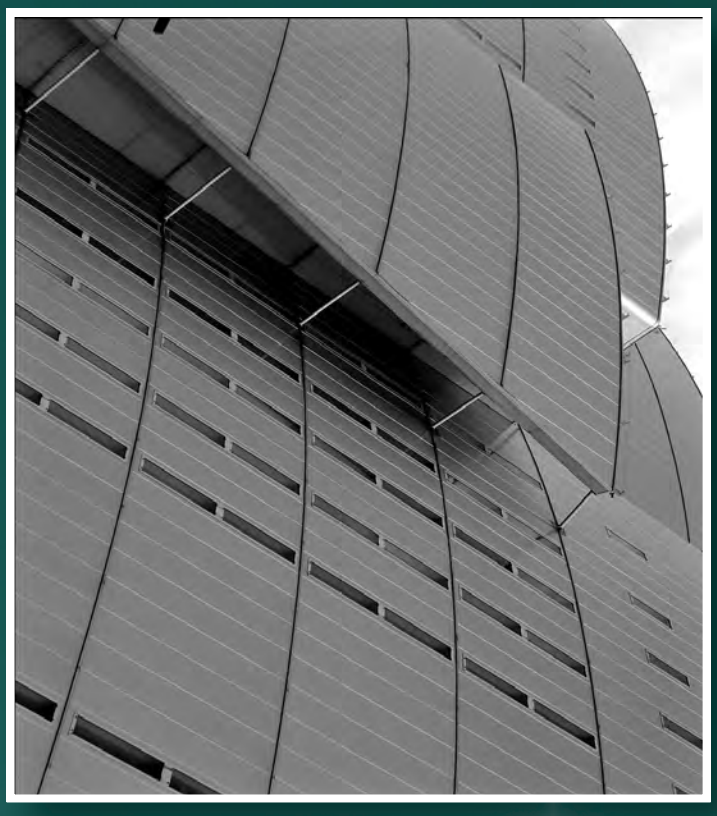


Chapter 9

Service-Orientation with .NET Part I: Service Contracts and Interoperability

9.1 Standardized Service Contract

9.2 Canonical Schema

9.3 Data Model Transformation

9.4 Canonical Protocol

9.5 Canonical Expression

SOA PRINCIPLES & PATTERNS REFERENCED IN THIS CHAPTER

- Canonical Expression [715]
- Canonical Protocol [716]
- Canonical Resources [717]
- Canonical Schema [718]
- Data Model Transformation [732]
- Decoupled Contract [735]
- Dual Protocols [739]
- Federated Endpoint Layer [745]
- Messaging Metadata [753]
- Protocol Bridging [764]
- Schema Centralization [769]
- Standardized Service Contract (693)

Each of the referenced service-orientation principles was briefly introduced in Chapter 3. The respective principle and pattern profile tables are available in Appendices C and D.

WCF (together with the .NET framework extensions and Windows Azure) establishes a broad, almost all-encompassing platform for service development, deployment, and hosting. When building entire service inventories upon such a platform, we need to consider that one of the primary goals of service-oriented computing in general is that of Increased Vendor Diversification Options.

This strategic goal aims to establish service-oriented technology architecture in such a manner that best-of-breed technologies can be leveraged on the back-end, while preserving a federated service endpoint layer on the consumer-side, which is related to another primary goal called Increased Federation.

An important aspect of the Increased Vendor Diversification Options goal is the "Options" part. This goal does not advocate that you diversify your IT enterprise. Doing so unnecessarily can lead to increased governance burden and increased cost of ownership. For example, if WCF, .NET, and Windows Azure continue to empower you to maximize business requirements fulfillment, the collective environment they establish can be highly effective for broad SOA adoption. By leveraging common platform class libraries, system services, repositories, and other mechanisms, we are, in effect, repeatedly applying the Canonical Resources [717] pattern and reaping its ownership benefits on a potentially grand scale.

The goal of Increased Vendor Diversification Options simply states that you should always retain the *option* of being able to bring in technologies and products from other vendors. That way, when justified, you can diversify in order to increase your business requirement fulfillment potential. In other words, avoid vendor lock-in so that, over time, you are not inhibited by a specific vendor's product roadmap (which may end up straying from the direction in which you need to evolve your business automation solutions).

To realize this goal we need to go back to the aforementioned goal of Increased Federation, a goal that is directly tied to the application of the Federated Endpoint Layer [745] pattern. By establishing a layer of standardized service contracts (endpoints) federated within a given service inventory boundary, we create an inter-service communications framework that is completely abstracted from the technologies, platforms and products that comprise individual, back-end, service architecture implementations.

It is this form of clean, far-reaching abstraction that gives us the freedom to diversify without having to rip-and-replace an entire ecosystem. Instead, we can continue to leverage existing vendor platforms as they remain useful and beneficial, and then refactor individual service architectures independently to augment and grow our inventory of services in tandem with on-going business change.

The success factors behind both the Increased Vendor Diversification Options and Increased Federation strategic goals can be mapped to the appropriate design, development, and architectural positioning of service contracts. These factors are critical to the realization of the Increased Intrinsic Interoperability goal that is core to the overall objective and long-term target state advocated by service-orientation.

The attainment of an intrinsic level of interoperability within each service is a broad topic with numerous sub-topics, many of which are addressed by various chapters in this book. This chapter kicks things off by exploring the application of .NET technologies with key principles and patterns that pertain to service contract development and the creation of federated service endpoints.

9.1 Standardized Service Contract

This principle advocates the standardization of service contracts that exist within a given service inventory boundary (Figure 9.1). Within this context, standardization can refer to the usage of industry standards (such as WSDL and XML Schema), but primarily the focus is on custom design standards that are pre-defined and regulated (such as canonical data models).

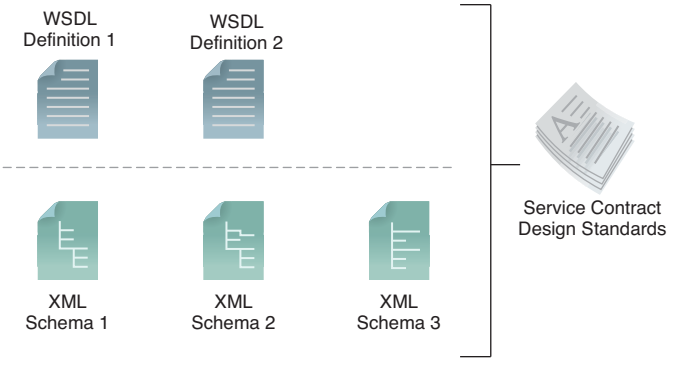


Figure 9.1
The application of the Standardized Service Contract (693) principle relies heavily on the use of design standards that regulate contract design across services within a given service inventory.

