

Code Name Indigo

Code Name Indigo: A Guide to Developing and Running Connected Systems with Indigo

[Don Box](#)



This article was based on a pre-PDC build of Microsoft Windows Code Name "Longhorn" and all information contained herein is subject to change.

Note: This document was developed prior to the product's release to manufacturing and, as such, we cannot guarantee that any details included herein will be exactly the same as those found in the shipping product. The information represents the product at the time this document was printed and should be used for planning purposes only. Information subject to change at any time without prior notice.

SUMMARY This article describes a collection of new programming frameworks that are part of "Longhorn," the upcoming version of Windows. "Indigo," the code name for this framework, provides rich support for service-oriented design that is complementary to traditional object-oriented approaches. Indigo marries the best features of .NET Remoting, ASMX, and .NET Enterprise Services into a unified programming and administration model. Indigo's deep support for standard protocols, including HTTP, XML, and SOAP, makes it easier to integrate applications and services without sacrificing security or reliability.

The upcoming version of Microsoft® Windows® code-named "Longhorn" will include a unified programming model and communications infrastructure for developing connected systems. This infrastructure, code-named "Indigo," simplifies development through a service-oriented programming model in which autonomous programs are composed using asynchronous message passing. To enable this programming model, Indigo provides a rich set of technologies for creating, consuming, processing, and transmitting messages.

One of the key benefits of Indigo is that it provides a unified programming model and protocol stack for all common language runtime (CLR)-based remoting technologies. Indigo marries the best of .NET Remoting, ASMX, System.Messaging, and .NET Enterprise Services into a single framework, allowing developers to freely combine features such as declarative transactions, pluggable interceptors and transports, and XML Schema-based serialization.

Moreover, Indigo unifies a wide range of transports (HTTP, TCP, UDP, IPC), security mechanisms (public and symmetric keys, certificates), topologies (point-to-point, end-to-end through intermediaries, peer-to-peer, publish-and-subscribe), and assurances (transacted, reliable, durable) enabling Indigo to provide rich connectivity to many existing systems.

Indigo makes service-oriented programming viable in a broad spectrum of mainstream applications. By taking advantage of various facilities of both the CLR and Windows, Indigo can be used in performance-sensitive situations such as single-host and even single-process integration. This scale-invariance makes Indigo-based services accessible to a broader range of deployment options than current technologies.

Because interoperability and reach are important, Indigo has deep support for both XML and SOAP, as well as for the emerging advanced Web Service protocols that add security, reliability, and transactions to SOAP-based applications.

As part of Windows Longhorn, Indigo is available to every Longhorn application. Indigo will also be available as a separate download for Windows XP and Windows Server™ 2003. A small number of advanced features or optimizations of Indigo may only be available on Longhorn due to intrinsic differences between the various operating systems. However, Indigo provides the developer with a consistent service-oriented programming model no matter which operating system it is deployed on.

Service-orientation Fundamentals

Indigo is based on the principles of service-oriented architecture and programming. Service-orientation is an important complement to object-orientation that applies the lessons learned from component software, message-oriented middleware and distributed object computing. Service-orientation differs from object-orientation primarily in how it defines the term "application." Object-oriented development focuses on applications that are built from interdependent class libraries. Service-oriented development focuses on systems that are built from a set of autonomous services. This difference has a profound impact on the assumptions one makes about the development experience.

In Indigo, a service is simply a program that one interacts with via message exchanges. A set of deployed services is a system. Individual services are built to last—the availability and stability of a given service is critical. The

aggregate system of services is built to allow for change—the system must adapt to the presence of new services that appear a long time after the original services and clients have been deployed, and these must not break functionality.

