THE PRENTICE HALL SERVICE TECHNOLOGY SERIES FROM THOMAS ERL

"This book illuminates the connection of the two domains—SOA and REST—in a manner that is concrete and practical, providing concise application to everyday architectural challenges. Fantastic!"
—Ryan Frazier, Technology Strategist, Microsoft

"...a tour de force that elegantly applies REST principles to the industry-standard SOA framework described in prior titles in this series.... This book is a must-read for anyone developing REST services."
—Dave Slotnick, Enterprise Architect, Rackspace Hosting

"This book undoubtedly will help SOA to reap the benefits from the main value propositions of Web architecture...."
—Dr. Erik Wilde, Architect, EMC Corporation

SOA with REST

Principles, Patterns & Constraints for Building Enterprise Solutions with REST

Co-authored and Edited by Thomas Erl, World's Top-Selling SOA Author
Co-authored by Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian
Foreword by Stefan Tilkov

PRENTICE HALL
ServiceTech PRESS

SOA with REST

Principles, Patterns & Constraints for Building Enterprise Solutions with REST

Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian

PRENTICE HALL

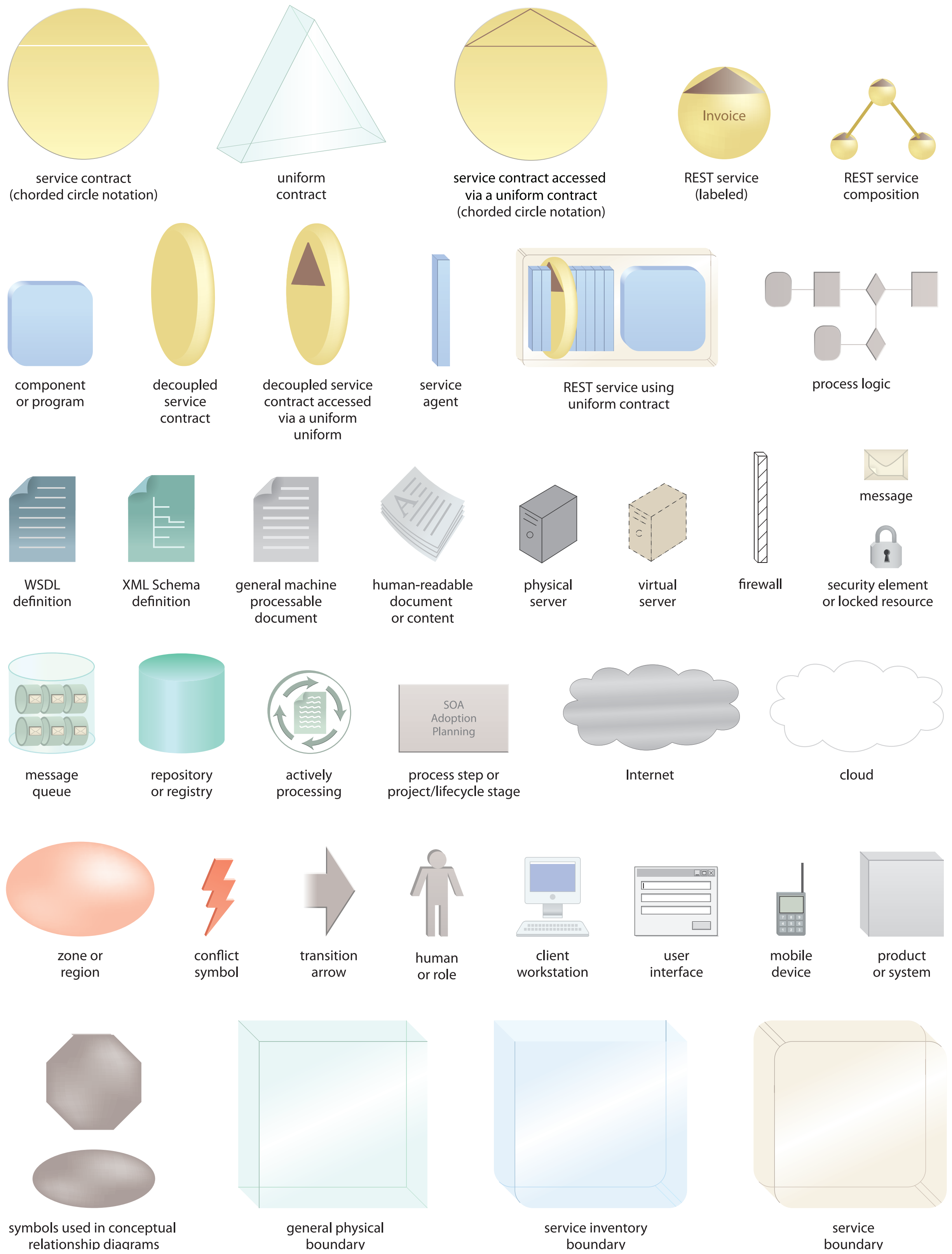
PRENTICE HALL

UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO

NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID

CAPE TOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

ServiceTech PRESS


service contract (chorded circle notation)
uniform contract
service contract accessed via a uniform contract (chorded circle notation)
Invoice
REST service (labeled)
REST service composition
component or program
decoupled service contract
decoupled service contract accessed via a uniform uniform
service agent
REST service using uniform contract
process logic
WSDL definition
XML Schema definition
general machine processable document
human-readable document or content
physical server
virtual server
firewall
message
security element or locked resource
message queue
repository or registry
actively processing
SOA Adoption Planning
process step or project/lifecycle stage
Internet
cloud
zone or region
conflict symbol
transition arrow
human or role
client workstation
user interface
mobile device
product or system
symbols used in conceptual relationship diagrams
general physical boundary
service inventory boundary
service boundary

Contents at a Glance

Chapter 10

Service-Oriented Design with REST

PRINCIPLES, PATTERNS, AND CONSTRAINTS REFERENCED IN THIS CHAPTER:

- Atomic Transaction [432]
- Cache {398}
- Canonical Expression [434]
- Canonical Schema [437]
- Entity Abstraction [463]
- Event-Driven Messaging [465]
- Idempotent Capability [470]
- Layered System {404}
- Legacy Wrapper [473]
- Logic Centralization [475]
- Process Abstraction [486]
- Service Abstraction (414)
- Service Discoverability (420)
- Service Loose Coupling (413)
- Stateless {395}
- Uniform Contract {400}
- Utility Abstraction [517]
- Validation Abstraction [518]

Using the conceptual service candidates modeled during the preceding service-oriented analysis process as a starting point, service-oriented design is dedicated to the physical design of service contracts. When it comes to contract design with REST, we need to be concerned with two particular areas:

1. The design of a uniform contract for a service inventory.
2. The design of individual service contracts within the service inventory and in compliance with the uniform contract.

The uniform contract needs to be firmly established before we begin creating service contracts that will be required to form dependencies on uniform contract features. As a service inventory grows and evolves, new services can still influence the design of a uniform contract, but uniform contract features are generally changed and added at a very deliberate pace.

Following the preceding sequence, this chapter begins with coverage of uniform contract design topics and then moves on to topics that pertain to the design of REST service contracts. The chapter concludes with a section on complex methods, an optional field of REST contract design and one suitable mainly for use within controlled environments, such as internal service inventories.

10.1 Uniform Contract Design Considerations

When creating a uniform contract for a service inventory, we have a responsibility to equip and limit its features so that it is streamlined to effectively accommodate requirements and restrictions unique to the service inventory. The default characteristics of Web-centric technology architecture can provide an effective basis for a service inventory's uniform contract, although additional forms of standardization and customization are likely to be required for non-trivial service inventory architectures.

The following sections explore how common elements of a uniform contract (methods, media types, and exceptions in particular) can be customized to meet the needs of individual service inventories.

Designing and Standardizing Methods

When we discuss methods in relation to the uniform contract, it is considered shorthand for a request-response communications mechanism that also includes methods, headers, response codes, and exceptions. Methods are centralized as part of the uniform contract in order to ensure that there are always a small number of ways of moving information around within a particular service inventory, and that existing service consumers will work correctly with new or modified services as they are added to the inventory. Whereas it is important to minimize the number of methods in the uniform contract, methods can and should be added when service inventory interaction requirements demand it. This is a natural part of evolving a service inventory in response to business change.

HTTP provides a solid foundation by supplying the basic set of methods (such as GET, PUT, DELETE, POST) proven by use on the Web and widely supported by off-the-shelf software components and hardware devices. But the need may arise to express other types of interactions for a service inventory. For example, you may decide to add a special method that can be used to reliably trigger a resource to execute a task at most once, rather than using the less reliable HTTP POST method.

HTTP is designed to be extended in these ways. The HTTP specification explicitly supports the notion of extension methods, customized headers, and extensibility in other areas. Leveraging this feature of HTTP can be effective, as long as new extensions are added carefully and at a rate appropriate for the number of services that implement HTTP within an inventory. This way, the total number of options for moving data around (that services and consumers are required to understand) remains manageable.

Less well-known HTTP methods have come and gone in the past. For example, at various times the HTTP specification has included a PATCH method consistent with a partial update or partial store communications mechanism. PATCH is currently specified separately from HTTP methods in the IETF's RFC 5789 document. Other IETF specifications, such as WebDAV's RFC 4918 and the Session Initiation Protocol's RFC 3261, introduced new methods as well as new headers and response codes (or special interpretations thereof).

NOTE

Later in this chapter we explore a set of sample, extended methods (referred to as *complex methods*), each of which utilizes multiple basic HTTP methods or utilizes a single basic HTTP method multiple times, to perform pre-defined, standardized interactions.

Common circumstances that can warrant the creation of new methods include:

- Hyperlinks may be used to facilitate a sequence of request-response pairs. When they start to read like verbs instead of nouns and tend to suggest that only a single method will be valid on the target of a hyperlink, we can consider introducing a new method instead. For example the “customer” hyperlink for an invoice resource suggests that GET and PUT requests might be equally valid for the customer resource. But a “begin transaction” hyperlink or a “subscribe” hyperlink suggest only POST is valid and may indicate the need for a new method instead.
- Data with must-understand semantics may be needed within message headers. In this case, a service that ignores this metadata can cause incorrect runtime behavior. HTTP does not include a facility for identifying individual headers or information within headers as “must-understand.” A new method can be used to enforce this requirement because the custom method will be automatically rejected by a service that doesn’t understand the request (whereas falling back on a default HTTP method will allow the service to ignore the new header information).

