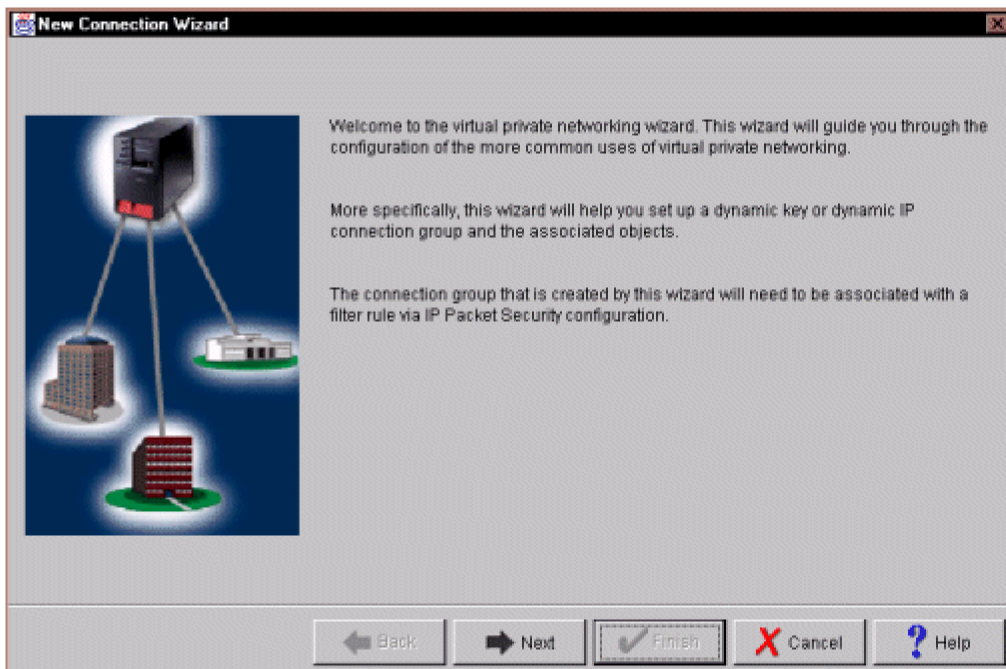
We took these steps to create the underwriter VPN.

Starting at Operations Navigator, expand the **system** (INSCO) **Network** **IP Security**. Select **Virtual Private Networking** and then open **Configuration**.



Select **File**, **New Connection**, and **Hosts to Hosts** to start the Connection Wizard.


Virtual Private Networking - Underwriter
File
Edit
View
Help
Explore
New Connection...
New Key Policy
New Data Policy
New Key Connection Group
New Dynamic Key Connection Group
New Dynamic Key Connection
New Manual Connection
New Dynamic IP Connection Group
New L2TP Connection
Delete
Properties
Exit
Gateway To Gateway
Host To Gateway
Gateway To Host
Host To Hosts
Gateway To Dynamic IP Users
Host To Dynamic IP Users
Description
Policies for establishing connections
Secure IP connection definitions
Create a new connection


New Connection Wizard
Welcome to the virtual private networking wizard. This wizard will guide you through the configuration of the more common uses of virtual private networking.
More specifically, this wizard will help you set up a dynamic key or dynamic IP connection group and the associated objects.
The connection group that is created by this wizard will need to be associated with a filter rule via IP Packet Security configuration.
Back
Next
Finish
Cancel
Help

Specify a connection name and description.

Connection Name

What would you like to name this connection group?

Name: INSCO

How would you like to describe this connection group?

Description: Dynamic VPN Connection to INSCO System

Back | Next | Finish | Cancel | Help

We chose the highest security, lowest performance selection for the key policy.