Service-oriented development is based on the four fundamental tenets that follow:

Boundaries are explicit A service-oriented application often consists of services that are spread over large geographical distances, multiple trust authorities, and distinct execution environments. The cost of traversing these various boundaries is nontrivial in terms of complexity and performance. Service-oriented designs acknowledge these costs by putting a premium on boundary crossings. Because each cross-boundary communication is potentially costly, service-orientation is based on a model of explicit message passing rather than implicit method invocation. Compared to distributed objects, the service-oriented model views cross-service method invocation as a private implementation technique, not as a primitive construct—the fact that a given interaction may be implemented as a method call is a private implementation detail that is not visible outside the service boundary.

Though service-orientation does not impose the RPC-style notion of a network-wide call stack, it can support a strong notion of causality. It is common for messages to explicitly indicate which chain(s) of messages a particular message belongs to. This indication is useful for message correlation and for implementing several common concurrency models.

The notion that boundaries are explicit applies not only to inter-service communication but also to inter-developer communication. Even in scenarios in which all services are deployed in a single location, it is commonplace for the developers of each service to be spread across geographical, cultural, and/or organizational boundaries. Each of these boundaries increases the cost of communication between developers. Service orientation adapts to this model by reducing the number and complexity of abstractions that must be shared across service boundaries. By keeping the surface area of a service as small as possible, the interaction and communication between development organizations is reduced. One theme that is consistent in service-oriented designs is that simplicity and generality aren't a luxury but rather a critical survival skill.

Services are autonomous Service-orientation mirrors the real world in that it does not assume the presence of an omniscient or omnipotent oracle that has awareness and control over all parts of a running system. This notion of service autonomy appears in several facets of development, the most obvious place being the area of deployment and versioning.

Object-oriented programs tend to be deployed as a unit. Despite the Herculean efforts made in the 1990s to enable classes to be independently deployed, the discipline required to enable object-oriented interaction with a component proved to be impractical for most development organizations. When coupled with the complexities of versioning object-oriented interfaces, many organizations have become extremely conservative in how they roll out object-oriented code. The popularity of the XCOPY deployment and private assemblies capabilities of the .NET Framework is indicative of this trend.

Service-oriented development departs from object-orientation by assuming that atomic deployment of an application is the exception, not the rule. While individual services are almost always deployed atomically, the aggregate deployment state of the overall system/application rarely stands still. It is common for an individual service to be deployed long before any consuming applications are even developed, let alone deployed into the wild. Amazon.com is one example of this build-it-and-they-will-come philosophy. There was no way the developers at Amazon could have known the multitude of ways their service would be used to build interesting and novel applications.

It is common for the topology of a service-oriented application to evolve over time, sometimes without direct intervention from an administrator or developer. The degree to which new services may be introduced into a service-oriented system depends on both the complexity of the service interaction and the ubiquity of services that interact in a common way. Service-orientation encourages a model that increases ubiquity by reducing the complexity of service interactions. As service-specific assumptions leak into the public facade of a service, fewer services can reasonably mimic that facade and stand in as a reasonable substitute.

The notion of autonomous services also impacts the way failures are handled. Objects are deployed to run in the same execution context as the consuming application. Service-oriented designs assume that this situation is the exception, not the rule. For that reason, services expect that the consuming application can fail without notice and often without any notification. To maintain system integrity, service-oriented designs use a variety of techniques to deal with partial failure modes. Techniques such as transactions, durable queues, and redundant deployment and failover are quite common in a service-oriented system.

Because many services are deployed to function over public networks (such as the Internet), service-oriented development assumes not only that incoming message data may be malformed but also that it may have been

transmitted for malicious purposes. Service-oriented architectures protect themselves by placing the burden of proof on all message senders by requiring applications to prove that all required rights and privileges have been granted. Consistent with the notion of service autonomy, service-oriented architectures invariably rely on administratively managed trust relationships in order to avoid per-service authentication mechanisms common in classic Web applications.

Services share schema and contract, not class Object-oriented programming encourages developers to create new abstractions in the form of classes. Most modern development environments not only make it trivial to define new classes, modern IDEs do a better job guiding you through the development process as the number of classes increases (as features like IntelliSense® provide a more specific list of options for a given scenario).