It is important to acknowledge that introducing custom methods can have negative impacts when exploring vendor diversity within an implementation environment. It may prevent off-the-shelf components (such as caches, load balancers, firewalls, and various HTTP-based software frameworks) from being fully functional within the service inventory. Stepping away from HTTP and its default methods should only be attempted in mature service inventories when the effects on the underlying technology architecture and infrastructure are fully understood.

Some alternatives to creating new methods can also be explored. For example, service interactions that require a number of steps can use hyperlinks to guide consumers through the requests they need to make. The HTTP Link header (RFC 5988) can be considered to keep these hyperlinks separate from the actual document content.

Designing and Standardizing HTTP Headers

Exchanging messages with metadata is common in service-oriented solution design. Because of the emphasis of composing a set of services together to collectively automate a given task at runtime, there is often a need for a message to provide a range of header information that pertains to how the message should be processed by intermediary service agents and services along its message path.

Built-in HTTP headers can be used in a number of ways:

- They can be used to add parameters related to a request method as an alternative to using query strings to represent the parameters within the URL. For example, the `Accept` header can supplement the GET method by providing content negotiation data.
- They can be used to add parameters related to a response code. For example the `Location` header can be used with the `201 Created` response code to indicate the identifier of a newly created resource.
- They can be used to communicate general information about the service or consumer. For example the `Upgrade` header can indicate that a service consumer supports and prefers a different protocol, while the `Referrer` header can indicate which resource the consumer came from while following a series of hyperlinks.

This type of general metadata may be used in conjunction with any HTTP method.

HTTP headers can also be utilized to add rich metadata. For this purpose custom headers are generally required, which re-introduces the need to determine whether or not the message content must be understood by recipients or whether it can optionally be ignored. This association of must-understand semantics with new methods and must-ignore semantics with new message headers is not an inherent feature of REST, but it is a feature of HTTP.

When introducing custom HTTP headers that can be ignored by services, regular HTTP methods can safely be used. This also makes the use of custom headers backwards-compatible when creating new versions of existing message types.

As previously stated in the *Designing and Standardizing Methods* section, new HTTP methods can be introduced to enforce must-understand content by requiring services to either be designed to support the custom method or to reject the method invocation attempt altogether. In support of this behavior, a new `Must-Understand` header can be created in the same format as the existing `Connection` header, which would list all of the headers that need to be understood by message recipients.

If this type of modification is made to HTTP, it would be the responsibility of the SOA Governance Program Office responsible for the service inventory to ensure that these semantics are implemented consistently as part of inventory-wide design standards. If custom, must-understand HTTP headers are successfully established within a service inventory, we can explore a range of applications of messaging metadata. For example, we can determine whether it is possible or feasible to emulate messaging metadata such as what is commonly used in SOAP messaging frameworks based on WS-* standards.

While custom headers that enforce reliability or routing content (as per the WS-ReliableMessaging and WS-Addressing standards) can be added to recreate acknowledgement and intelligent load balancing interactions, other forms of WS-* functions are subject to built-in limitations of the HTTP protocol. The most prominent example is the use of WS-Security to enable message-level security features, such as encryption and digital signatures. Message-level security protects messages by actually transforming the content so that intermediaries along a message path are unable to read or alter message content. Only those message recipients with prior authorization are able to access the content.

This type of message transformation is not supported in HTTP/1.1. HTTP does have some basic features for transforming the body of the message alone through its `Content-Encoding` header, but this is generally limited to compression of the message body and does not include the transformation of headers. If this feature was used for encryption purposes the meaning of the message could still be modified or inspected in transit, even though the body part of the message could be protected. Message signatures are also not possible in HTTP/1.1 as there is no canonical form for an HTTP message to sign, and no industry standard that determines what modifications intermediaries would be allowed to make to such a message.

Designing and Standardizing HTTP Response Codes

HTTP was originally designed as a synchronous, client-server protocol for the exchange of HTML pages over the World Wide Web. These characteristics are compatible with REST constraints and make it also suitable as a protocol used to invoke REST service capabilities.

Developing a service using HTTP is very similar to publishing dynamic content on a Web server. Each HTTP request invokes a REST service capability and that invocation concludes with the sending of a response message back to the service consumer.

A given response message can contain any one of a wide variety of HTTP codes, each of which has a designated number. Certain ranges of code numbers are associated with particular types of conditions, as follows:

- `100-199` are informational codes used as low level signaling mechanisms, such as a confirmation of a request to change protocols.

- `200-299` are general success codes used to describe various kinds of success conditions.
- `300-399` are redirection codes used to request that the consumer retry a request to a different resource identifier, or via a different intermediary.
- `400-499` represent consumer-side error codes that indicate that the consumer has produced a request that is invalid for some reason.
- `500-599` represent service-side error codes that indicate that the consumer's request may have been valid but that the service has been unable to process it for internal reasons.

The consumer-side and service-side exception categories are helpful for "assigning blame," but do little to actually enable service consumers to recover from failure. This is because, while the codes and reasons provided by HTTP are standardized, how service consumers are required to behave upon receiving response codes is not. When standardizing service design for a service inventory, it is necessary to establish a set of conventions that assign response codes concrete meaning and treatment.

Table 10.1 provides common descriptions of how service consumers can be designed to respond to common response codes.

Response Code	Reason Phrase	Treatment
`100`	Continue	Indeterminate
`101`	Switching Protocols	Indeterminate
`1xx`	Any other 1xx code	Failure
`200`	OK	Success
`201`	Created	Success
`202`	Accepted	Success
`203`	Non-Authoritative Information	Success
`204`	No Content	Success

Response Code	Reason Phrase	Treatment
`205`	Reset Content	Success
`206`	Partial Content	Success
`2xx`	Any other 2xx code	Success
`300`	Multiple Choices	Failure
`301`	Moved Permanently	Indeterminate (Common Behavior: Modify resource identifier and retry.)
`302`	Found	Indeterminate (Common Behavior: Change request to a GET and retry using nominated resource identifier.)
`303`	See Other	
`304`	Not Modified	Success (Common Behavior: Use cached response.)
`305`	Use Proxy	Indeterminate (Common Behavior: Connect to identified proxy and resend original message.)
`307`	Temporary Redirect	Indeterminate (Common Behavior: Retry once to nominated resource identifier.)
`3xx`	Any other 3xx code	Failure
`400`	Bad Request	Failure

continues

Response Code	Reason Phrase	Treatment
`401`	Unauthorized	Indeterminate (Common Behavior: Retry with correct credentials.)
`402`	Payment Required	Failure
`403`	Forbidden	Failure
`404`	Not Found	Success if request was DELETE, else Failure
`405`	Method Not Allowed	Failure
`406`	Not Acceptable	Failure
`407`	Proxy Authentication Required	Indeterminate (Common Behavior: Retry with correct credentials.)
`408`	Request Timeout	Failure
`409`	Conflict	Failure
`410`	Gone	Success if request was DELETE, else Failure
`411`	Length Required	Failure
`412`	Precondition Failed	Failure
`413`	Request Entity Too Large	Failure
`414`	Request-URI Too Long	Failure
`415`	Unsupported Media Type	Failure
`416`	Requested Range Not Satisfiable	Failure
`417`	Expectation Failed	Failure

Response Code	Reason Phrase	Treatment
`4xx`	Any other 4xx code	Failure
`500`	Internal Server Error	Failure
`501`	Not Implemented	Failure
`502`	Bad Gateway	Failure
`503`	Service Unavailable	Repeat if Retry-After header is specified. Otherwise, Failure.
`504`	Gateway Timeout	Repeat if request is idempotent. Otherwise, Failure.
`505`	HTTP Version Not Supported	Failure
`5xx`	Any other 5xx code	Failure

Table 10.1
HTTP response codes, and typical corresponding consumer behavior.

As is evident when reviewing Table 10.1, HTTP response codes go well beyond the simple distinction between success and failure. They provide an indication of how consumers can respond to and recover from exceptions.

Let's take a closer look at some of the values from the Treatment column in Table 10.1:

- *Repeat* means that the consumer is encouraged to repeat the request, taking into account any delay specified in responses such as `503 Service Unavailable`. This may mean sleeping before trying again. If the consumer chooses not to repeat the request, it must treat the method as failed.
- *Success* means the consumer should treat the message transmission as a successful action and must therefore not repeat it. (Note that specific success codes may require more subtle interpretation.)
- *Failed* means that the consumer must not repeat the request unchanged, although it may issue a new request that takes the response into account. The consumer should treat this as a failed method if a new request cannot be generated. (Note that specific failure codes may require more subtle interpretation.)

- *Indeterminate* means that the consumer needs to modify its request in the manner indicated. The request must not be repeated unchanged and a new request that takes the response into account should be generated. The final outcome of the interaction will depend on the new request. If the consumer is unable to generate a new request, then this code must be treated as failed.

Because HTTP is a protocol, not a set of message processing logic, it is up to the service to decide what status code (success, failure, or otherwise) to return. As previously mentioned, because consumer behavior is not always sufficiently standardized by HTTP for machine-to-machine interactions, it needs to be explicitly and meaningfully standardized as part of an SOA project.

For example, indeterminate codes tend to indicate that service consumers must handle a situation using their own custom logic. We can standardize these types of codes in two ways:

- Design standards can determine which indeterminate codes can and cannot be issued by service logic.
- Design standards can determine how service consumer logic must interpret those indeterminate codes that are allowed.

Customizing Response Codes

The HTTP specification allows for extensions to response codes. This extension feature is primarily there to allow future versions of HTTP to introduce new codes. It is also used by some other specifications (such as WebDAV) to define custom codes. This is typically done with numbers that are not likely to collide with new HTTP codes, which can be achieved by putting them near the end of the particular range (for example, 299 is unlikely to ever be used by the main HTTP standard).

Specific service inventories can follow this approach by introducing custom response codes as part of the service inventory design standards. In support of the Uniform Contract {400} constraint, custom response codes should only be defined at the uniform contract level, not at the REST service contract level.

When creating custom response codes, it is important that they be numbered based on the range they fall in. For example, 2xx codes should be communicating success, while 4xx codes should only represent failure conditions.

Additionally, it is good practice to standardize the insertion of human-readable content into the HTTP response message via the Reason Phrase. For example, the code 400 has a default reason phrase of “Bad Request.” This is enough for a service consumer to handle the response as a general failure, but it doesn’t tell a human anything useful about the actual problem. Setting the reason phrase to “The service consumer request is missing the Customer address field.” or perhaps even “Request body failed validation against schema http://example.com/customer” is more helpful, especially when reviewing logs of exception conditions that may not have the full document attached.