Key Policy

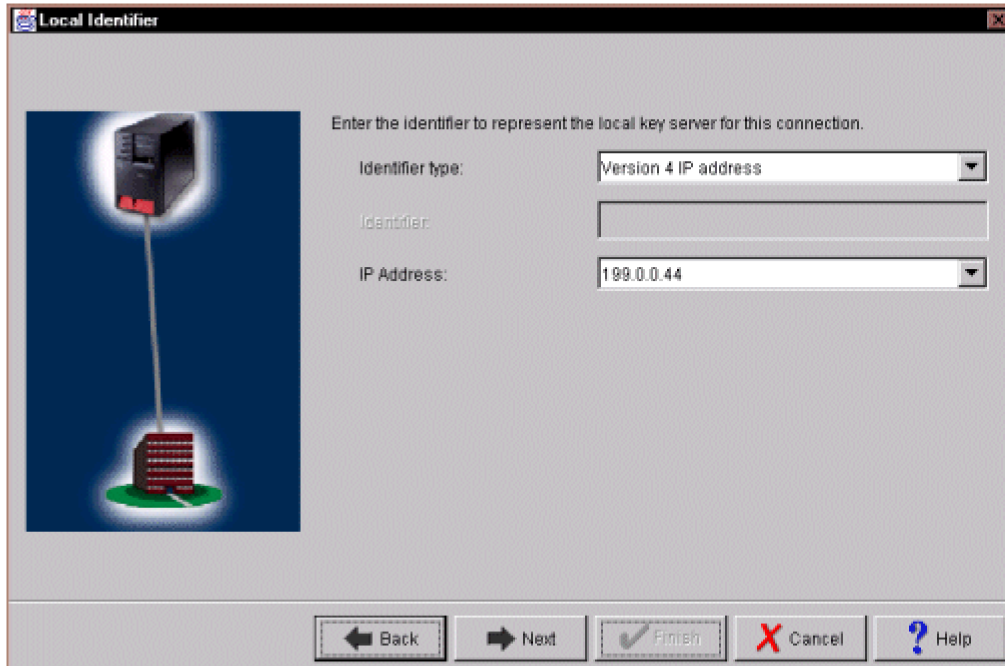
What key policy do you want to use to protect your keys?

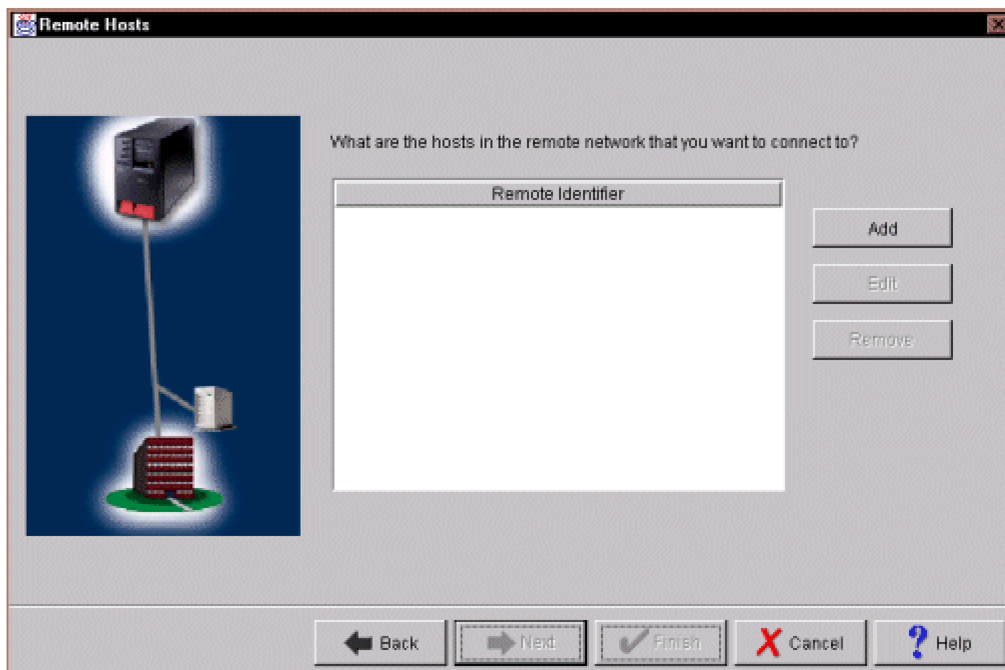
- (•) Highest security, lowest performance
- ( ) Balance security and performance
- ( ) Minimum security, highest performance
- ( ) Key policy previously created

Policy:

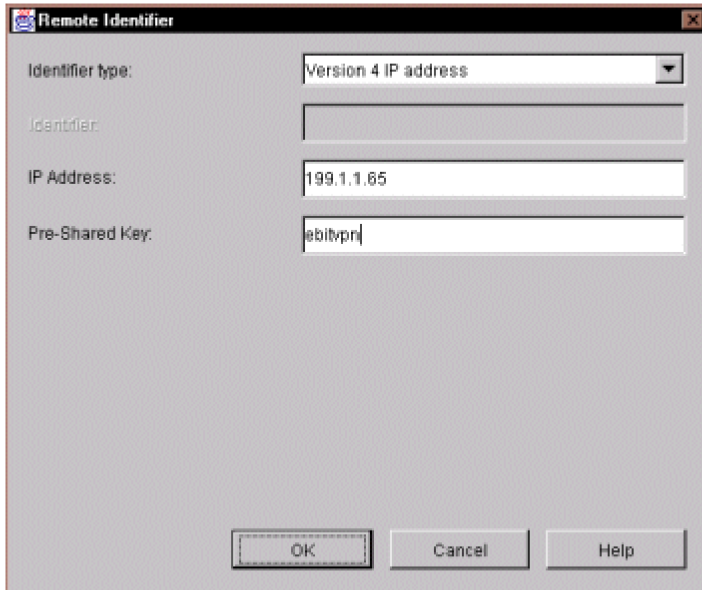
Back | Next | Finish | Cancel | Help

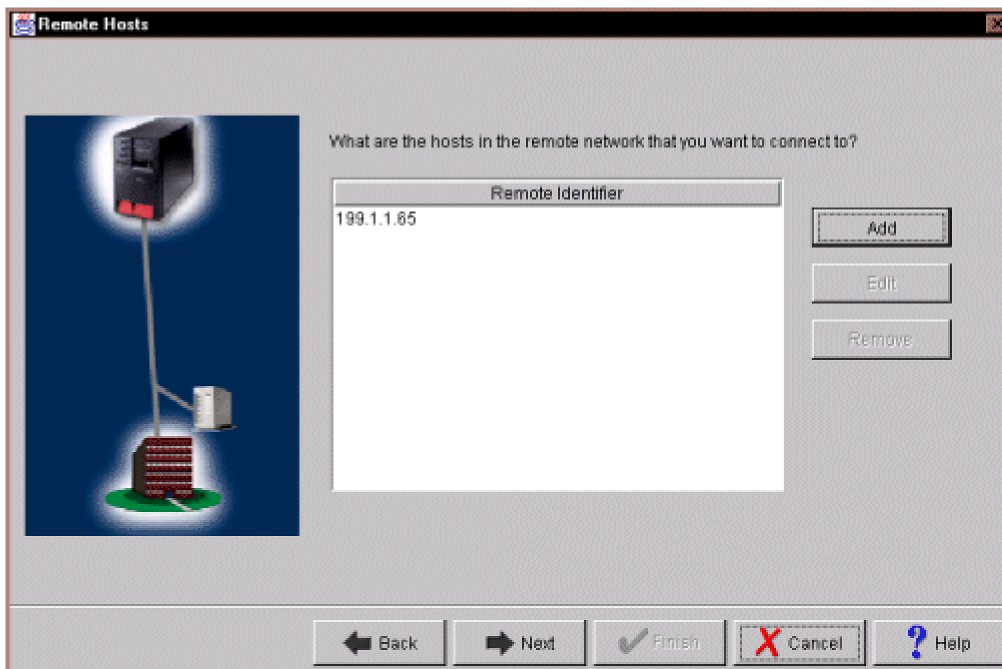
Next, we specified the remote VPN server and authentication information. Here we are using a pre-shared key.