Classes are convenient abstractions as they share both structure and behavior in a single named unit. Service-oriented development has no such construct. Rather, services interact based solely on schemas (for structures) and contracts (for behaviors). Every service advertises a contract that describes the structure of messages it can send and/or receive as well as some degree of ordering constraints over those messages. This strict separation between structure and behavior vastly simplifies deployment, as distributed object concepts such as marshal-by-value require a common execution and security environment which is in direct conflict with the goals of autonomous computing.

Services do not deal in types or classes per se; rather, only with machine readable and verifiable descriptions of the legal "ins and outs" the service supports. The emphasis on machine verifiability and validation is important given the inherently distributed nature of how a service-oriented application is developed and deployed. Unlike a traditional class library, a service must be exceedingly careful about validating the input data that arrives in each message. Basing the architecture on machine-validatable schema and contract gives both developers and infrastructure the hints they need to protect the integrity of an individual service as well as the overall application as a whole.

Because the contract and schema for a given service are visible over broad ranges of both space and time, service-orientation requires that contracts and schema remain stable over time. In the general case, it is impossible to propagate changes in schema and/or contract to all parties who have ever encountered a service. For that reason, the contract and schema used in service-oriented designs tend to have more flexibility than traditional object-oriented interfaces. It is common for services to use features such as XML element wildcards (like `xsd:any`) and optional SOAP header blocks to evolve a service in ways that do not break already deployed code.

Service compatibility is determined based on policy Object-oriented designs often confuse structural compatibility with semantic compatibility. Service-orientation deals with these two axes separately. Structural compatibility is based on contract and schema and can be validated (if not enforced) by machine-based techniques (such as packet-sniffing, validating firewalls). Semantic compatibility is based on explicit statements of capabilities and requirements in the form of policy.

Every service advertises its capabilities and requirements in the form of a machine-readable policy expression. Policy expressions indicate which conditions and guarantees (called assertions) must hold true to enable the normal operation of the service. Policy assertions are identified by a stable and globally unique name whose meaning is consistent in time and space no matter which service the assertion is applied to. Policy assertions may also have parameters that qualify the exact interpretation of the assertion. Individual policy assertions are opaque to the system at large, which enables implementations to apply simple propositional logic to determine service compatibility.

What is Indigo?

Indigo is a set of .NET Framework-based technologies for building and running connected systems. Indigo is the next step in the evolutionary path that started with COM, COM+, and MSMQ, and continues through .NET Remoting, ASMX, System.Messaging, and .NET Enterprise Services. Indigo subsumes these technologies and offers a single unified programming experience for developing services using any CLR-compliant language.

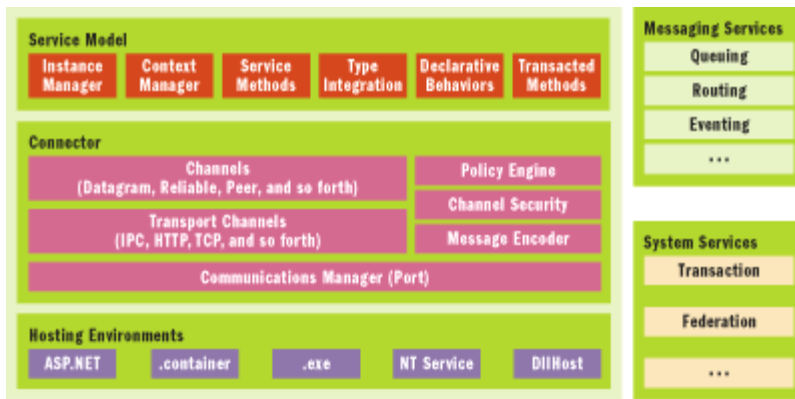


Figure 1 Indigo Architecture

As shown in **Figure 1**, the major subsystems of Indigo are the Indigo service model, the Indigo connector, hosting environments, and system and messaging services.