Consumers can associate generic logic to handle response codes in each of these ranges, but may also need to associate specific logic to specific codes. Some codes can be limited so that they are only generated if the consumer requests a special feature of HTTP, which means that some codes can be left unimplemented by consumers that do not request these features.

Uniform contract exceptions are generally standardized within the context of a particular new type of interaction that is required between services and consumers. They will typically be introduced along with one or more new methods and/or headers. This context will guide the kind of exceptions that are created. For example, it may be necessary to introduce a new response code to indicate that a request cannot be fulfilled due to a lock on a resource. (WebDAV provides the `423 Locked` code for this purpose.)

When introducing and standardizing custom response codes for a service inventory uniform contract we need to ensure that:

- each custom code is appropriate and absolutely necessary
- the custom code is generic and highly reusable by services
- the extent to which service consumer behavior is regulated is not too restrictive so that the code can apply to a large range of potential situations
- code values are set to avoid potential collision with response codes from relevant external protocol specifications
- code values are set to avoid collision with custom codes from other service inventories (in support of potential cross-service inventory message exchanges that may be required)

Response code numeric ranges can be considered a form of exception inheritance. Any code within a particular range is expected to be handled by a default set of logic, just as if the range were the parent type for each exception within that range.

In this section, we have briefly explored response codes within the context of HTTP. However, it is worth noting that REST can be applied with other protocols (and other exception models). It is ultimately the base protocol of a service inventory architecture that will determine how normal and exceptional conditions are reported.

For example, you could consider having a REST-based service inventory standardized on the use of SOAP messages that result in SOAP-based exceptions instead of HTTP exception codes. This allows the response code ranges to be substituted for inheritance of exceptions.

Designing Media Types

During the lifetime of a service inventory architecture we can expect more changes will be required to the set of a uniform contract's media types than to its methods. For example, a new media type will be required whenever a service or consumer needs to communicate machine-readable information that does not match the format or schema requirements of any existing media type.

Some common media types from the Web to consider for service inventories and service contracts include:

- `text/plain; charset=utf-8` for simple representations, such as integer and string data. Primitive data can be encoded as strings, as per built-in XML Schema data types.
- `application/xhtml+xml` for more complex lists, tables, human-readable text, hypermedia links with explicit relationship types, and additional data based on microformats.org and other specifications.
- `text/uri-list` for plain lists of URIs.
- `application/atom+xml` for feeds of human-readable event information or other data collections that are time-related (or time ordered).

More standard media types can be found in the IANA media type registry, as explained in Appendix B. Before inventing new media types for use within a service inventory, it is advisable to first carry out a search of established industry media types that may be suitable.

Whether choosing existing media types or creating custom ones, it is helpful to consider the following best practices:

- Each specific media type should ideally be specific to a schema. For example, `application/xml` or `application/json` are not schema-specific, while `application/atom+xml` used as a syndication format is specific enough to be useful for content negotiation and to identify how to process documents.
- Media types should be abstract in that they specify only as much information as their recipients need to extract via their schemas. Keeping media types abstract allows them to be reused within more service contracts.
- New media types should reuse mature vocabularies and concepts from industry specifications whenever appropriate. This reduces the risk that key concepts have been missed or poorly constructed, and further improves compatibility with other applications of the same vocabularies.
- A media type should include a hyperlink whenever it needs to refer to a related resource whose representation is located outside the immediate document. Link relation types may be defined by the media type's schema or, in some cases, separately, as part of a link relation profile.
- Custom media types should be defined with must-ignore semantics or other extension points that allow new data to be added to future versions of the media type without old services and consumers rejecting the new version.
- Media types should be defined with standard processing instructions that describe how a new processor should handle old documents that may be missing some information. Usually these processing instructions ensure that earlier versions of a document have compatible semantics. This way, new services and consumers do not have to reject the old versions.

All media types that are either invented for a particular service inventory or reused from another source should be documented in the uniform contract profile, alongside the definition of uniform methods.

HTTP uses Internet media type identifiers that conform to a specific syntax. Custom media types are usually identified with the notation:

```
application/vnd.organization.type+supertype
```

…where `application` is a common prefix that indicates that the type is used for machine consumption and standards. The `organization` field identifies the vendor namespace, which can optionally be registered with IANA.

The `type` part is a unique name for the media type within the organization, while the `supertype` indicates that this type is a refinement of another media type. For example, `application/vnd.com.examplebooks.purchase-order+xml` may indicate that:

- the type is meant for machine consumption
- the type is vendor-specific, and the organization that has defined the type is "examplebooks.com"
- the type is for purchase orders (and may be associated with a canonical Purchase Order XML schema)
- the type is derived from XML, meaning that recipients can unambiguously handle the content with XML parsers

Types meant for more general inter-organizational use can be defined with the media type namespace of the organization ultimately responsible for defining the type. Alternatively, they can be defined without the vendor identification information in place by registering each type directly, following the process defined in the RFC 4288 specification.

Designing Schemas for Media Types

Within a service inventory, most custom media types created to represent business data and documents will be defined together with XML schemas. This essentially applies the Canonical Schema [437] pattern in that it establishes a set of standardized data models that are reused by REST services within the inventory to whatever extent feasible.

For this to be successful, especially with larger collections of services, schemas need to be designed to be flexible. This means that it is generally preferable for schemas to enforce a coarse level of validation constraint granularity that allows each schema to be applicable for use with a broader range of data interaction requirements.

REST requires media types and their schemas to be defined only at the uniform contract level. If a service capability requires a unique data structure for a response message, it must still use one of the canonical media types provided by the uniform contract. Designing schemas to be flexible and weakly typed can accommodate a variety of service-specific message exchange requirements.

NOTE

To explore techniques for weakly typing XML Schema definitions, see Chapters 6, 12, and 13 in the book *Web Service Contract Design & Versioning for SOA*, as well as the description for the Validation Abstraction [518] pattern.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.com/schema/po"
  xmlns="http://example.com/schema/po">
  <xsd:element name="LineItemList" type="LineItemListType"/>
  <xsd:complexType name="LineItemListType">
    <xsd:element name="LineItem" type="LineItemType"
      minOccurs="0"/>
  </xsd:complexType>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:anyURI"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.1

One of the most straightforward ways of making a media type more reusable is to design the schema to support a list of zero or more items. This enables the media type to permit one instance of the underlying type, but also allows queries that return zero or more instances. Making individual elements within the document optional can also increase reuse potential.

Service-Specific XML Schemas

It is technically possible for individual REST service contracts to introduce contract-specific XML schemas, but in doing so we need to accept that the Uniform Contract {400} constraint will be violated.

This may be warranted when a service capability needs to generate a response message containing unique data (or a unique combination of data) for which:

- no suitable canonical schemas exist
- no new canonical schema should be created due to the fact that it would not be reusable by other services.

A consequence of non-compliance to Uniform Contract {400} is potentially increased levels of negative coupling between service consumers and the service offering service capabilities based on service-specific media types. Service-specific media types should be clearly identified and effort should be made to minimize the quantity of logic that is directly exposed to and made dependent upon these types.

SUMMARY OF KEY POINTS

- We can design and standardize custom HTTP methods and response codes. We can also standardize how built-in HTTP methods and response codes are used (or whether they are used).
- There are numerous existing media types we can choose to use (and reuse) within a service inventory, many of which are registered with the IANA (and other industry bodies). We can also design and standardize custom media types to represent common types of data and documents that are exchanged within the service inventory.
- Schemas encompassed by media types are naturally standardized when made part of a uniform contract. For schemas to be reusable, they generally need to be designed with flexibility in mind, ensuring reduced levels of validation constraint granularity.

10.2 REST Service Contract Design

This next section explores design techniques and considerations specific to individual REST service contracts and how they relate to their overarching uniform contract.

Designing Services Based on Service Models

In Chapters 4 and 9 we described the three common service models used to establish base functional contexts that categorize and group services within a service inventory into three common logical layers. The choice of service model for a given REST service can affect our approach to service contract design. The following sections briefly raise some key considerations and provide one sample REST service contract design for each service model.

Task Services

Task services will typically have few service capabilities, sometimes limited to only a single one (Figure 10.1). This is due to the fact that a task service contract's primary use is for the execution of automated business process (or task) logic. The service capability can be based on a simple verb, such as Start or Process. That verb, together with the name of the task service (that will indicate the nature of the task) is often all that is required for synchronous tasks.

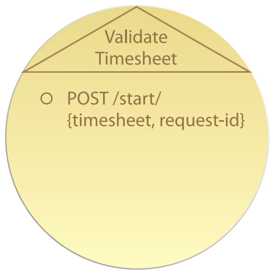


Figure 10.1
A sample task service, recognizable by the verb in its name. The contract only provides a single service capability that will be used by the composition initiator to trigger the execution of the Validate Timesheet business process that the task service logic encapsulates. In this case, the service capability receives a timesheet resource identifier that will be used as the basis of the validation logic, plus a unique consumer-generated request identifier that supports reliable triggering of the process. (Note that the composition initiator is explained in Chapter 11.)

Additional service capabilities can be added to support asynchronous interactions as shown in Figure 10.2. For example, tasks that involve human interaction or batch processing will retain the state of the on-going business process between requests and will typically allow access to this state by exposing service capabilities for this purpose.

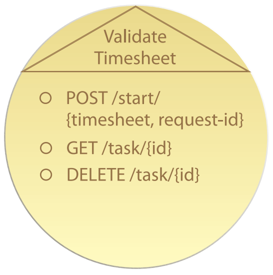


Figure 10.2
Two additional service capabilities are added to allow consumers to asynchronously check on the progress of the timesheet validation task, and to cancel the task while it is in progress.

REST-based task services will often have service capabilities triggered by a POST request. However, this method is not inherently reliable. A number of techniques exist to achieve a reliable POST, including the inclusion of additional headers and handling of response messages, or the inclusion of a unique consumer-generated request identifier in the resource identifier.

To provide input to a parameterized task service it will make sense for the task service contract to include various identifiers into the capability's resource identifier template (that might have been parameters in a SOAP message). This frees up the service to expose additional resources rather than defining a custom media type as input to its processing.