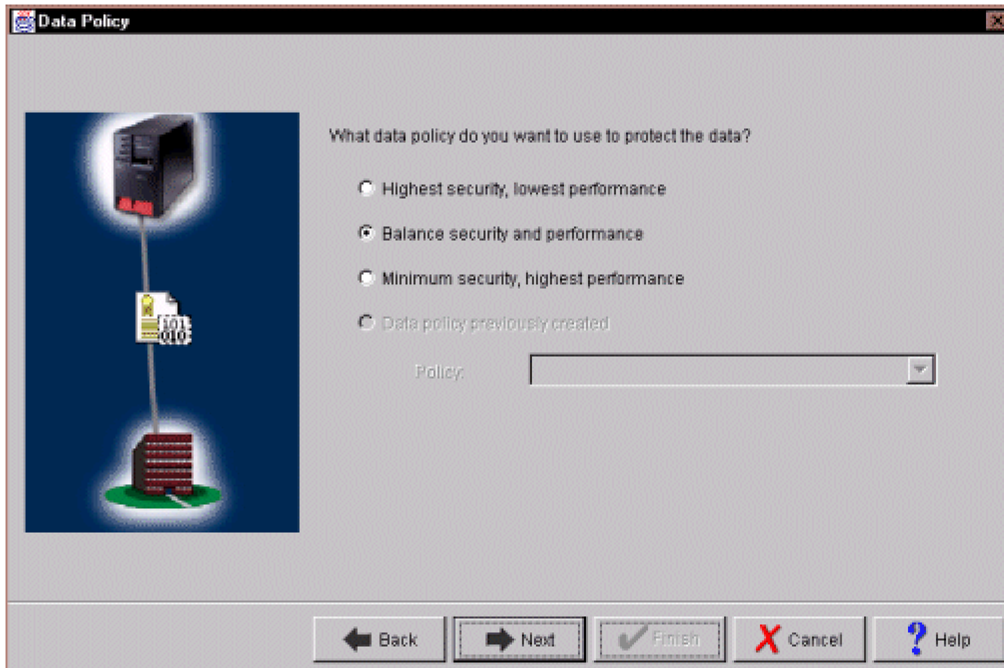


Next, we specified the remote VPN server and authentication information. Here we are using a pre-shared key.

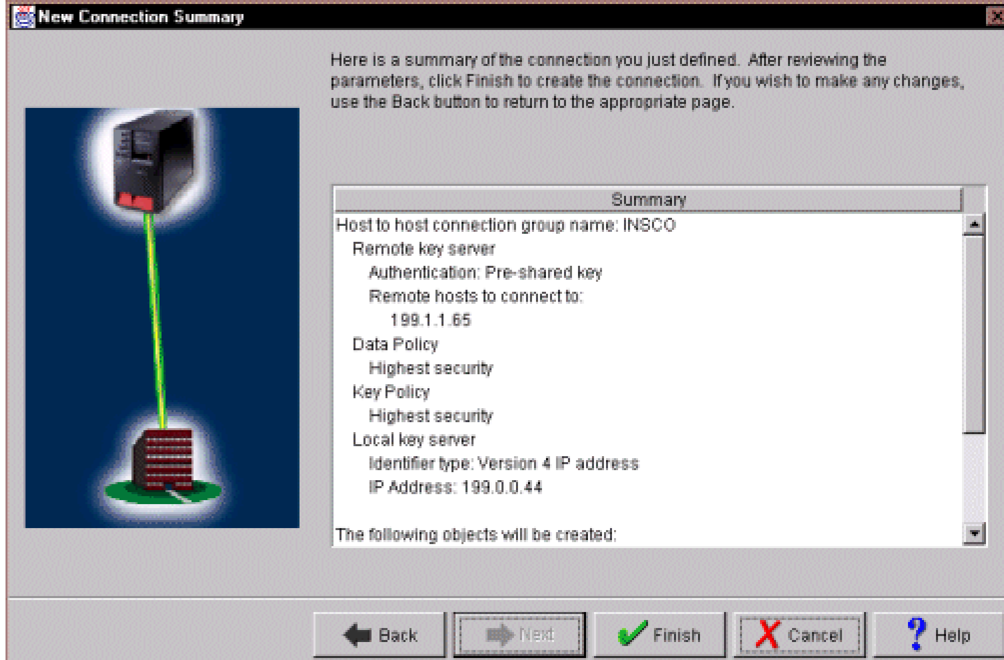




We chose to balance security and performance for our data policy.



Next, click the Finish button to create the VPN connection.



VPN configuration details (underwriter)

Key Policy	
Name	INSCOHS
Initiator Negotiation	Aggressive Mode
Key Protection Transforms	
Authentication Method	Pre-shared Key
Pre-Shared Key Value	ebitvpn
Hash Algorithm	SHA
Encryption Algorithm	3DES-CBC
Diffie-Hellman Group	Default 768-bit MODP
Key Management	
Maximum Key Lifetime (minutes)	1440
Maximum Size Limit (kilobytes)	No size limit

Data Policy	
Name	INSCOBS
Use Diffie-Hellman Perfect Forward Secrecy	No
Diffie-Hellman Group	
Authentication Method	Pre-shared Key
Data Protection Proposals	ESP
Encapsulation Mode	Transport
Protocol	ESP
Authentication Algorithms	HMAC-MD5
Encryption Algorithms	DES-CBC
Key Expiration	
Expire after (minutes)	60
Expire at size limit (kilobytes)	No size limit

Key Connection Group	
Name	INSCO
Remote Key Server	Yes
Identifier Type	Version 4 IP Address
IP Address	199.1.1.65
Key Policy	INSCOHS
Local Key Server	
Identifier Type	Version 4 IP Address
IP Address	199.0.0.44