Contract-First

To ensure that contracts are consistently standardized requires that we seize programmatic control over how contracts are created, which is the basis of the *contract-first* approach that is predominantly used when applying this principle.

There are various ways of going about contract-first development with .NET. The primary consideration is in determining in what sequence to define the various parts of a service contract.

For example, here is a common three-step process:

1. Create or Reuse Data Contract

A WCF data contract most commonly exists as an XML schema that defines the data structure and data types (as part of the overall data model) for a given set of data that is exchanged by a service capability.

Following patterns, such as Canonical Schema [718] and Schema Centralization [769], this step may involve creating a new data contract to be used by one or more services or it may involve reusing an already existing (and standardized) data contract. The latter circumstance may be due to a data contract that was previously customized or there may be a requirement to use a data contract that exists as an industry standard XML Schema (such as HR-XML or LegalML).

NOTE

Schema Centralization [769] and reusable schemas are discussed shortly in the *Canonical Schema* section.

2. Create Message Contract

The body content of a given message transmitted or received by a service capability is primarily predefined by the data contract. The message contract encompasses the data contract and further defines metadata in the message header, as per Messaging Metadata [753]. Message contracts within WCF are primarily built using the SOAP `Body` and `Header` constructs. Part of this step may also involve predefining fault contracts for exception conditions.

SERVICE MODELING & SERVICE CANDIDATES

Prior to applying any contract-first development approach, it is generally assumed that some extent of service modeling has already been completed. The service modeling process is part of the service-oriented analysis stage within a service's overall delivery cycle. Service modeling produces conceptual services called *service candidates* that form the basis of service contracts. Often significant up-front analysis is carried out in order to define several service candidates for a specific service inventory before physically building any one service contract. This effectively creates a service inventory blueprint that allows the basic parts of service contracts to be well-defined and further refined through iteration, prior to entering the design and development phases.

3. Create Interface Contract

The interface contract is commonly equated to the abstract description of a WSDL document wherein operation contracts are defined. When working with REST services, the interface contract can be considered the subset of HTTP methods that are supported as part of the overall uniform contract for a given service.

The interface contract and its operations or methods express the externally invokable functionality offered by a service. An interface contract pulls together the data and message contracts and associates them with appropriate operations or methods.

NOTE

For detailed coverage of carrying out contract-first processes with .NET technologies, see the *Decoupled Contract* section in Chapter 10.

Standardized Service Contract (693) and Patterns

Beyond carrying out a contract-first approach to service design, there are many more facets to applying the Standardized Service Contract (693) principle. Several of these additional aspects will come up in the subsequent sections exploring SOA design patterns related to service interoperability.

SUMMARY OF KEY POINTS

- The application of the Standardized Service Contract (693) principle commonly involves following a contract-first approach to service-oriented design.
- The Standardized Service Contract (693) principle is closely associated with several patterns, including Canonical Schema [718], Schema Centralization [769], Canonical Protocol [716], and Canonical Expression [715], all of which support its application in different ways.

9.2 Canonical Schema

The XML Schema Definition Language is a highly successful industry standard that has received broad cross-platform support within, and well beyond, the SOA industry. With this language you can use industry standard markup syntax to not only express the structure and validation rules of business documents, but you can also use a series of built-in data types to represent the actual data. This allows you to define complete data models in a manner that is independent of any proprietary database or data representation technology.

Canonical Schema [718] establishes standardized XML Schema definitions (Figure 9.2), which makes this a pattern that can be applied in direct support of Standardized Service Contract (693).

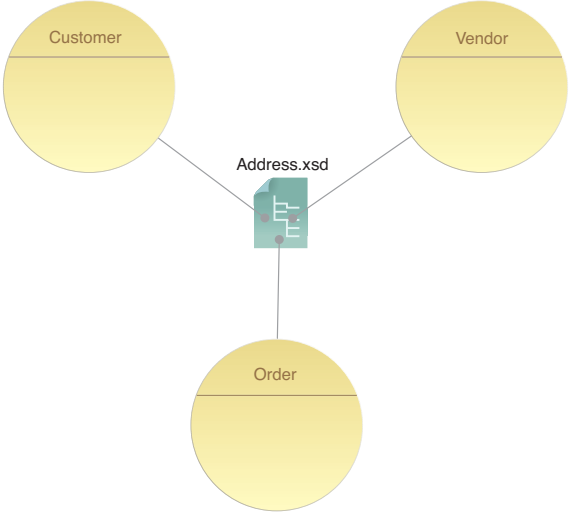


Figure 9.2
By sharing a canonical schema definition, different services increase their respective compatibility and interoperability.

Schema Centralization [769], a related pattern that mandates that schema definitions be shared among service contracts, can be applied together with Canonical Schema [718] to help create a flexible and streamlined data architecture that acts as a foundation layer for a set of federated service contracts (Figure 9.3). The abstraction achieved by such a data architecture allows you to build underlying services using .NET, Java, or any other back-end implementation platform that supports XML Schema processing.

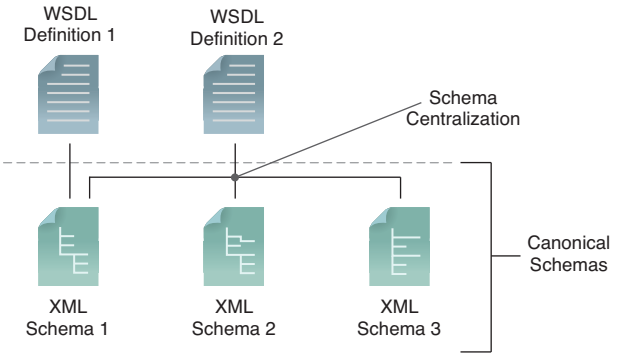


Figure 9.3

When canonical schemas are centralized, they are essentially reused by different service contracts, resulting in increased normalization across the service data and endpoint architecture.