Indigo service model The Indigo service model makes service-oriented development explicit and simple from any CLR-targeted language. Developers use declarative attributes to mark up which aspects of their type should form the external contract of a service. The Indigo service model supports two kinds of contracts: data contracts and service contracts. Data contracts are structural in nature and describe the external format of a CLR type. A data contract is analogous to an XML Schema definition. A service contract is behavioral in nature and describes the pattern of message exchanges that are used to interact with a service. Service contracts are analogous to WSDL portType definitions. The Indigo service model supports both code-first and contract-first development models, including support for Web Service Description Language (WSDL) and XML Schema import/export.

The primary function of the Indigo service model is to associate user-defined code with incoming messages. This is accomplished using the [ServiceMethod] attribute. Methods marked with this attribute support either one-way messaging or correlated request/reply messaging, depending on the value of the OneWay property. Methods marked as service methods can be declared to work in terms of the entire message or (if desired) in terms of typed parameters. The messages themselves can be viewed as either strongly-typed objects or as XML information sets (infosets).

The Indigo service model provides instance and context management for a service. Indigo routes incoming messages to instances of user-defined service types. Developers use declarative attributes to control how instances are associated with incoming messages as well as how the session management feature of Indigo routes multiple messages to a common session object. Additionally, the Indigo service model provides developers with explicit control over the propagation of execution context (causality, call context, transactions) between cooperative services.

Finally, the Indigo service model provides declarative behaviors that automate security and reliability. These attributes influence the behavior of Indigo as methods enter and leave a service, enforcing the policies that are initially declared by the developer and then made concrete through configuration by the administrator.

Like ASMX, Indigo enables services to be consumed without sharing assemblies or source code. Additionally, Indigo provides support for ASMX compatibility that will allow many ASMX-based Web Services to become Indigo-based services with relatively little modification.

The Indigo connector The Indigo connector is a managed framework that provides secure and reliable message-based connectivity. The Indigo connector is a layered I/O framework that is based on the SOAP data and processing model. The Indigo connector allows you to build message-based applications independent of transport or target platform using a small number of concepts (port, message, channel) all of which are described later in this article. The Indigo connector is designed to have a small footprint and excellent throughput and latency characteristics, which enables it to be used in performance-sensitive systems.

The Indigo connector is based on ports and channels. A port represents a location in network space and provides a base Uniform Resource Identifier (URI) for all services that it hosts. A channel is an I/O device that services use to send and receive messages through their port. Some channels are transport channels that map underlying communication mechanisms to Indigo. Other channels are messaging channels that provide a particular messaging discipline or quality of service on top of one or more transport channels. The connector also includes a message encoder that allows encodings to be added to the system to translate between the abstract data model used internally by Indigo and byte sequences used by a specific transport. The serialization layer takes the world of CLR

objects and classes and distills them down to a simple data model that is suitable for sharing across the service boundaries.

The Indigo connector has explicit support for intermediaries that include firewalls, proxies, and gateways. The intermediary model of the Indigo connector is consistent with the SOAP processing model. While direct communication with a service is supported, Indigo assumes that a direct connection is the exceptional circumstance, not the rule. Instead, Indigo adapts to configuration, metadata, and service policy to determine the best route for a message and will take advantage of gateways or proxies as needed. Moreover, the Indigo connector vastly simplifies the creation of application-level gateways that can offload work from back-end services by providing message validation, security enforcement, and content-based load balancing.

The presence of intermediaries often means that there is no single end-to-end transport connection between the sender and ultimate receiver. To ensure that message delivery is reliable in these situations, Indigo implements end-to-end reliable messaging over SOAP using the WS-ReliableMessaging protocol. The Indigo connector provides two modes of reliable messaging: express and durable. Express messaging uses in-memory buffers to hold messages in transit as they travel from the source to the destination. There are no guarantees that an express message will be sent if the sender crashes immediately after sending the message. In contrast, durable messages use on-disk buffers and provide delivery guarantees even in the face of system crashes.