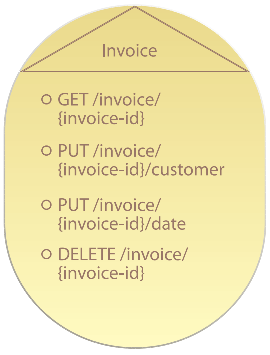
If the task service automates a long-running business process it will return an interim response to its consumer while further processing steps may still need to take place. If the task service includes additional capabilities to check on or interact with the state of the business process (or composition instance), it will typically include a hyperlink to one or more resources related to this state in the initial response message.

Entity Services

Each entity service establishes a functional boundary associated with one or more related business entities (such as invoice, claim, customer, etc.). Entity services are the prime means by which Logic Centralization [475] is applied to business logic within a service inventory. The types of service capabilities exposed by a typical entity service are focused on functions that process the underlying data associated with the entity (or entities). Figure 10.3 provides some examples.

Entity service contracts are typically dominated by service capabilities that include inherently idempotent and reliable GET, PUT, or DELETE methods. However, more complex methods may be needed. Many entity services will need to support updating

Figure 10.3
An entity service based on the invoice business entity that defines a functional scope that limits the service capabilities to performing invoice-related processing only. This agnostic Invoice service will be reused and composed by any automated business process that needs to work with or process invoice records. For example, the Invoice service may be invoked by the Validate Timesheet task service to retrieve invoice data linked to client information collected from a timesheet record. The Validate Timesheet service may then use this data to verify that what the client was billed matches what the employee logged in the timesheet.


Invoice
GET /invoice/{invoice-id}
PUT /invoice/{invoice-id}/customer
PUT /invoice/{invoice-id}/date
DELETE /invoice/{invoice-id}

their state consistently with changes to other entity services. Entity services will also often include query capabilities for finding entities or parts of entities that match certain criteria, and therefore return hyperlinks to related and relevant entities.

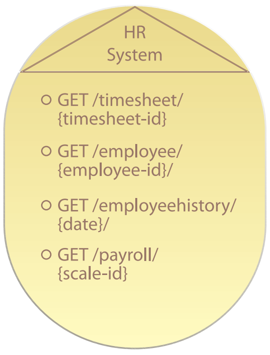
Utility Services

Utility services are, like entity services, expected to be agnostic and reusable. However, unlike entity services, they do not usually have pre-defined functional scopes. While individual utility services group related service capabilities, the services' functional boundaries can vary dramatically. The example illustrated in Figure 10.4 is a utility service acting as a wrapper for a legacy system (as per the Legacy Wrapper [473] pattern).

NOTE

To learn more about service models and service layers, see the Process Abstraction [486], Entity Abstraction [463], and Utility Abstraction [517] patterns.

Figure 10.4
This utility service is based on the application of the Legacy Wrapper [473] pattern in that it provides a service contract that encapsulates a legacy HR system (and is accordingly named the HR System service). The service capabilities it exposes provide generic, read-only data access functions against the data stored in the underlying legacy repository. For example, the Employee entity service (composed by the Verify Timesheet task service) may invoke an employee data-related service capability to retrieve data. This type of utility service may provide access to one of several available sources of employee and HR-related data.


HR System
GET /timesheet/{timesheet-id}
GET /employee/{employee-id}/
GET /employeehistory/{date}/
GET /payroll/{scale-id}

Designing and Standardizing Resource Identifiers

The fundamental requirement of an effective REST service contract design is its ability to express the identity of resources that consumers can interact with as part of their service capability invocations.

At a technical level the structure of a resource identifier is often irrelevant to a service consumer. Any service consumer that follows a simple hyperlink only cares that the destination of the hyperlink is the correct resource. It doesn't try to interpret the meaning of the resource identifier itself, past the information required to actually make the connection to the responsible service.

With that said there are a number of reasons we proceed past the point of standardizing the syntax of resource identifiers to the point of standardizing structure and vocabulary within resource identifiers:

1. The more descriptive and consistent the structure of resource identifiers is for similar service capabilities, the easier it is for humans to interpret and understand services and their capabilities. This directly supports the application of Service Discoverability (420).
2. Some resource identifier structures lend themselves better to the future needs of their service contract than others. They do so by providing obvious places where additional resources and related capabilities can be inserted in the resource identifier namespace.
3. Designing flexible resource identifiers can reduce negative coupling, while increasing backwards compatibility and, potentially, forwards compatibility (as explained in Chapter 15).
4. In some cases, service consumers need to insert information into resource identifiers, either by adding data values into the query component of a URL using a standard syntax, or by following a URL template to insert data throughout the URL. If the vocabulary is not reusable between multiple services then these variable portions of URLs become a back door for negative forms of coupling between the consumers to the service contract.

The latter two items directly support the application of the Service Loose Coupling (413) principle.

Service Names in Resource Identifiers

The first area of standardization we'll explore is the use of service names within resource identifier statements. This brings us back to the study of URI syntax, which we began in the *URIs (and URLs and URNs)* section in Chapter 6. Briefly revisit this section to re-familiarize yourself with the examples.

In the last example provided in this section:

```
invoices.example.com
```

… identifies the service within the URL:

```
http://invoices.example.com/
```

Another service:

```
customers.example.com
```

… may initially share the same IP address as:

```
invoices.example.com
```

… as a result of being deployed in a shared hosting environment.

When `customers.example.com` is moved to its own separate physical hardware, the IP addresses can be easily updated via the Domain Name System (DNS) without modifying the logical name of the service. Consumers that refer to `customers.example.com` will automatically begin communicating with the new IP address, and therefore will place no further burden on the old hosting environment.

If, instead, the service names were part of the *path* of the URL, the authority would have to refer to the hosting environment itself.

A URL for the Invoice service that starts with:

```
http://services.example.com/invoice
```

… would always resolve to the IP address of:

```
services.example.com
```

… rather than a specific IP address for the service.

If the Customer service were then moved to a new hosting environment, all of the hyperlinks held by service consumers would have to change or the requests sent to the service would still have to continue passing through the `services.example.com` host.

When combining REST with service-orientation, the authority needs to be synonymous with the service name in order to maximize the application potential of the Service Autonomy (SDP) principle. The authority is always what is looked up by the service consumer so that it can make the necessary TCP/IP connections. It is also used to identify proxies between the service and its consumers. Sometimes, multiple services will be hosted within the same virtual server or cluster, and these service names will resolve to the same IP address. But, by ensuring that each service has a unique authority, the service can be easily shifted to other IP addresses as service deployment arrangements change.

Other URI Components

The path and query components of the URI provide context for service capabilities within a given service. This context is combined with the service identifier in the authority and with the method of each request to determine which service capability a given consumer seeks to invoke.

The `{fragment}` component of the URI reference is never sent to the service, and is only used as a placeholder to store instructions for the service consumer to know how to process the response when it arrives. If a service consumer needs to use the aforementioned URI reference to invoke a GET request, it would send the part of the URI reference up to and including the query. The `{fragment}` component would be intentionally omitted. For example, a `page2` fragment may indicate to the service consumer that it should start processing at page 2 in the returned document. Where exactly to find such a point in the document depends on the media type of the document.

If some of the components of a URI are missing, the URI reference may become a relative URI. In that case, the context of the URI is used to determine what exactly it is pointing to.

For example, a relative URI of:

```
/invoices/INV042
```

… would be expanded as:

```
http://invoice.example.com/invoices/INV042
```

Relative URIs are often a useful way to refer to related resources without referring to additional context, such as the name of the service. The base URI to resolve a relative URI against can come from XML directives, HTTP headers, the location that a document was retrieved from, or from a range of other sources, depending on the conventions associated with the media type in use.

Resource Identifier Overlap

Resources can be any specific utility, entity, task, queue, report, statistic, or in fact anything related to the service that can be referred to in a context. The identifier for a resource can contain as much or as little context as is needed to specify the concept the resource embodies. A resource could be "today's weather in Vancouver, Canada." A separate resource could capture "yesterday's weather in Vancouver, Canada" while yet another family of resources could capture the weather in Vancouver for specific historical dates. The concepts that resources embody will sometimes overlap, so that some or all of the same data is retrieved via different resource identifiers. Other resources will encapsulate concepts that are distinct in their own right and do not overlap.

We can imagine that a URI such as:

```
http://weather/canada/vancouver/date/today
```

… will return the same value when retrieved as the URI:

```
http://weather/canada/vancouver/date/{date}
```

… with a date set to today's date. However, these are different resources and perhaps even different service capabilities. When the date switches over to the next day, the `today` resource will point to the new day's weather. The resource based on the old `{date}` will still refer to the historical weather at that particular date.

Similarly, an invoice might appear as its own URI but may also have its data summarized as part of an invoice list or report resource. The invoice URI may further have subordinate resources, such as a special resource indicating its paid status. In all of these cases, the URIs are different, but the data and the service logic that implement requests to each one overlap.

The context of the resource as identified in its URI may be dynamic or session-specific. For example, the following URI:

```
http://mybank/accounts/myaccount?after=XACT102
```

… may refer to the transactions in a bank account that occurred after transaction number 102. This may have been returned from the service to a particular consumer as a placeholder between transactions the consumer has reconciled and those that have not yet been reconciled. This kind of resource captures session information and acts as a container for session state, allowing the service to avoid having to retain these details.

Queries can also be encapsulated in resource identifiers. Query terms, such as a required temperature range or the maximum temperature value past a particular date, can be included in a resource identifier. When the first request is sent to this identifier the resource automatically springs into existence, performs its processing, and returns a result. The consumer and any middleware are not aware of whether a resource is static or dynamic. The interface to the resource does not change, and features such as caching, work just as well with static and dynamic resources. The implementation of each resource is hidden from consumers.

In Chapter 6 we covered the use of forms and resource identifier templates to allow service consumers to input parameters into resource identifiers without introducing service-specific coupling. If URIs are being constructed by human users, forms can be provided for them to fill out as part of producing the URI. This does not introduce tight coupling between the service and service consumer, as the consumer does not need to understand the data that passes through it. Only the human user needs to determine what data to place in which form fields. However, a service consumer that is not being driven by a human user will need to know which specific variables to insert into a given resource identifier. If automated service consumers are supplying parameters directly as part of URIs, it is advisable to clearly differentiate between elements of the URI that consumers are considered likely to have, in order to populate themselves and to identify these fields in a way that is documented in the uniform contract profile for the service inventory.