Creating Schemas with Visual Studio

To build canonical XML schemas that we will eventually want to centralize, we need to master the XML Schema Definition Language and customize these schemas using an editor, such as the one provided by Visual Studio.

> **NOTE**
>
> This book does not provide tutorial coverage of the XML Schema Definition Language. If you are new to XML Schema, refer to Chapters 6, 12, and 13 of the book *Web Service Contract Design & Versioning for SOA*.

Let's now put together some simple schemas. Figure 9.4 shows a preview of the person schema displayed in the Visual Studio 2010 Schema View. We can also refer to the person schema as the person *type*, because it essentially establishes a complex type for the person entity.

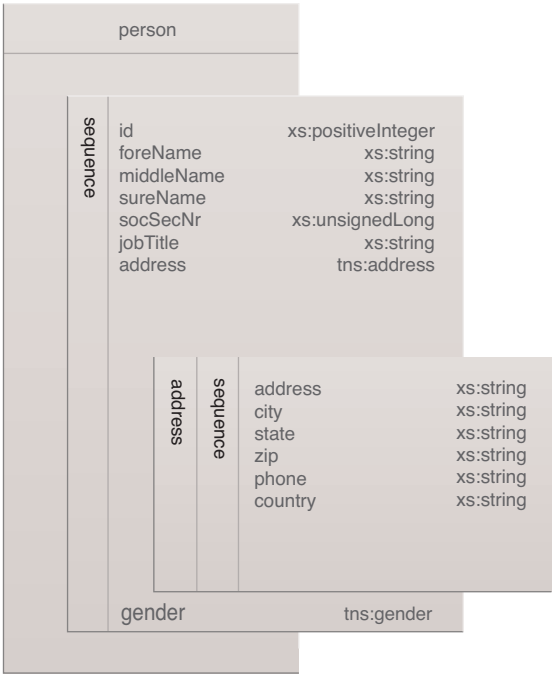


Figure 9.4
XML Schema types defined.

We'll begin with putting together the gender type first:

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
  targetNamespace="http://schemas.example.org/
    enterprise/models/v1"
  xmlns="http://schemas.example.org/
    enterprise/models/v1"
  xmlns:mstns="http://schemas.example.org/
    enterprise/models/v1"
  version="1.0.1"
  elementFormDefault="qualified">
  <xs:simpleType name="gender">
    <xs:restriction base="xs:string">
      <xs:enumeration value="male" />
      </xs:enumeration value="female" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Example 9.1

The validation logic in this schema only accepts string values that are "male" and "female." The gender type is saved as a separate XML schema file called gender.xsd.

Next, let's create the address type:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://schemas.example.org/
    enterprise/models/v1"
  xmlns="http://schemas.example.org/
    enterprise/models/v1"
  xmlns:mstns="http://schemas.example.org/
    enterprise/models/v1"
  version="1.0.1"
  elementFormDefault="qualified">
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="address" type="xs:string" />
      <xs:element name="city" type="xs:string" />
      <xs:element name="state" type="xs:string" />
      <xs:element name="zip" type="xs:string" />
      <xs:element name="phone" type="xs:string" />
      <xs:element name="country" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Example 9.2

The address type accepts strings for all of its child elements. This type is also saved in a separate schema file (called address.xsd).

Finally, here's the content for the person type, which is stored in the person.xsd file:

```
<xs:schema xmlns:xs=
  "http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.example.org/
    enterprise/models/v1"
  targetNamespace="http://schemas.example.org/
    enterprise/models/v1"
  xmlns:mstns="http://schemas.example.org/
  enterprise/models/v1"
  version="1.1.20"
```

```
  elementFormDefault="qualified">
  <xs:include schemaLocation="Gender.xsd"/>
  <xs:include schemaLocation="Address.xsd"/>
  <xs:complexType name="person">
    <xs:sequence>
      <xs:element name="ID" type="xs:positiveInteger" />
      <xs:element name="foreName"
        type="xs:string" minOccurs="1"/>
      <xs:element name="middleName" type="xs:string" />
      <xs:element name="surName"
        type="xs:string" minOccurs="1"/>
      <xs:element name="socSecNr" type="xs:unsignedLong"
        minOccurs="1" maxOccurs="1"/>
      <xs:element name="jobTitle"
        type="xs:string" minOccurs="0" maxOccurs="1" />
      <xs:element name="address" type="tns:address" />
      <xs:element name="gender" type="tns:gender" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="person" type="tns:person" />
</xs:schema>
```

Example 9.3

Note how the types of elements `address` and `gender` refer to the types created in the address.xsd and gender.xsd respectively. Two statements are necessary in order to make the `address` element refer to the type defined in the address.xsd file. The first refers to the previously created schema:

```
<xs:include schemaLocation="address.xsd"/>
```

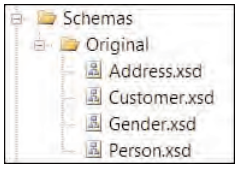
The second specifies that the element `address` should be of this type:

```
<xs:element name="address" type="tns:address"/>
```

Also note that there is an element in the person XML schema that has the type of person. This makes it possible to send the person type as a message. The gender and address types do not have such an element because we are not planning to use them on their own to define individual messages.

At this point our catalog of XML schemas looks like this:

Figure 9.5
The catalog of XML schemas.



Generating .NET Types

The types we have created can be used "as is" for message definitions in BizTalk, but to use them for WCF (or ASMX) services, we need to generate .NET types using a utility program called svcutil that is shipped with Visual Studio. To generate these types, use the following statement at the Visual Studio command prompt:

```
svcutil  /dcOnly  /l:cs  person.xsd  address.xsd  gender.xsd
```

Apart from referring to the previously created XML schema files, we are also using these two switches:

- /dcOnly – instructs svcutil to create data contracts for us
- /l:cs – instructs svcutil that the language (`l`) of the generated code should be *C#* (`cs`)