Dynamic Key Group	
Name	INSCO
System Role	Both systems are hosts
Initiation	Either system can initiate this connection
Policy	INSCO
Data Management Security Policy	INSCOBS
Connection Lifetime	Never expires
Local Address	Single value from connection
Local Ports	Connection
Remote Address	Single value from connection
Remote Ports	Connection
Protocol	Connection

Dynamic Key Connection	
Name	INSCO:L1
Remote Key Server	
Key Connection Group	INSCO
Identifier	199.1.1.65
Start when TCP/IP is started?	Start Automatically
Local Address	199.0.0.44
Remote Address	199.1.1.65
Services	
Local Port	80
Remote Port	Any
Protocol	TCP

VPN IP filter rules (underwriter)

The IP filter rules for the underwriter VPN connection to the INSCO system follow.

Filter Set	Action	Direction	Source	Service	Destination	Fragment	Journaling
INSCOVPN	PERMIT	INBOUND	199.1.1.65	UDP	199.0.0.44	*	OFF
INSCOVPN	PERMIT	OUTBOUND	199.0.0.44	UDP	199.1.1.65	*	OFF
INSCOVPN	IPSEC	OUTBOUND	199.0.0.44	*	199.1.1.65	*	OFF

These rules are in addition to those documented in our last report.

HTTP port 80 - Source: INSCO DMZ Servers, Destination: INSCO Payment Manager

		Source		Destination				
Action	**Protocol**	**Operation**	**Port #**	**Operation**	**Port #**	**Interface**	**Routing**	**Direction**
permit	tcp	gt	1023	eq	80	Nonsecure	route	inbound
permit	tcp	gt	1023	eq	80	Secure	route	outbound
permit	tcp/ack	eg	80	gt	1023	Secure	route	inbound
permit	tcp/ack	eq	80	gt	1023	Nonsecure	route	outbound

HTTPS port 443 - Source: INSCO DMZ Servers, Destination: INSCO Internal Server

		Source		Destination				
Action	**Protocol**	**Operation**	**Port #**	**Operation**	**Port #**	**Interface**	**Routing**	**Direction**
permit	tcp	gt	1023	eq	443	Nonsecure	route	inbound
permit	tcp	gt	1023	eq	443	Secure	route	outbound
permit	tcp/ack	eg	443	gt	1023	Secure	route	inbound
permit	tcp/ack	gt	443	eq	1023	Nonsecure	route	outbound

LDAP port 389 - Source: INSCO DMZ Servers, Destination: INSCO Internal Server

		Source		Destination				
Action	**Protocol**	**Operation**	**Port #**	**Operation**	**Port #**	**Interface**	**Ruting**	**Direction**
permit	tcp	gt	1023	eq	389	Nonsecure	route	inbound
permit	tcp	gt	1023	eq	389	Secure	route	outbound
permit	tcp/ack	eg	389	gt	1023	Secure	route	inbound
permit	tcp/ack	eq	389	gt	1023	Nonsecure	route	outbound

Data queue port 8472 - Source: INSCO DMZ Servers, Destination: INSCO Internal Server

		Source		Destination				
Action	**Protocol**	**Operation**	**Port #**	**Operation**	**Port #**	**Interface**	**Routing**	**Direction**
permit	tcp	gt	1023	eq	8472	Nonsecure	route	inbound
permit	tcp	gt	1023	eq	8472	Secure	route	outbound
permit	tcp/ack	eg	8472	gt	1023	Secure	route	inbound
permit	tcp/ack	eq	8472	gt	1023	Nonsecure	route	outbound

Trademarks

IBM, AS/400, DB2, DB2 Universal Database, Operating System/400, OS/400, RS/6000, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation and/or its subsidiaries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

License and disclaimer

Every effort has been made to present a fair assessment of the product families discussed in this paper. The opinions and recommendations expressed in this paper are those of the authors, not necessarily those of IBM.

This material contains IBM copyrighted sample programming source code (“Sample Code”). IBM grants you a nonexclusive license to compile, link, execute, display, reproduce, distribute and prepare derivative works of this Sample Code. The Sample Code has not been thoroughly tested under all conditions. IBM, therefore, does not guarantee or imply its reliability, serviceability, or function. IBM provides no program services for the Sample Code.

All Sample Code contained herein is provided to you "AS IS" without any warranties of any kind. **THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**

COPYRIGHT

(C) Copyright IBM CORP. 2000
All rights reserved.
US Government Users Restricted Rights -
Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.