For example, the aforementioned bank account URI:

```
http://mybank/accounts/myaccount?after=XACT102
```

… suggests to readers of the service contract that it is likely to be the service consumer that fills out the `after` field within the URI. In order for an automated service consumer to avoid tight coupling with the service contract, the `after` field should become part of the uniform contract. When `after` is used, it should have the same meaning, regardless of which service the consumer is talking to.

NOTE

Resource identifiers contain data for their corresponding services to interpret. In order to invoke the correct service capability, the business context of a request must be specified by the consumer and understood by the service. As explained in previous chapters, resource identifiers can be discovered by a service consumer through hyperlinking, or by direct entry of resource identifiers into configuration data. In these cases the resource identifier can usually be treated as opaque by the service consumer. The consumer does not attempt to parse information out of the identifier, nor does it need to insert additional information. Resource identifier templates allow consumers to insert data into resource identifiers in predefined ways, while treating the overall structure of the resource identifiers as being opaque.

Resource identifiers that are handled in this manner by the service consumer act as a message from the service that is held onto by the consumer and passed back to the service with subsequent requests. These messages can contain identifiers for entities, session state data, or any other data the service will need the next time a request comes in for processing. Treating resource identifiers as opaque within service consumers means that we can reduce (or loosen) the coupling between a service and its consumers. The service can change the content or structure of its resource identifiers without needing corresponding changes to service consumer logic.

Resource Identifier Design Guidelines

Here are a few tips for optimizing resource identifiers in support of SOA. Each of these can form the basis of a design standard in support of the Canonical Expression [434] pattern:

- Try to avoid including a variable part of the URL as the first path segment, or anywhere not preceded by a static path segment describing the context. For example, avoid `http://invoice.example.com/{invoice}`. Although we may use this type of notation for simplicity's sake early in the service capability modeling lifecycle, once we enter the service-oriented design stage it can make it difficult to extend the namespace. This is because any new path could be interpreted as including an invoice identifier. Consider introducing a prefix to qualify the variable part, for example using `http://invoice.example.com/invoice/{invoice}`, instead.

- Trailing slashes usually indicate a collection of resources. One common convention is that a GET request to a URL with a trailing slash will retrieve a list of these resources, while a POST to the URL will create a new resource. For example, `http://invoice.example.com/unpaid/` may support a GET request to obtain all unpaid invoices, while a POST to `http://invoice.example.com/invoice/` may create an invoice with a resource identifier of `http://invoice.example.com/invoice/INV042`. This again is a departure from the notation used during the service-oriented analysis project stage, where we use the trailing slash, together with the initial slash, as delimiters to represent a resource.
- Single out those resource identifiers that are canonical names (URNs), and make these URLs as simple as possible. Avoid including a query component in the resource identifier and avoid special characters such as ';', '=', and '&'. For example, `http://invoice.example.com/?invoice=INV042` is not a good identifier for invoice number 42, while `http://invoice.example.com/invoice/INV042` is a better choice. Simpler identifiers are easier to embed into other resource identifiers, and easier for a human to read and understand; a prime requirement of the Service Discoverability (420) principle.
- Always refer to canonical names (URNs) by their full resource identifier. For example, the `http://invoice.example.com/query{?customer}` URL should be expanded to `http://invoice.example.com/query?customer=http://customer.example.com/customer/C1234`. This allows the Invoice service to directly interact with the customer resource for additional information (if required), without needing to construct its own resource identifier for the customer.
- Explicitly separate query parameters expected to be inserted by human users or service consumers into resource identifiers in the query component of the URL. For example, `http://invoice.example.com/search{?paid,due-date,min-amount,max-amount,customer}` can be interpreted as indicating that `paid`, `due-date`, `min-amount`, `max-amount`, and `customer` are all likely to be inserted into the resource identifier via human input or by a service consumer. The vocabulary used in the query component of the resource identifier is likely to come under increased governance scrutiny compared to other components of the resource.
- Variables that a service consumer needs to insert into URL templates can produce undesirable forms of coupling to be introduced between the consumer and the service contract. This is in direct opposition to the design goals of the Service Loose Coupling (413) principle. Each variable to be inserted needs to have

an agreed upon meaning among a service and its consumers. The simplest way to tackle this is to standardize the names, syntax, data types, and meaning of variables across multiple services as part of the uniform contract definition. It is straightforward to consider standardizing variable names such as `dtstart` and `dtend` to identify the start and end dates and times of a given query. For example, this type of vocabulary can be reused to query an invoice service as `http://invoice.example.com/query?dtstart=2015-03-06T10:00:00&dtend=2015-04-06T10:00:00`, a calendar of events, or a correspondence log for particular time periods.

NOTE

Business entities are prime targets for inclusion in a controlled resource identifier vocabulary. We have already seen examples in this chapter where a consumer queries the Invoice entity service for a list of invoices related to a particular customer. In this case, the expansion of this parameter would be the full resource identifier for that entity. As new service capabilities are defined, new vocabulary will be discovered. It will be important to keep the vocabulary up-to-date and to be able to identify which elements of the vocabulary are genuinely reused in practice across different service contracts, versus those that are service-specific.

Designing with and Standardizing REST Constraints

Although the set of REST constraints are, individually, separate and distinct design rules with corresponding design goals, there is room for interpretation concerning whether each constraint should be strictly applied. Due to the importance of standardizing how services are built as part of a service inventory, it is recommended that how REST constraints themselves are applied also be clearly standardized.

Stateless {395}

The two basic interpretations of rules established by the Stateless {395} constraint are:

- The looser interpretation is that session state is any data that a request message might refer to that does not have an explicit resource identifier. Under this definition, session state can be given a resource identifier within the service to transform it into service state. It can then be deferred by the service into a database or other dedicated repository. Further requests (by the same consumer or by other consumers) refer to the state by its resource identifier and so they can be understood independently of previous requests.

- A stricter interpretation is that session state is any data bound to a specific service consumer that would normally need to be destroyed when that consumer exits an on-going service activity, or when that consumer stops interacting with the service. Under this interpretation, associating a resource identifier with the data does not transform it into service state and it must still not be retained by the service between requests.

The usage of the Stateless {395} constraint requires a clear design standard, both in regards to the interpretation of the constraint as well as the extent to which it is applied to the service inventory.

Additionally, if any exceptions to or violations of Stateless {395} are allowed, these need to be well-defined so that there is an opportunity to adjust the service inventory's underlying infrastructure accordingly.

Cache {398}

As explained in Chapter 5, this constraint requires that any request whose response could potentially be reused for subsequent requests needs to incorporate the facility to include cache control metadata. This constraint mostly applies to data retrieval methods, such as GET and HEAD. However, it can also apply to some uses of POST and other forms of requests that can be classified as primarily retrieving data from a service.

Two basic forms of caching exist:

- A response message is considered reusable for a particular period of time. For example, a message containing report data can state that its content will remain valid for 24 hours. This allows the caching infrastructure to continue returning the same response without having to re-invoke the service for the duration of that period. The HTTP header used for this kind of caching is `Cache-Control` with a `maxage` field.
- A response message is considered reusable only if its validity is checked each time it is used. For example, a log of recent transactions may be reused until a new transaction is added. In this case, each time a cache handles a request it explicitly checks with the service to ensure that no further transactions have occurred before returning the cached response. The HTTP headers used for this kind of caching are `ETag` in responses and `If-None-Match` in requests.

In order to decide whether to even attempt to reuse a cached response the cache needs a mechanism for determining whether two requests are equivalent for caching purposes. Requests are often equivalent if their method and resource identifier are the same;

however, request headers can play a role in whether requests are equivalent or not. In support of this, it can be helpful to introduce a design standard regarding the usage of the HTTP `Vary` header that can be applied to identify which request headers were used as part of generating a response and, by a process of elimination, which headers were ignored. This feature allows requests that are slightly different to still reuse the same cached response.

In addition to a response being able to identify which request headers were used in generating the response, it is helpful to have a further design standard that establishes a canonical form for request messages so that they can be compared for equivalence. HTTP has a basic canonicalization mechanism that can be used to remove redundant whitespace and to merge duplicate headers.

Uniform Contract {400}

HTTP requires that methods and media types be "standard." In the context of REST this does not simply mean standardization, but instead refers to "reuse in practice" by multiple services. Methods, media types, headers, exception types, resource identifier syntax, and any other element of messages (other than the specific resource identifiers chosen as part of service contracts to expose service capabilities) are all required to be reused by multiple services in order to comply with Uniform Contract {400}. In some cases (as described earlier), even parts of the resource identifiers may be standardized as well.

Although the mere usage of a uniform contract introduces a natural level of service inventory standardization, there are aspects that need further attention and custom standardization.

Design standards need to be in place to address the following:

- New methods and media types added to the uniform contract need to be clearly identified and closely monitored as they progress toward a mature state. If actual reuse by multiple service contracts does not happen, it may be necessary to start treating these new extensions as being service-specific.
- Any methods or media types that are intended to be service-specific need to be governed as such to ensure that the quantity of logic that is directly exposed to these extensions is minimized in favor of coupling logic to more reusable methods and media types.
- Some service contracts may also not lend themselves to compliance with the inventory's overarching uniform contract. It can therefore be useful to have a design standard that determines under what circumstances exceptions may be permitted.

With regards to the last item on the preceding list, there should be strong governance in place to ensure that before allowing service-specific methods and media types, uniform methods and media types are always carefully and thoroughly considered first.

Layered System {404}

Layered System {404} requires that consumers and services not be able to tell whether they are communicating with each other directly, or via a series of intermediaries that understand the uniform contract. To comply with this constraint, new methods need to be analyzed to ensure intermediaries are able to pass requests and responses on towards their intended recipients, and to adequately hide the existence of intermediaries when they are present.

One key requirement of Layered System {404} is that enough information be present in each message for it to reach its intended recipient. This means we cannot, upon making a connection to the service, strip out the data that allowed that connection. For example, it is not valid to remove the service name embedded within a resource identifier after making a connection to the service. If the connection turned out to really only be to an intermediary then the intermediary would not be able to determine which service should receive the message. Instead, all requests should include their full resource identifier.

Another requirement is that consumers should not need to speak a different protocol, use different methods, or use different headers to communicate with an intermediary as compared to communicating with an actual service. If removing the intermediary stops the communication from working, the architecture is in breach of Layered System {404}.

SUMMARY OF KEY POINTS

- Defining the reuse of uniform contract methods and media types is a service inventory responsibility, as is enforcing the compliance of these uniform contract elements to REST constraints as part of design standards.
- Services based on different service models will tend to introduce different service contract design considerations and characteristics.
- The use of resource identifiers can be standardized for a given service inventory at both the syntax and vocabulary levels.