The code that scvutil generates based upon our types looks like this:

```
namespace schemas.example.org.enterprise.models.v1
{
  using System.Runtime.Serialization;
  [System.Diagnostics.DebuggerStepThroughAttribute()]
  [System.CodeDom.Compiler.GeneratedCodeAttribute
    ("System.Runtime.Serialization", "4.0.0.0")]
  [System.Runtime.Serialization.DataContractAttribute
    (Name = "person", Namespace =
    "http://schemas.example.org/enterprise/models/v1")]
  public partial class person : object,
    System.Runtime.Serialization.IExtensibleDataObject
  {
    private System.Runtime.Serialization.
      ExtensionDataObject extensionDataField;
    private long idField;
    private string foreNameField;
```

```
private string middleNameField;
private string surNameField;
private ulong socSecNrField;
private string jobTitleField;
private schemas.example.org.enterprise.models.v1.
  address addressField;
private schemas.example.org.enterprise.models.v1.
  gender genderField;
public System.Runtime.Serialization.
  ExtensionDataObject ExtensionData
{
  get
  {
    return this.extensionDataField;
   }
   set
   {
     this.extensionDataField = value;
    }
  }
  [System.Runtime.Serialization.
    DataMemberAttribute(IsRequired = true)]
  public long id
  {
    get
  {
    return this.idField;
  }
    set
  {
    this.idField = value;
  }
}
[System.Runtime.Serialization.
  DataMemberAttribute(IsRequired = true,
  EmitDefaultValue = false, Order = 1)]
public string foreName
{
  get
{
  return this.foreNameField;
}
  set
{
  this.foreNameField = value;
```

```
    }
  }
  [System.Runtime.Serialization.
    DataMemberAttribute(IsRequired = true,
    EmitDefaultValue = false, Order = 2)]
  public string middleName
  {
    get
  {
    return this.middleNameField;
  }
    set
  {
    this.middleNameField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false, Order = 3)]
  public string surName
  {
    get
  {
    return this.surNameField;
  }
    set
  {
    this.surNameField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, Order = 4)]
  public ulong socSecNr
  {
    get
  {
    return this.socSecNrField;
  }
  set
  {
    this.socSecNrField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (EmitDefaultValue = false, Order = 5)]
  public string jobTitle
```

```
  {
    get
  {
    return this.jobTitleField;
  }
    set
  {
    this.jobTitleField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false, Order = 6)]
  public schemas.example.org.enterprise.models.v1.address address
  {
    get
  {
    return this.addressField;
  }
    set
  {
    this.addressField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, Order = 7)]
  public schemas.example.org.enterprise.models.v1.gender gender
  {
    get
  {
    return this.genderField;
  }
    set
  {
    this.genderField = value;
  }
 }
}
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute
  ("System.Runtime.Serialization", "4.0.0.0")]
    [System.Runtime.Serialization.DataContractAttribute
  (Name = "address", Namespace = "http://schemas.example.org/
    enterprise/models/v1")]
    public partial class address : object,
      System.Runtime.Serialization.IExtensibleDataObject
```

```
    {
    private System.Runtime.Serialization.
      ExtensionDataObject extensionDataField;
    private string addressMemberField;
    private string cityField;
    private string stateField;
    private string zipField;
    private string phoneField;
    private string countryField;
    public System.Runtime.Serialization.
      ExtensionDataObject ExtensionData
    {
      get
    {
      return this.extensionDataField;
    }
      set
    {
      this.extensionDataField = value;
    }
  }
  [System.Runtime.Serialization.DataMemberAttribute
    (Name = "address", IsRequired = true,
     EmitDefaultValue = false)]
  public string addressMember
  {
    get
  {
    return this.addressMemberField;
  }
    set
  {
    this.addressMemberField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false)]
  public string city
  {
    get
  {
    return this.cityField;
  }
    set
  {
```

```
    this.cityField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false)]
  public string state
  {
    get
  {
    return this.stateField;
  }
    set
  {
    this.stateField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false)]
  public string zip
  {
    get
  {
    return this.zipField;
  }
    set
  {
    this.zipField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false, Order = 4)]
  public string phone
  {
    get
  {
    return this.phoneField;
  }
    set
  {
    this.phoneField = value;
  }
}
[System.Runtime.Serialization.DataMemberAttribute
  (IsRequired = true, EmitDefaultValue = false, Order = 5)]
  public string country
```

```
  {
    get
  {
    return this.countryField;
  }
    set
  {
    this.countryField = value;
  }
 }
}
[System.CodeDom.Compiler.GeneratedCodeAttribute
  ("System.Runtime.Serialization", "4.0.0.0")]
[System.Runtime.Serialization.DataContractAttribute
  (Name = "gender", Namespace = "http://schemas.
  example.org/enterprise/models/v1")]
  public enum gender : int
  {
    [System.Runtime.Serialization.
      EnumMemberAttribute()]
    male = 0,
    [System.Runtime.Serialization.
      EnumMemberAttribute()]
    female = 1,
  }
}
```

Example 9.4

NOTE

As an alternative to the svcutil utility, you can also use the xsd.exe utility, which will create .NET types that are serialized using the `XMLSerializer` rather than the `DataContractSerializer`.

Using the `DataContract` Library

You can also create these types by using code directly with .NET, instead of working with XML schema markup code. This next example shows how to create the three types in .NET using the `DataContract` and `DataMember` attributes.

```
 [DataContract(Name = "person", Namespace =
  "http://schemas.example.org/enterprise/models/v1")]
  public class Person : object,
  System.Runtime.Serialization.IExtensibleDataObject
{
  [DataMember(IsRequired = true)]
  public int Id
  {
    get;
    set;
  }
  [DataMember(IsRequired=true,
    EmitDefaultValue=false, Order = 1)]
  public string ForeName
  {
    get;
    set;
  }
  [DataMember(IsRequired = true,
    EmitDefaultValue=false, Order = 2)]
  public string MiddleName
  {
    get;
    set;
  }
  [DataMember(IsRequired = true,
    EmitDefaultValue=false, Order = 3)]
  public string SurName
  {
    get;
    set;
  }
  [DataMember(IsRequired = true,
    EmitDefaultValue=false, Order = 4)]
  public ulong SocSecNr
  {
    get;
    set;
  }
  [DataMember(Order = 5)]
  public string JobTitle
  {
    get;
    set;
  }
  [DataMember(Order = 6)]
```

```
  public Address Address
  {
    get;
    set;
  }
  [DataMember(Order = 7)]
  public Gender Gender
  {
    get;
    set;
  }
  private System.Runtime.Serialization.
    ExtensionDataObject extensionDataField;
  public System.Runtime.Serialization.
    ExtensionDataObject ExtensionData
  {
    get
  {
    return this.extensionDataField;
  }
    set
  {
    this.extensionDataField = value;
  }
}
```

Example 9.5

The address and gender types are created as separate classes that use `DataContract` and `DataMember` attributes. The `DataContract` attribute is used to let .NET know that this is a data contract and that it should be serialized using `DataContractSerializer`. This serializer will only serialize fields that are annotated with the `DataMember` attribute.