The Indigo connector provides intrinsic support for secure messaging that provides end-to-end protection independent of message transport. Both message-based and session-based modes are supported. Session-based security is the preferred model as both signing and encryption can use a more lightweight session key that is negotiated as part of session establishment. Message-based security is needed for disconnected or datagram scenarios in which the receiver of the message is not available at the time of transmission. Both modes support the WS-Security family of protocols (WS-Trust and WS-Federation), ensuring interoperability with non-Indigo Web Services even in the face of intermediaries. Indigo also takes advantage of transport-level security when possible (for example, when a direct TCP/Secure Sockets Layer (SSL) connection between sender and receiver is used).

Hosting environments The Indigo connector and service model can be explicitly loaded and used in any AppDomain on the system. This makes Indigo well suited for exposing a service-oriented facade to any piece of managed code. Initializing the Indigo connector in your AppDomain provides you with basic "service dialtone" independent of how your AppDomain is created. In fact, how your AppDomain is started is completely outside the scope of the role of the Indigo connector.

To make Indigo accessible to the widest possible range of developers, various hosting systems currently used on the Windows platform (such as svchost.exe and dllhost.exe) have been adapted to specifically support Indigo-based services. The most interesting of these hosting systems is ASP.NET and IIS.

With the release of Indigo, ASP.NET now provides a system-wide facility for managing AppDomains and OS processes. This allows Indigo-based services to be demand-started based on messages that arrive on any properly configured transport. Indigo has built-in support for TCP, HTTP, and Interprocess Communication (IPC) transports. Indigo hosting allows applications to be associated with URI suffixes much like the IIS model for mapping URI suffixes to VRoots. Unlike IIS, however, Indigo is not tied to a particular transport protocol or programming model.

Indigo-based services that are hosted using ASP.NET gain an additional set of management services, including configurable recycling and passivation both at the process/AppDomain level as well as at the individual service level. Indigo also supports basic health monitoring for processes, and AppDomains. A notable new feature to ASP.NET's hosting facilities is the ability for service code to participate in service management (start, stop) requests. This helps services that have intermediate work avoid process recycling at inopportune moments. Most new server-based Indigo services are expected to use ASP.NET hosting, which results in a common configuration and management experience for all managed middle-tier code.

System and messaging services An Indigo-based system has several system services that provide essential functionality to all services. One of these services is identity federation. An upcoming version of the security system in Windows supports a federation service that allows identities from foreign trust boundaries to be managed and validated. The Windows federation service uses WS-Federation to securely broker the authentication between the service and the corresponding trust authority.

Indigo also provides significant support for transactional programming. Indigo-enabled versions of Windows support a service-based transaction manager that may be accessed via the System.Transactions framework or the WS-AtomicTransactions protocol. The new System.Transactions framework makes transactional programming simple and efficient throughout the platform (it supports SQL Server™, ADO.NET, MSMQ, distributed transaction coordinator (DTC), etc). System.Transactions supports both an explicit programming model based on the ITransaction interface as well as an implicit programming model in which transactions are automatically managed by Indigo. Both models

are available to Indigo-based apps.

System.Transactions provides a new in-memory transaction manager that allows volatile transactions (that is, transactions that do not involve durable resources) to commit or roll back efficiently without incurring any disk I/O. To support durability, the in-memory transaction manager will transparently promote volatile transactions to durable transactions by coordinating through a disk-based transaction manager like the DTC the moment a durable resource manager enlists itself with a transaction.

System.Transactions defines a simple managed interface for writing both volatile and durable resource managers. System.Transactions also supports any resource manager that can speak either OLE transactions or the broadly adopted WS-AtomicTransaction protocol. To enable efficient transacted access to the file system, the Longhorn version of System.Transactions directly supports both the Kernel Transaction Manager (KTM) and the Transactional NTFS (TxNTFS).

Indigo also provides implementations of many of the common abstractions used in messaging applications. Specifically, Indigo provides simple yet extensible versions of queues, eventing, and content-based routing that can be used in any Indigo-based app.