CASE STUDY EXAMPLE

By following proven REST service contract design techniques, together with custom design standards established specifically for the MUA enterprise, MUA architects use the service candidates modeled in Chapter 9 as input for a service-oriented design process.

The results of this effort are documented in the following sections.

Confer Student Award Service Contract (Task)

A student who submits an award conferral application will do so through a Web browser. A separate user interface is therefore designed to allow users to enter the application details. It is the submission of this browser-based form that initiates the task service.

Upon receiving the submission, a server-side script organizes the form data into an XML document based on the following media type:

```
application/vnd.edu.mua.student-award-conferral-application+xhtml+xml
```

Example 10.2 provides a submitted application form completed with sample data collected from the human user. This represents the data set that kick-starts and drives the execution of an entire instance of the Confer Student Award business process.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
  <head>
    <title>Student Award Conferral Application</title>
  </head>
  <body>
    <p>Student:
      <a rel="student"
        href="http://student.mua.edu/student/555333">
        John Smith (Student Number 555333)
      </a>
    </p>
    <p>Award:
      <a rel="award"
        href="http://award.mua.edu/award/BS/CompSci">
        Bachelor of Science with Computer Science Major
      </a>
    </p>
    <p>Event:
      <a rel="event"
        href="http://event.mua.edu/fall-graduation">
        fall graduation event
      </a>
    </p>
  </body>
</html>
```

Example 10.2

Sample application data, as submitted to the Web server. This document structure contains both human-readable and machine-processable information.

Figure 10.5 displays the Confer Student Award service contract. The preceding media type is deliberately designed to include human-readable and machine-readable data in a form suitable for long-term archival. The document is submitted to a service capability corresponding directly to the Start capability defined in the Confer Student Award service candidate (Figure 9.13).

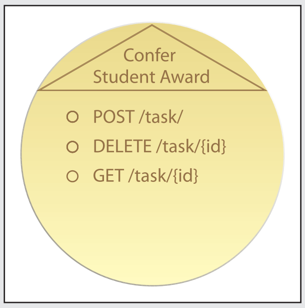


Figure 10.5

The Confer Student Award service contract.

As also shown in Figure 10.5, during the design process for this service contract it was decided to add new service capabilities to provide the following functions:

- `DELETE /task/{id}` – This capability was added to allow an executing instance of the Confer Student Award business process to be terminated.
- `GET /task/{id}` – This capability allows the state of an executing instance of the Confer Student Award business process to be queried.

Note that the sensitive nature of this kind of application means that the `GET /task/{id}` capability can be accessed only by authorized staff and by the student. The `DELETE /task/{id}` capability is only accessible by the student to cancel the application process.

Event Service Contract (Entity)

The Event entity service is equipped with a `GET /event/{id}` service capability used to query event information and which corresponds to the Get Details capability candidate from the Event service candidate (Figure 9.14).

During the service-oriented design process, architects decided to add further `GET /event/{id}/calendar` and `GET /event/{id}/description` capabilities (Figure 10.6) that allow for the retrieval of more specific event information. These capabilities were not added specifically in support of the Confer Student Award business process, but more so to provide a broader range of anticipated reusable functionality.

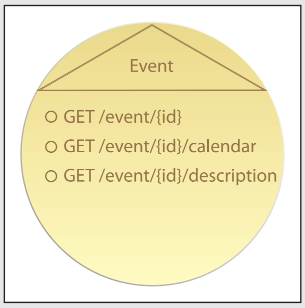


Figure 10.6
The Event service contract.

Award Service Contract (Entity)

In addition to implementing the three service capabilities from the original Award service candidate (Figure 9.15), SOA architects within MUA decide to make some further changes.

Back in Step 4 of the REST service modeling process (Chapter 9) MUA analysts determined that the following action was to be encompassed by the Confer Student Award task service logic:

- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules

However, with the rules being specific to each award type they determine that it should be the Award entity service that applies the bulk of these rules. Nevertheless, some generic checks do need to be applied so the logic is divided between the Confer Student Award task service and the Award entity service.

To avoid the task service from needing to pass full transcript details into the Award entity service for verification, it is decided to use a code-on-demand approach. The Award entity service provides the logic, but the logic is executed by the task service. The decision to define the logic centrally within the Award entity service is justified based on the need to produce human-readable output (for students), alongside machine-readable output (for the Confer Student Award service). As a result, the entity service provides a new `GET /award/{id}/conferral-rules` service capability (Figure 10.7) that supports the output of two formats for the rules logic: the first in human-readable form and the second in a form that can be readily embedded into the task service's logic.

MUA architects choose JavaScript for this purpose because they find that JavaScript runtimes are readily available for many of the technology platforms that have been used to develop services within the inventory. Choosing JavaScript over other technologies also accounts for it being the language of choice for the user-interface tier of the service inventory.

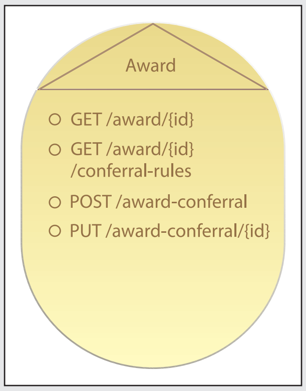


Figure 10.7
The Award service contract.

The same service capability is able to return the conferral rules in JavaScript or as human-readable HTML. The decision as to which transformation to carry out depends on which `Accept` header was provided by the service consumer. For example, the Confer Student Award task service requests the `application/javascript` media type, while service consumers requiring human-readable output will request the `text/html` media type.

Student Transcript Service Contract (Entity)

The Student service was originally intended as a centralized entity service that would encompass all student-related functionality and data access. However, iterations of the REST service modeling process that occurred subsequent to the examples covered in Chapter 9 resulted in a service inventory blueprint that revealed the Student service candidate as being far more coarse grained than any other. This was primarily due to the complexity of the Student entity and its relationships to other related entities.

Upon review of the Student service candidate it was determined to create a set of student-related entity services. One of these more specialized variations became the Student Transcript service candidate (Figure 10.8).

Because the Confer Student Award business process only requires access to student transcript information, it only needs to compose the Student Transcript service, not the actual Student service. As shown in Figure 10.9, the Student Transcript service contains service capabilities that correspond to the service capability candidates provided by the Student Transcript service candidate.

Figure 10.8
The Student Transcript service candidate that was defined subsequent to the Student service candidate from Chapter 9. This service effectively replaces the Student service in the Confer Student Award service composition.

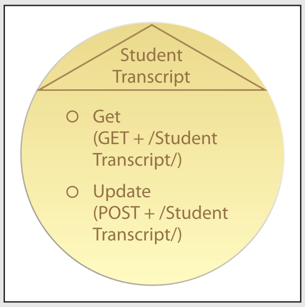
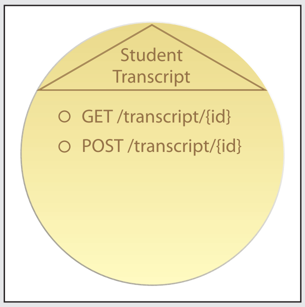

Student Transcript
Get (GET + /Student Transcript/)
Update (POST + /Student Transcript/)

Figure 10.9
The Student Transcript service contract.



Notification and Document Service Contracts (Utility)

The Notification service and Document service process similar human-readable data. Notifications sent via e-mail or hard copy can both be encoded as a human-readable document format, such as HTML or PDF.

The Notification service is retained for e-mail notifications while the Document service has been evolved into a printer-centric and postal-delivery-centric utility service. The Confer Student Award task service can send a document to the student in the preferred format by looking up the preferred delivery method in the original application form.

As shown in Figure 10.10, the Notification and Document services can each be invoked with the POST method.

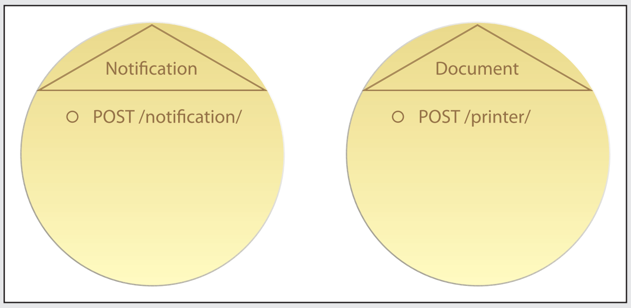


Figure 10.10
The Notification and Document service contracts.

The sample student (John Smith) from the application form used as input for the Confer Student Award task service has nominated his contact preference with a hyperlink to `mailto:s555333@student.mua.edu`. The service inventory standard for handling such an address is to transform the URL into `http://notification.mua.edu/sender?to=s555333@student.mua.edu` and use a POST method for its delivery. John Smith's notification will be delivered via e-mail to this address.

10.3 Complex Method Design

The uniform contract establishes a set of base methods used to perform basic data communication functions. As we've explained, this high-level of functional abstraction is what makes the uniform contract reusable to the extent that we can position it as the sole, over-arching data exchange mechanism for an entire inventory of services. Besides its inherent simplicity, this part of a service inventory architecture automatically results in the baseline standardization of service contract elements and message exchange.