Apart from signaling to the `DataContractSerializer` that a particular element should be serialized, the `DataMember` attribute accepts named arguments.

Specifically, the named arguments used in Example 9.5 are:

- `Order` – affects the order in which the fields of the type are serialized
- `IsRequired` – instructs consumers and services whether or not a field is required
- `EmitDefailtValue` – when set to false, the default value (for example, 0 for an int) will not be serialized

The `DataMember` attribute has several other arguments that can come in handy. For example, should you require a different name in the .NET code and the serialized message, the `Name` attribute lets you change the name of the serialized value.

Figure 9.6 shows the resulting library created so far.

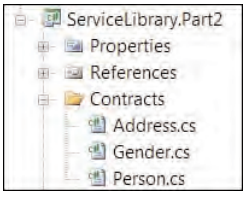


Figure 9.6
The Service Library defined so far, including Address, Gender, and Person class definitions.

SUMMARY OF KEY POINTS

- Canonical Schema [718] establishes the requirement for schemas used by services within a service inventory to be standardized.
- Visual Studio provides an editor that allows for the definition of XML schemas.
- With .NET, XML Schema types can be created with the XML Schema markup language or through the use of the DataContract library.

9.3 Data Model Transformation

One goal of the Standardized Service Contract (693) principle is to avoid having to transform data at runtime. This means that the more successfully and broadly we are able to apply this principle, the less need we will have for patterns like Data Model Transformation [732]. However, even when seeking service contract standardization, there are situations where this pattern is necessary.

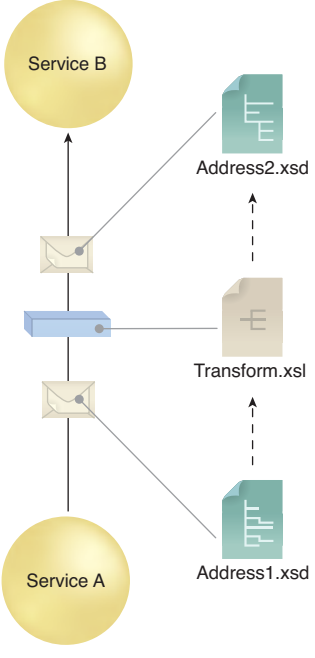
For example:

- When an IT enterprise has multiple domain service inventories, each collection of services can be subject to different design standards. In this case, when required to enable cross-inventory communication (or when creating a service composition comprised of services from multiple service inventories), any disparity in message data models will need to be overcome by applying Data Model Transformation [732].

- When data needs to be shared between different organizations (or organizational entities), Data Model Transformation [732] will generally be required unless Canonical Schema [718] has been successfully applied via the use of custom or industry-standard message schemas.
- When services encapsulate legacy systems and resources, they will inevitably need to transform data between legacy data models and the standardized data model defined in the service contracts. In this case, Data Model Transformation [732] is carried out within the service architecture.
- When services within a service inventory are not all successfully standardized (meaning the Standardized Service Contract (693) principle was not applied to its full extent), Data Model Transformation [732] will be required to enable the necessary interoperability.

Data Model Transformation [732] is generally carried out by creating mapping logic between disparate schemas or data types (Figure 9.7). This type of logic can often become complex and is sometimes even impossible to develop when the disparity between data models is too large.

Figure 9.7
Service A sends a message to Service B. The message sent by Service A contains address data that was defined by a schema that is different than the schema Service B uses in its service contract for this same information. Therefore, transformation logic is processed by a service agent in order to transform the message at runtime into data that complies with Service B's schema.



For example, you may encounter mandatory fields in one model that don't exist in the other. In such a case, transforming in one direction may work, but transforming in the opposite direction may not. The following example demonstrates by showing how we may not be able to determine which value(s) belong in the `middleNames` element:

Data Model #1

```
<person1>
  <foreNames>Max</foreNames>
  <middleNames>Carl</middleNames>
  <surNames>von Sydow</surNames>
</person1>
```

Data Model #2

```
<person2>
  <name>Max Carl von Sydow</name>
</person2>
```

Example 9.6
Transforming from Data Model #1 to Data Model #2 works, but the opposite transformation is more difficult.

Besides the potential complexity of mapping logic, there are other well-known impacts of applying this pattern. The additional logic will introduce development and governance effort, and can further affect the performance of services and service compositions (sometimes significantly so, especially with more complex mapping logic).

The following sections briefly show three ways to apply Data Model Transformation [732] using .NET technologies.

Object-to-Object

A message sent by a service consumer to a service can be serialized from XML into an object, translated into another object, and then serialized into XML again. This may be a suitable approach when you must use all or most of the data in the message, either for passing the information onto another service or for some other purpose.

The first step in this process is to understand the mapping requirements. Let's take, for example, a scenario where we need to transform data defined by a person type into a customer type (Figure 9.8). The logic behind this transformation could be as follows:

- map the `person.id` field to the `customer.id` field
- map the `person.foreName` field to the `customer.firstName` field

- map the `person.surName` field to the `customer.lastName` field
- map the `person.address.phone` field to the `customer.phone` field
- map the `person.gender` field to the `customer.gender` field