Programming Indigo

The Indigo model is based on four simple concepts. First, a message is a transmissible piece of data that adheres to the SOAP data model. Messages have zero or more headers and a body and in Indigo are represented by the concrete Message class. Though Indigo allows objects with arbitrary methods to be stored in a message, all message parts may be interacted with uniformly according to the SOAP and XML data models.

Second, messages flow between ports. Indigo ports are endpoints for communications between services. A port can send and receive messages independent of transport. Ports come in two flavors: anonymous and named. Named ports can be explicitly addressed independent of any session or connection. Anonymous ports can only be addressed implicitly as part of an established communication session. The only messages an anonymous port will receive are responses to messages sent from that port.

Third, a service is an opaque piece of code behind a port. Indigo refers to the executable code that resides behind a port as a service. Services have both policy and contract; policy indicates the capabilities and requirements of the service and contract indicates the acceptable message exchanges that the service can participate in.

Fourth, services communicate with their ports via channels. Services can create and use a range of channels to build an application; the properties, policies, and behaviors of a channel vary based on the characteristics of the communication services the channel provides. For example, a dialog channel can be created that connects another service (identified by a URL). The dialog channel provides the ability to set a transfer assurance policy of "exactly-once, in-order" delivery; such a channel can be used to ensure messages are delivered reliably.

The diagram in **Figure 2** shows the relationship between messages, ports, channels, and services.

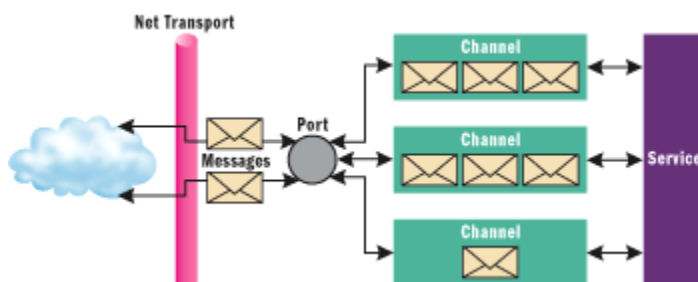


Figure 2 Relationships

The easiest way to make these concepts concrete is to look at some code. The code shown in this section is written against the PDC build of Indigo—naturally the details are subject to change between now and the final release. Here is the smallest possible Indigo service written in C# using only the Indigo connector:

```
using System;
using System.MessageBus;

class app {
    static void Main() {
        // create and open a named port
        Port port = new Port(new
            Uri("soap://localhost/simple"));
        port.Open();
    }
}
```

```
// wait forever
    Console.ReadLine();
}
}
```

This service creates a new named port whose relative URI is `"/simple"` and is accessible over all installed transports. Indigo treats the `"soap:"` URI scheme as the transport-neutral address of the service. The Indigo connector will project this address onto both TCP and HTTP transports automatically, enabling the service to be addressed via any of these URIs:

```
soap://localhost/simple
http://localhost/simple
soap.tcp://localhost/simple
```

Unless an application needs to explicitly control transport selection, `soap:` is the preferred URI scheme and therefore is used throughout the remainder of this article.

The service just shown is extremely flexible—you can send it any sort of message over any transport and it will silently drop the message onto the floor. Writing code that touches the message requires a little more work. [Figure 3](#) shows a service that when sent a message whose content is a simple string, prints that string to the console. This service assumes that the body of the message is a simple string and will consistently and reliably fail if the body of the message is anything else.

To allow services to cope with the body in a more data-oriented form, Indigo also gives you access to message content as raw XML-structured data using `System.Xml`. The service in [Figure 4](#) dumps the content of every message into an XML text file.

This program takes advantage of the fact that the body of the message is always accessible via a streaming `XmlReader` interface. This capability is important when dealing with large messages, as the forward-only traversal model of `XmlReader` allows incremental processing to take place along the way as message data is streamed into memory.