The standardization of HTTP on the World Wide Web results in a protocol specification that describes the things that services and consumers "may," "should," or "must" do to be compliant with the protocol. The resulting level of standardization is intentionally only as high as it needs to be to ensure the basic functioning of the Web. It leaves a number of decisions as to how to respond to different conditions up to the logic within individual services and consumers. This "primitive" level of standardization is important to the Web where we can have numerous foreign service consumers interacting with third-party services at any given time.

A service inventory, however, often represents an environment that is private and controlled within an IT enterprise. This gives us the opportunity to customize this standardization beyond the use of common and primitive methods. This form of customization can be justified when we have requirements for increasing the levels of predictability and quality-of-service beyond what the World Wide Web can provide.

For example, let's say that we would like to introduce a design standard whereby all accounting-related documents (invoices, purchase orders, credit notes, etc.) must be retrieved with logic that, upon encountering a retrieval failure, automatically retries the retrieval a number of times. The logic would further require that subsequent retrieval attempts do not alter the state of the resource representing the business documents (regardless of whether a given attempt is successful).

With this type of design standard, we are essentially introducing a set of rules and requirements as to how the retrieval of a specific type of document needs to be carried out. These are rules and requirements that cannot be expressed or enforced via the base, primitive methods provided by HTTP. Instead, we can apply them in addition to the level of standardization enforced by HTTP by assembling them (together with other possible types of runtime functions) into aggregate interactions. This is the basis of the *complex method*.

A complex method encapsulates a pre-defined set of interactions between a service and a service consumer. These interactions can include the invocation of standard HTTP methods. To better distinguish these base methods from the complex methods that encapsulate them, we'll refer to base HTTP methods as *primitive methods* (a term only used when discussing complex method design.)

Complex methods are qualified as "complex" because they:

- can involve the composition of multiple primitive methods
- can involve the composition of a primitive method multiple times
- can introduce additional functionality beyond method invocation
- can require optional headers or properties to be supported by or included in messages

As previously stated, complex methods are generally customized for and standardized within a given service inventory. For a complex method to be standardized, it needs to be documented as part of the service inventory architecture specification. We can define a number of common complex methods as part of a uniform contract that then become available for implementation by all services within the service inventory.

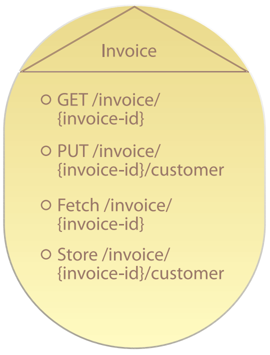
Complex methods have distinct names. The complex method examples that we cover shortly are called:

- Fetch – A series of GET requests that can recover from various exceptions.
- Store – A series of PUT or DELETE requests that can recover from various exceptions.
- Delta – A series of GET requests that keep a consumer in sync with changing resource state.
- Async – An initial modified request and subsequent interactions that support asynchronous request message processing.

Services that support a complex method communicate this by showing the method name as part of a separate service capability (Figure 10.11), alongside the primitive methods that the complex method is built upon. When project teams create consumer programs for certain services, they can determine the required consumer-side logic for a complex method by identifying what complex methods the service supports, as indicated by its published service contract.

Figure 10.11

An Invoice service contract displaying two service capabilities based on primitive methods and two service capabilities based on complex methods. We can assume that the two complex methods incorporate the use of the two primitive methods, but we can confirm this by studying the design specification that documents the complex methods.


Invoice
GET /invoice/{invoice-id}
PUT /invoice/{invoice-id}/customer
Fetch /invoice/{invoice-id}
Store /invoice/{invoice-id}/customer

NOTE

When applying the Service Abstraction (414) principle to REST service composition design, we may exclude entirely describing some of the primitive methods from the service contract. This can be the result of design standards that only allow the use of a complex method in certain situations. Going back to the previous example about the use of a complex method for retrieving accounting-related documents, we may have a design standard that prohibits these documents from being retrieved via the regular GET method (because the GET method does not enforce the additional reliability requirements).

It is important to note that the use of complex methods is by no means required. Outside of controlled environments in which complex methods can be safely defined, standardized, and applied in support of the Increased Intrinsic Interoperability goal, their use is uncommon and generally not recommended. When building a service inventory architecture we can opt to standardize on certain interactions through the use of complex methods or we can choose to limit REST service interaction to the use of primitive methods only. This decision will be based heavily on the distinct nature of the business requirements addressed and automated by the services in the service inventory.

Despite their name, complex methods are intended to add simplicity to service inventory architecture. For example, let's imagine we choose not to use pre-defined complex methods and then realize that there are common rules or policies that should have been applied to numerous services and their consumers. In this case, we will have built multiple services and consumers that behave unpredictably. When a service returns a redirection code, we can't be sure that all consumers will act upon it, and a temporary communication failure can have unexpected ramifications. Lack of policy can also result in unnecessarily redundant message processing logic. The fact that the implementations will continue to remain out of synch make this a convoluted architecture that is unnecessarily complex. This is exactly the problem that the use of complex methods is intended to avoid.

The upcoming sections introduce a set of sample complex methods organized into two sections:

- Stateless Complex Methods
- Stateful Complex Methods

Note that these methods are by no means industry standard. Their names and the type of message interactions and primitive method invocations they encompass have been customized to address common types of functionality.

NOTE

The *Case Study Example* section at the end of this chapter further explores this subject matter. In this example, in response to specific business requirements, two new complex methods (one stateless, the other stateful) are defined.

Stateless Complex Methods

This first collection of complex methods encapsulate message interactions that are compliant with the Stateless {395} constraint.

Fetch Method

Instead of relying only on a single invocation of the HTTP GET method (and its associated headers and behavior) to retrieve content, we can build a more sophisticated data retrieval method with features such as:

- automatic retry on timeout or connection failure
- required support for runtime content negotiation to ensure the service consumer receives data in a form it understands

- required redirection support to ensure that changes to the service contract can be gracefully accommodated by service consumers
- required cache control directive support by services to ensure minimum latency, minimum bandwidth usage, and minimum processing for redundant requests

We'll refer to this type of enhanced read-only complex method as a Fetch. Figure 10.12 shows an example of a pre-defined message interaction of a Fetch method designed to perform content negotiation and automatic retries.

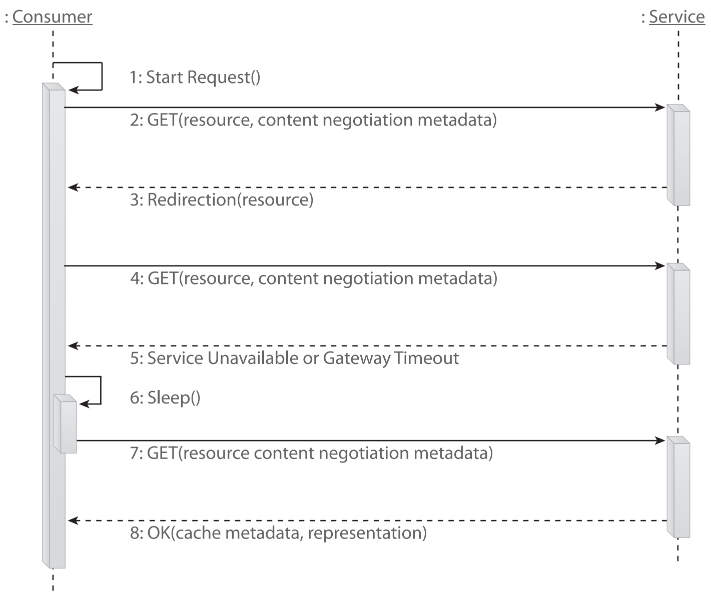

: Consumer
: Service
1: Start Request()
2: GET(resource, content negotiation metadata)
3: Redirection(resource)
4: GET(resource, content negotiation metadata)
5: Service Unavailable or Gateway Timeout
6: Sleep()
7: GET(resource content negotiation metadata)
8: OK(cache metadata, representation)

Figure 10.12
An example of a Fetch complex method comprised of consecutive GET method calls.

Store Method

When using the standard PUT or DELETE methods to add new resources, set the state of existing resources, or remove old resources, service consumers can suffer request timeouts or exception responses. Although the HTTP specification explains what each exception means, it does not impose restrictions as to how they should be handled. For this purpose, we can create a custom Store method to standardize necessary behavior.

The Store method can have a number of the same features as a Fetch, such as requiring automatic retry of requests, content negotiation support, and support for redirection exceptions. Using PUT and DELETE, it can also defeat low bandwidth connections by always sending the most recent state requested by the consumer, rather than needing to complete earlier requests first.