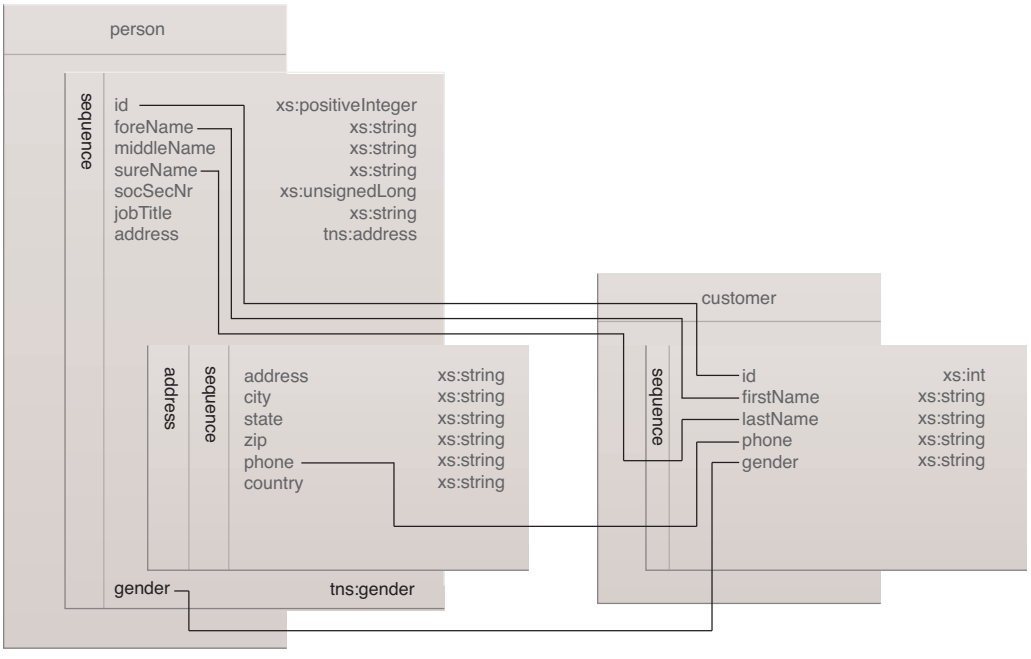


Figure 9.8
Mapping fields from the person type to the customer type.

After generating proxy clients for the different services with the "Add service reference" feature of Visual Studio, we also have .NET code that represents a person and a customer. The translation could then be programmed using a static extension method as shown here:

```
public static Customer TransformPersonToCustomer
  (this Person person)
{
  Customer customer = new Customer();
  customer.id = person.Id;
  customer.firstName = person.ForeName;
  customer.lastName = person.SurName;
  customer.phone = person.Address.PhoneNr;
```

```
  customer.gender = IntToEnum<customergenderType>((int)
    person.gender);
  return customer;
}
public static T IntToEnum<T>(int value)
{
  return (T)Enum.ToObject(typeof(T), value);
}
```

Example 9.7

This code was written knowing that person and customer use the same integer representations of male and female. The next example shows how this translation logic is applied on the message retrieved from the first service before sending it to the second. (Note that since this logic was written as an extension method it can be called as if it was a method of the person class.)

```
PersonServiceClient personService = new PersonServiceClient();
var aPerson = personService.GetPerson(2);
var customerFromPerson =
  getPersonData.TransformPersonToCustomer();
CustomerServiceClient customerService = new
  CustomerServiceClient();
customerService.UpdateCustomer(customerFromPerson);
```

Example 9.8

LINQ-to-XML

Sometimes you may want to transform a type but you only need to transform a subset of the overall type. In those cases, deserializing a large document and building a large object graph can be wasteful and can make code more sensitive to future changes in data structures.

To handle this situation you can use LINQ-to-XML on the raw message that is returned from a service, as follows:

```
Message messageOut = channel.Request(messageIn);
XmlReader readResponse = messageOut.GetReaderAtBodyContents();
XmlDocument doc = new XmlDocument();
doc.Load(readResponse);
var xDoc = XDocument.Parse(doc.OuterXml);
XNamespace xmlns2 = xDoc.Root.Attribute("xmlns").Value;
```

```
var transformed = from d in xDoc.Descendants
  (xmlns2 + "personElement")

select new Customer
{
  firstName = d.Element(xmlns2 + "foreName").Value,
  lastName = d.Element(xmlns2 + "surName").Value,
  id = Convert.ToInt32(d.Element(xmlns2 + "id").Value),
  gender = (Customer.genderType)
    Enum.Parse(typeof(Customer.genderType),
    d.Element(xmlns2 + "gender").Value),
    phone = d.Element(xmlns2 + "address").
    Element(xmlns2 + "phonenr").Value,
};
```

Example 9.9

In this example, the response content from the message issued by the service is received; along with the namespace, the incoming XML structure is transformed into an object. Note that this code would continue to work even if the namespace of the response changes. The only change that can break this code is if one of the elements that are explicitly asked for is removed or has its name altered. An additional benefit of this approach is that we can process data types that are not otherwise easily handled in WCF, such as the `xsd:choice` construct.

XSLT Transformation

An option that is useful for avoiding deserialization is XSLT. By defining mapping logic with the XSLT language, we only take the data that we are interested in into our objects and leave the rest. XSLT can be used by different parts of the .NET platform and is a commonly supported industry standard, meaning that it may also be used with some legacy systems.

Here's a sample XSLT transformation:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:b="http://schemas.example.org/enterprise/models/v1"
  xmlns:a="http://schemas.example.org/enterprise/models/v2">
  <xsl:template match="/">
    <a:customerElement>
      <a:id>
```

```
        <xsl:value-of select="b:personElement/b:id"/>
      </a:id>
      <a:firstName>
        <xsl:value-of select="b:personElement/b:foreName"/>
      </a:firstName>
      <a:lastName>
        <xsl:value-of select="b:personElement/b:surName"/>
      </a:lastName>
      <a:phone>
        <xsl:value-ofselect=
        "b:personElement/b:address/b:phonenr"/>
      </a:phone>
      <a:gender>
        <xsl:value-of select="b:personElement/b:gender"/>
      </a:gender>
    </a:customerElement>
  </xsl:template>
</xsl:stylesheet>
```

Example 9.10

The semantics of this transformation logic are quite straight forward in that the markup shows how to find the values for an element.

Using WCF, you could apply this XSLT transformation as follows:

```
Message messageOut = channel.Request(messageIn);
XmlReader readResponse = messageOut.GetReaderAtBodyContents();
XslCompiledTransform xslt = new XslCompiledTransform();
xslt.Load("XMLMessages/TransformationPersonToCustomer.xslt");
using (MemoryStream ms = new MemoryStream())
{
  XmlWriterSettings ws = new XmlWriterSettings();
  ws.Encoding = Encoding.UTF8;
  using (XmlWriter xmlWriter = XmlWriter.Create(ms, ws))
  {
    xslt.Transform(readResponse, xmlWriter);
  }
  xmlWriter
}
```

Example 9.11

SUMMARY OF KEY POINTS

- Although Data Model Transformation [732] is a pattern we try to avoid when applying Standardized Service Contract (693) and Canonical Schema [718], it is still commonly applied within service-oriented architectures.
- With .NET, three common ways of applying Data Model Transformation [732] are object-to-object, LINQ-to-XML, and XSLT.

9.4 Canonical Protocol

In heterogeneous environments it is common for systems to have difficulties communicating directly. Some may be using the TCP as a transport protocol, while others may only be capable of using HTTP over TCP, or even SOAP (over HTTP and TCP). Even when two legacy systems use the same protocol, they might be using different versions, which can result in the same communication-level incompatibility.

A technique for overcoming these problems is via Protocol Bridging [764]. In essence, this pattern involves placing an intermediary in between two pieces of software that converts between the incompatible protocols, thereby enabling them to exchange data. As with Data Model Transformation [732], applying this pattern will lead to increased development effort and increased performance overhead.

The Standardized Service Contract (693) principle further helps establish the standardized interoperability on the protocol level with the help of the Canonical Protocol [716] pattern, which requires that communication protocol (including protocol versions) be regulated among services within the same service inventory boundary.

Of course, this leads to the question of which protocol to choose. The choice of protocol will be dependent on the choice of service implementation medium. Currently there are three common service implementation options:

- components
- Web services
- REST services

The following sections explore the differences of each in relation to building services with WCF.

Web Service

A Web service uses a WSDL definition and one or more XML schemas to specify its interfaces. Its protocol is usually based on the use of SOAP over HTTP. Figure 9.9 shows a Web service implemented in WCF with the `IUserBankService` interface and Figure 9.10 illustrates the `DataContract` for representing the `User` class.

Figure 9.9
This Web service has methods (operations) for creating, getting, updating and modifying a user.

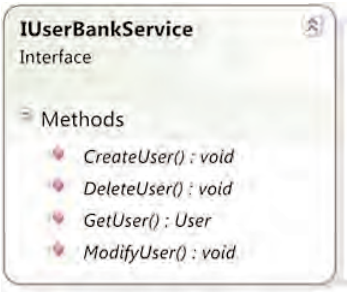
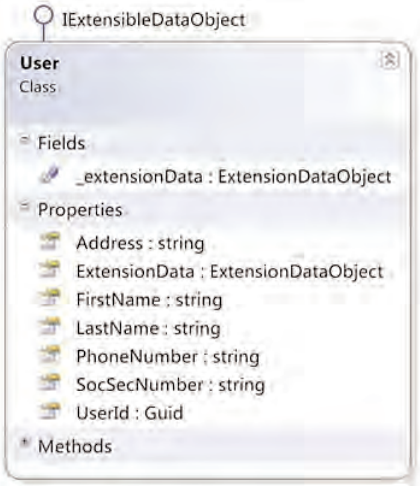


Figure 9.10
The `User` class and associated properties.



As shown in the following example, the interface is created and annotated with the `ServiceContract` and `OperationContract` attributes.

```
[ServiceContract]
  public interface IUserBankService
  {
    [OperationContract]
    void CreateUser(Core.Models.User user);
```

```
    [OperationContract]
    Core.Models.User GetUser(Guid userId);
    [OperationContract]
    void ModifyUser(Core.Models.User modifiedUser);
    [OperationContract]
    void DeleteUser(Guid userId);
  }
```

Example 9.12

The `ServiceContract` attribute indicates that it's a WCF service and the `OperationContract` attribute indicates that the method that is annotated with this attribute needs to be exposed by the service. Note that these attributes are used irrespective of the kind of service we're creating with WCF.

The interface is then implemented in a class, as shown here:

```
public class UserBankService:IUserBankService
{
  public void CreateUser(Core.Models.User user)
  {
    throw new NotImplementedException();
  }
  public Core.Models.User GetUser(Guid userId)
  {
    throw new NotImplementedException();
  }
  public void ModifyUser(Core.Models.User modifiedUser)
  {
    throw new NotImplementedException();
  }
  public void DeleteUser(Guid userId)
  {
    throw new NotImplementedException();
  }
}
```

Example 9.13

As you can see, no attributes are used on the class as they were already used on the interface. To make the service actually do something, we would need to populate the method definitions with code.

Note that we could have decorated the class and the methods in the class with the WCF attributes. By instead decorating the interface, we have applied the Decoupled Contract [735] pattern by separating the service definition from its implementation.

Second, there is nothing in our code so far that specifies that this should be a Web service. To make it into a Web service, we can add a configuration. This next example shows the relevant part of the configuration that implements this service as an actual Web service:

```
...
<system.serviceModel>
  <services>
    <service name="Core.Services.UserBankService">
      <endpoint address="..." binding="basicHttpBinding"
        contract="Core.Services.IUserBankService">
          ...
      </endpoint>
    </service>
  </services>
  ...
</system.serviceModel>
...
```

Example 9.14

`basicHttpBinding` is what makes this service into a Web service, as it instructs WCF to use a WS-BasicProfile Web service communication mechanism with HTTP as transport and messages encoded as text/XML.

REST Service

When designing a service as a REST service we can still use WCF and the resulting code is actually quite similar to that of a Web service implementation. Figure 9.11 shows an interface that corresponds to the previous Web service example.

As with the Web service interface definition, the REST service interface is annotated with WCF attributes:

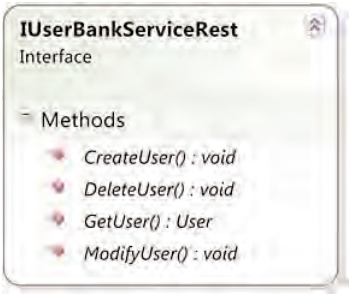


Figure 9.11
The interface definition for the REST service is identical to the previous definition for the Web service, except for the name.