The Indigo connector defines a small kernel of managed types for processing messages. The most central of these types is the concrete `Message` class, which is shown in [Figure 5](#). The `Message` class maintains an ordered list of headers and a single body which represents the content of the message. For convenience, the `Action` header (as defined by WS-Addressing as well as the HTTP bindings for SOAP) is called out as a distinguished property. Each message header implements the abstract `MessageHeader` interface (shown in [Figure 6](#)) that exposes core SOAP-isms such as `mustUnderstand` and `actor/role`. The body of the message must implement the abstract `MessageContent` interface which supports both object-based and XML-based access.

Next to the `Message` class, the most important message processing type is easily `IMessageHandler`. The `IMessageHandler` interface represents a piece of message processing code and has a single method that that is used to invoke the message processor: `ProcessMessage`.

```
public interface IMessageHandler {
    bool ProcessMessage(Message msg);
    // async support elided for clarity
}
```

In addition to the primary `ProcessMessage` method, the `IMessageHandler` interface also provides several methods for interacting with the `IAsyncResult` idiom used by asynchronous method execution. Implementations that want the default asynchronous behavior (such as the examples shown at the beginning of this section) often derive from the `SyncMessageHandler` abstract base class and provide only a `ProcessMessage` method body.

Recall that messages support streaming I/O over the body of the message. This means that message handlers that traverse the body effectively render the message useless for all subsequent use. For this reason, the `ProcessMessage` method returns a boolean indicating whether or not the message has been "consumed." If the body has not been consumed, handlers return `true` from `ProcessMessage` indicating that the message is still viable for further processing. If a handler consumes the message body, it is required to return `false` from `ProcessMessage` and is required to call `Close` on the message object as well, freeing up any of the underlying resources held by the message.

The following example demonstrates how to consume a message using a series of message handlers:

```
static void
```

```

ConsumeMessage(IList<IMessageHandler> code, Message data) {
    bool live = true;
    for (int i=0; i < code.Count && live; i++)
        live = code[i].ProcessMessage(data);

    if (live)
        data.Close();
}

```

The third and final core message processing type is `IMessageProducer`. `IMessageProducer` is a simple interface that models message sources and, as shown here, has only one public member, the `Handler` property:

```

public interface IMessageProducer {
    IMessageHandler Handler { get; set; }
}

```

You've already seen an implementation of this interface. The `Port.ReceiveChannel` property used in the previous service examples implements the `IMessageProducer` interface. Code that wants to process messages that arrive on a port simply attach themselves through the `Handler` property of the receive channel.

Up to this point, you have only seen how messages are received. Indigo obviously allows messages to be sent as well—this capability is exposed via a port's send channels. Every port has a default send channel (exposed via the `Port.SendChannel` property) that can be used to send a message that is explicitly addressed using a `To` header (per the WS-Addressing specification). This channel is typically only used to send reply messages created by the `Message.CreateReply` method:

```

static bool Process(Port port, Message request)
{
    Message data = request.CreateReply(null,
                                       "Blue");
    IMessageHandler channel = port.SendChannel;
    if (channel.ProcessMessage(data))
        data.Close();
}

```

The more common usage is to create a per-address send channel by calling the `Port.CreateSendChannel` method as follows:

```

static void SendAMessage(Port source,
                        Uri target) {
    Message data = new Message(null, "Cyan");
    IMessageHandler channel =
        source.CreateSendChannel(target);
    if (channel.ProcessMessage(data))
        data.Close();
}

```

In this case, the send channel will add the required addressing headers to the message prior to handing it off to the underlying transport manager.

Where Are We?

This article has provided just a glimpse into the capabilities of and the motivations behind Indigo. Given the broad scope of the Indigo project, many aspects of the Indigo programming experience have been glossed over by this introductory article. As the beta release of Indigo grows nearer, watch the [MSDN® Web Services Developer Center](#) for more complete coverage of Indigo.