```
[ServiceContract]
public interface IUserBankServiceRest
{
  [OperationContract]
  [WebInvoke(Method = "POST", BodyStyle =
    WebMessageBodyStyle.Bare,
    ResponseFormat = WebMessageFormat.Xml)]
  void CreateUser(Core.Models.User user);
  [OperationContract]
  [WebGet(UriTemplate="users/{userId}")]
  Core.Models.User GetUser(string userId);
  [OperationContract]
  [WebInvoke(Method = "PUT")]
  void ModifyUser(Core.Models.User modifiedUser);
  [OperationContract]
  [WebInvoke(Method = "DELETE")]
  void DeleteUser(Guid userId);
}
```

Example 9.15

The `ServiceContract` and `OperationContract` attributes are still there, but we also added `WebInvoke` and `WebGet` attributes. These attributes (originally introduced in .NET framework 3.5) are specific for a REST service implementation and specify operation behaviors.

The `WebInvoke` attribute makes it possible for methods to invoke using the HTTP protocol. This attribute takes some arguments, and the most significant of these is the method argument. The valid values of the method argument correspond to the POST, PUT, and DELETE HTTP methods. The HTTP protocol offers additional methods, but these are the only ones supported by the `WebInvoke` attribute. (The `WebGet` attribute also allows you to specify that a method should be possible to invoke using HTTP GET.)

After creating the interface we again need a class that implements it. Just as with a Web service, we can use all the attributes directly on the class.

Component

Components differ from Web service and REST service implementation options in that they are more technology and platform specific, especially in relation to transport protocols. A component is implemented in a certain language and uses certain frameworks. Therefore, in order to use a component you need to have access to the component technology locally.

Another differentiator is that the service is not called remotely. Rather, you instantiate a component locally and use its API, which is why components are said to be more tightly coupled than Web services and REST services.

In the following example we can use the same class as we used earlier when we implemented a Web service. Instead of calling it remotely as a Web service we use it as follows:

```
UserBankService serviceAPI = new UserBankService();
  serviceAPI.CreateUser(new Core.Models.User()
  {
    UserId = Guid.NewGuid(),
    Address = "MyAdress",
    FirstName = "John",
    LastName = "Smith",
    PhoneNumber = "0332133333",
    SocSecNumber = "730X29"
    }
);
```

Example 9.16

Another WCF Option: Named Pipes

When you develop services in WCF you can also consider the use of named pipes as the transport protocol. This option is similar to using a WCF library as a component because you cannot choose the technology platform for the consumer freely.

The benefit, compared to the component option, is that a service exposed through named pipes runs as an independent process. However, a service exposed using named pipes can only be accessed when the service consumer is installed on the same machine.

Access can be enabled by changing the binding of the service to `NetNamedPipeBinding`.

Dual Protocols with WCF

Although limiting service interaction within a service inventory to one transport protocol is desirable, it can sometimes introduce limitations that make some service requirements hard to fulfill. There may be circumstances that warrant the use of a secondary protocol to complement the primary protocol, as per Dual Protocols [739]. This pattern is commonly applied when the primary protocol introduces performance issues or when a new protocol is introduced as the primary protocol and a period of transition is allowed for the migration of services from the now demoted protocol (the secondary protocol) to the new primary protocol.

WCF enables the application of Dual Protocols [739] by allowing additional endpoints to be added to services via configuration with little or no change to existing code. Configuring a new endpoint is a matter of adding a new address and binding—the `ServiceContract` part can be reused.

SUMMARY OF KEY POINTS

- Canonical Protocol [716] is concerned with establishing baseline interoperability on the transport and messaging protocol layers.
- Different service implementation mediums will generally require different applications of this pattern.
- The Dual Protocols [739] pattern allows for primary and secondary protocols to be standardized within a service inventory.

9.5 Canonical Expression

When a service is created and included in a service inventory, it is important that future service consumers (or rather those humans that develop the software that consume services) will be able to understand the capabilities that the service exposes. To make this easier to understand we utilize naming conventions, as per Canonical Expression [715]. These conventions should be applied to both the name of the service as well as its individual service capabilities.

Service Naming Conventions

The name of a service should generally communicate its functional context. The goal of Canonical Expression [715] is to realize this clarity but also to ensure that all services within a given service inventory are named consistently.

For example, you may want to avoid having an Order service and a PurchaseStatistics service in the same inventory (assuming that Order and Purchase refers to the same thing). To resolve this type of situation, the services could be renamed with "Order" or "Purchase," but not both. As shown in Figure 9.12, this means that the service should be named Purchase or PurchaseStatistics – or – Order or OrderStatistics.

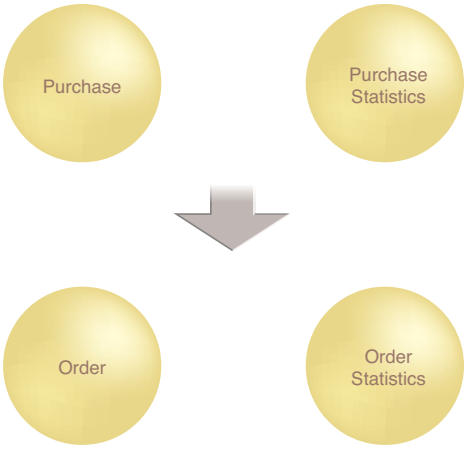


Figure 9.12
Service naming conventions limit the naming options of services.

Service Capability Naming Conventions

Naming service capabilities depends on the type of implementation medium used for the service.

With Web service operations, naming preferences are similar to the naming conventions of methods used with components.

For example, each operation or method name should:

- describe its purpose (preferably using a verb+noun format)
- be as long as necessary
- describe the return value (if there is one)

For Web service operation and component method naming, it is usually required to standardize certain types of wording to ensure consistency. For example, here are some words that could be used interchangeably:

- estimate ⇔ forecast
- forward ⇔ relay
- cease ⇔ finish

For example, we want to avoid having a GetOrderStatus operation alongside a RetrieveStatisticsPerOrderMode operation because "Get" and "Retrieve" can be considered comparable verbs. With naming standards we can ensure that the capabilities will be consistent (like GetOrderStatus and GetStatisticsPerOrderStatus).

For REST services, the focus is on the naming of resources. In order to support consumer requirements, it may be necessary to expose several resources with overlapping naming data. In this case, naming becomes especially important as you wouldn't want consumer designers to misunderstand which resource their program needs to access.

Because REST services rely on the use of Uniform Contract (generally via HTTP methods), the method does not define the naming of the actual capability. The design effort is instead shifted to determining which resources to expose as well as the structure of input and output messages.

SUMMARY OF KEY POINTS

- Canonical Expression [715] aims to establish content consistency across service contract definitions within a service inventory.
- Most commonly, this pattern is applied via naming conventions for services and service capabilities.