Don Box is an architect on the Indigo project at Microsoft, working on next generation Web Service protocols and plumbing. His interests include type systems for XML and Web Services, metadata, discovery, and service-oriented software integration. Don's work with Web Services began in 1998 as one of the original authors of the SOAP specification. Don's latest book is *Essential .NET Volume 1: The Common Language Runtime* (Addison-Wesley, 2002).

From the [January 2004](#) issue of [MSDN Magazine](#).
Get it at your local newsstand, or better yet, [subscribe](#).

Figure 3 Print Single Message

```

using System;
using System.MessageBus;

class MsgPrint : SyncMessageHandler {
    public override bool ProcessMessage(Message msg)
    {
        Console.WriteLine(
            msg.Content.GetObject(typeof(string))
        );
        msg.Close();
        return false;
    }
}

class app {
    static void Main() {
        Port port = new Port(new
            Uri("soap://localhost/simple"));
        port.ReceiveChannel.Handler = new MsgPrint();
        port.Open();
        // wait forever
        Console.ReadLine();
    }
}

```

Figure 4 Dump Message to File

```

using System;
using System.Xml;
using System.MessageBus;

class MsgDump : SyncMessageHandler {
    public override bool ProcessMessage(Message msg)
    {
        string fn = String.Format("output{0}.xml",
            DateTime.Now.Ticks);
        XmlWriter xw = new XmlTextWriter(fn, null);
        xw.WriteNode(msg.Content.Reader, false);
        xw.Close();
        msg.Close();
        return false;
    }
}

class app {
    static void Main() {
        Port port = new Port(new
            Uri("soap://localhost/simple"));
        port.ReceiveChannel.Handler = new MsgDump();
        port.Open();
        // wait forever
        Console.ReadLine();
    }
}

```

Figure 5 The Message Class

```

public class Message {

    // public constructors
    public Message();
    public Message(Uri action);
    public Message(Uri action, object obj);
    public Message(Uri action, MessageContent content);
    public Message(MessageException exception);

    // create message targeted at sender
    public Message CreateReverse();
    public Message CreateReverse(Uri action);
    public Message CreateReverse(Uri action, object obj);
    public Message CreateReverse(Uri action, MessageContent content);
    public Message CreateReverse(MessageException exception);

    // create correlated reply targeted at sender
    public Message CreateReply();
    public Message CreateReply(Uri action);
    public Message CreateReply(Uri action, object obj);
    public Message CreateReply(Uri action, MessageContent content);
}

```

```

    public Message CreateReply(MessageException exception);

// public operations
    public Message Clone();
    public void Close();

// public properties
    public Uri Action { get; set; }
    public MessageHeaderCollection Headers { get; }
    public MessageContent Content { get; set; }
}

```

Figure 6 Message Headers and Body

```

public abstract class MessageHeader {
// protected constructors
    protected MessageHeader(bool mustUnderstand, Uri role);
    protected MessageHeader(MessageHeader source);

// SOAP processing/data model properties
    public abstract string Name { get; }
    public abstract string Namespace { get; }
    public bool MustUnderstand { get; }
    public Uri Role { get; }
    public bool DidUnderstand { get; set; }

// header serialization methods and properties
    public abstract MessageHeader Clone();
    public virtual XmlElement Element { get; }
    public abstract void WriteTo(XmlWriter writer);
    public abstract bool CanWrite { get; }
    public virtual bool CanTransmit { get; }
}

public abstract class MessageContent {
// object I/O
    public abstract object GetObject(Type type);

// XML I/O
    public virtual XmlReader Reader { get; }
    public abstract void WriteTo(XmlWriter writer);

// async XML I/O support
    public virtual IAsyncResult BeginWriteTo(XmlWriter writer,
                                             AsyncCallback callback,
                                             object state);
    public virtual void EndWriteTo(IAsyncResult result);

// public properties and methods
    public virtual bool IsEmpty { get; }
    public virtual bool IsException { get; }

    public abstract MessageContent Clone();
    public virtual void Close();
}

```

© 2003 Microsoft Corporation. All rights reserved.