The same way that individual primitive HTTP methods can be idempotent, the Store method can be designed to behave idempotently. By building upon primitive idempotent methods, any repeated, successful request messages will have no further effect after the first request message is successfully executed.

For example, when setting an invoice state from "Unpaid" to "Paid":

- a "toggle" request would not be idempotent because repeating the request toggles the state back to "Unpaid."
- the "PUT" request is idempotent when setting the invoice to "Paid" because it has the same effect, no matter how many times the request is repeated

It is important to understand that the Store and its underlying PUT and DELETE requests are requests *to* service logic, not an action carried out on the service's underlying database. As shown in Figure 10.13, these types of requests are stated in an idempotent

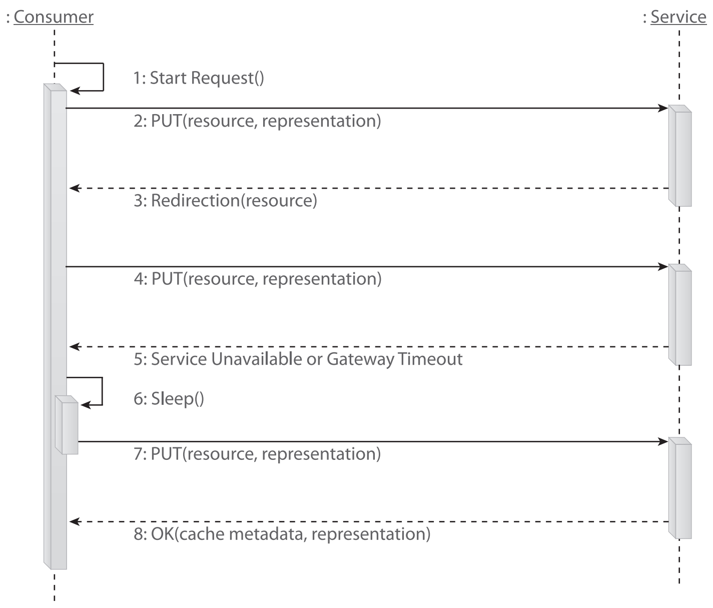

: Consumer
: Service
1: Start Request()
2: PUT(resource, representation)
3: Redirection(resource)
4: PUT(resource, representation)
5: Service Unavailable or Gateway Timeout
6: Sleep()
7: PUT(resource, representation)
8: OK(cache metadata, representation)

Figure 10.13
An example of the interaction carried out by a Store complex method.

manner in order to efficiently allow for the retrying of requests without the need for sequence numbers to add reliable messaging support.

NOTE

Service capabilities that incorporate this type of method are an example of the application of the Idempotent Capability [470] pattern.

Delta Method

It is often necessary for a service consumer to remain synchronized with the state of a changing resource. The Delta method is a synchronization mechanism that facilitates stateless synchronization of the state of a changing resource between the service that owns this state and consumers that need to stay in alignment with the state.

The Delta method follows processing logic based on the following three basic functions:

1. The service keeps a history of changes to a resource.
2. The consumer gets a URL referring to the location in the history that represents the last time the consumer queried the state of the resource.
3. The next time the consumer queries the resource state, the service (using the URL provided by the consumer) returns a list of changes that have occurred since the last time the consumer queried the resource state.

Figure 10.14 illustrates this using a series of GET invocations.

The service provides a "main" resource that responds to GET requests by returning the current state of the resource. Next to the main resource it provides a collection of "delta" resources that each return the list of changes from a nominated point in the history buffer.

The consumer of the Delta method activates periodically or when requested by the core consumer logic. If it has a delta resource identifier it sends its request to that location. If it does not have a delta resource identifier, it retrieves the main resource to become synchronized. In the corresponding response the consumer receives a link to the delta for the current point in the history buffer. This link will be found in the `Link` header (RFC 5988) with relation type `Delta`.

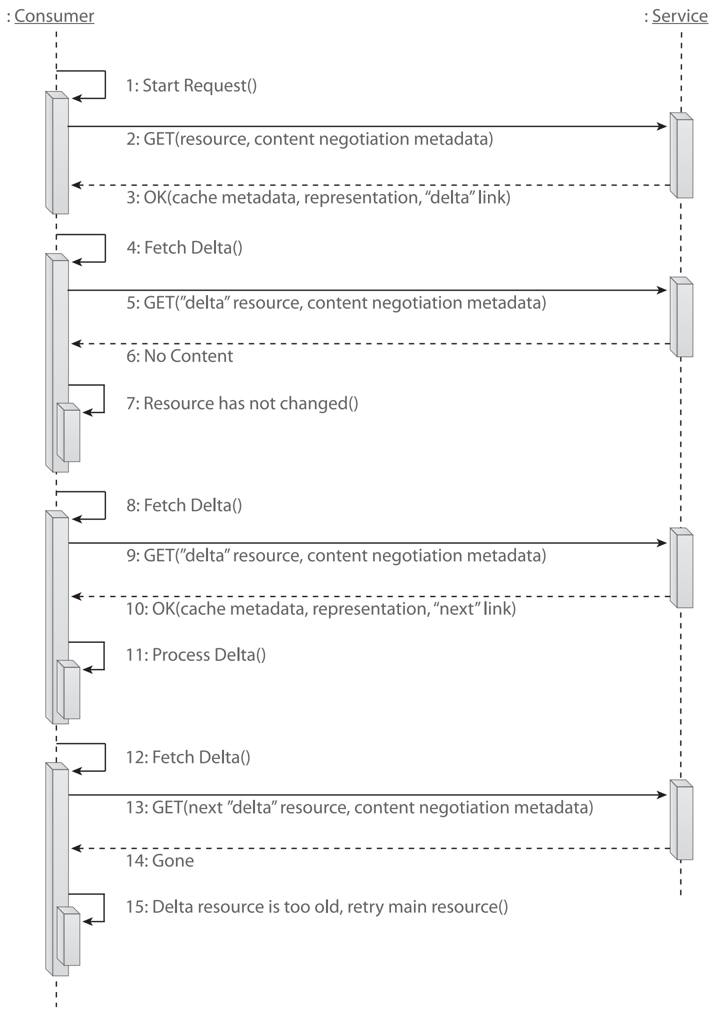

: Consumer
: Service
1: Start Request()
2: GET(resource, content negotiation metadata)
3: OK(cache metadata, representation, "delta" link)
4: Fetch Delta()
5: GET("delta" resource, content negotiation metadata)
6: No Content
7: Resource has not changed()
8: Fetch Delta()
9: GET("delta" resource, content negotiation metadata)
10: OK(cache metadata, representation, "next" link)
11: Process Delta()
12: Fetch Delta()
13: GET(next "delta" resource, content negotiation metadata)
14: Gone
15: Delta resource is too old, retry main resource()

Figure 10.14
An example of the message interaction encompassed by the Delta complex method.

The requested delta resource can be in any one of the following states:

1. It can represent a set of one or more changes that have occurred to the main resource since the point in history that the delta resource identifier refers to. In this case, all changes in the history from the nominated point are returned along with a link to the new delta for the current point in the history buffer. This link will be found in the `Link` header with relation type `Next`.

2. It may not have a set of changes because no changes have occurred since its nominated point in the history buffer, in which case it can return the `204 No Content` response code to indicate that the service consumer is already up-to-date and can continue using the delta resource for its next retrieval.
3. Changes may have occurred, but the delta is now expired because the nominated point in history is now so old that the service has elected not to preserve the changes. In this situation, the resource can return a `410 Gone` code to indicate that the consumer has lost synchronization and should re-retrieve the main resource.

Delta resources use the same caching strategy as the main resource.

The service controls how many historical deltas it is prepared to accumulate based on how much time it expects consumers will take (on average) to get up-to-date, or in some cases where a full audit trail is maintained for other purposes the number of deltas can be indefinite. The amount of space required to keep this record is constant and predictable regardless of the number of consumers, leaving it up to each individual service consumer to keep track of where it is in the history buffer.

Async Method

This complex method provides pre-defined interactions for the successful and canceled exchange of asynchronous messages. It is useful for when a given request requires more time to execute than what the standard HTTP request timeouts allow.

Normally, if a request takes too long, the consumer message processing logic will time out or an intermediary will return a `504 Gateway Timeout` response code to the service consumer. The Async method provides a fallback mechanism for handling requests and returning responses that does not require the service consumer to maintain its HTTP connection open for the total duration of the request interaction.

As shown in Figure 10.15, the service consumer issues a request, but does so specifying a call-back resource identifier. If the service chooses to use this identifier, it responds with the `202 Accepted` response code, and may optionally return a resource identifier in the `Location` header to help it track the place of the asynchronous request in its processing queue. When the request has been fully processed, its result is delivered by the service, which then issues a PUT or POST request to the call-back address of the service consumer.

If the service consumer issues a DELETE request (as shown in Figure 10.16) while the Async request is still in the processing queue (and before a response is returned), a

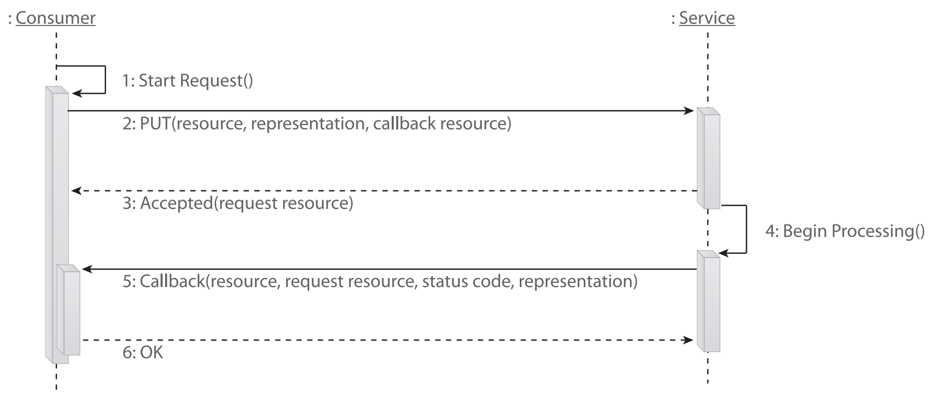

: Consumer
: Service
1: Start Request()
2: PUT(resource, representation, callback resource)
3: Accepted(request resource)
4: Begin Processing()
5: Callback(resource, request resource, status code, representation)
6: OK

Figure 10.15
An asynchronous request interaction encompassed by the Async complex method.

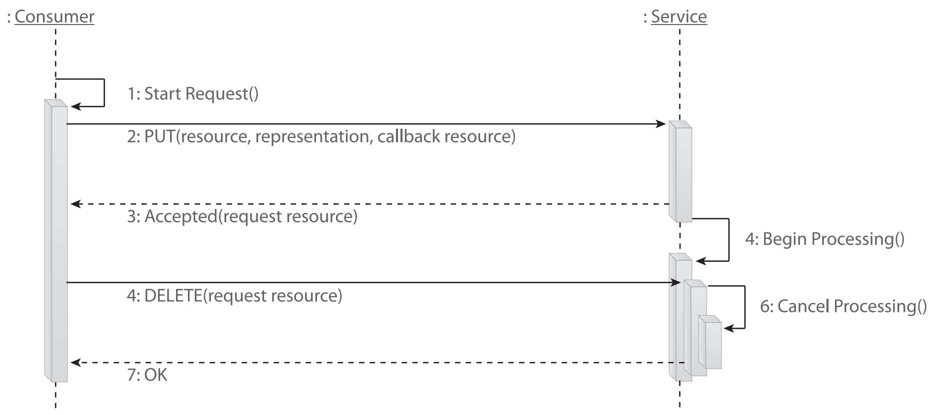

: Consumer
: Service
1: Start Request()
2: PUT(resource, representation, callback resource)
3: Accepted(request resource)
4: Begin Processing()
4: DELETE(request resource)
6: Cancel Processing()
7: OK

Figure 10.16
An asynchronous cancel interaction encompassed by the Async complex method.

separate pre-defined interaction is carried out to cancel the asynchronous request. In this case, no response is returned and the service cancels the processing of the request.

If the consumer cannot listen for call-back requests, it can use the asynchronous request identifier to periodically poll the service. Once the request has been successfully handled, it is possible to retrieve its result using the previously described Fetch method before deleting the asynchronous request state. Services that execute either interaction encompassed by this method must have a means of purging old asynchronous requests if service consumers are unavailable to pick up responses or otherwise “forget” to delete request resources.

Stateful Complex Methods

These next complex methods use REST as the basis of service design but incorporate interactions that intentionally breach the Stateless {395} constraint. Although the scenarios represented by these methods are relatively common in traditional enterprise application designs, this kind of communication is not considered native to the World Wide Web. The use of stateful complex methods can be warranted when we accept the reduction in scalability that comes with this design decision.

Trans Method

The Trans method essentially provides the interactions necessary to carry out a two-phase commit between one service consumer and one or more services (as per the application of the Atomic Transaction [432] pattern). Changes made within the transaction are guaranteed to either successfully propagate across all participating services, or all services are rolled back to their original states.

This type of complex method requires a "prepare" function for each participant before a final commit or rollback is carried out. Functionality of this sort is not natively supported by HTTP. Therefore, we need to introduce a custom PREP-PUT method (a variant of the PUT method), as shown in Figure 10.17.

In this example the PREP-PUT method is the equivalent of PUT, but it does not commit the PUT action. A different method name is used to ensure that if the service does not

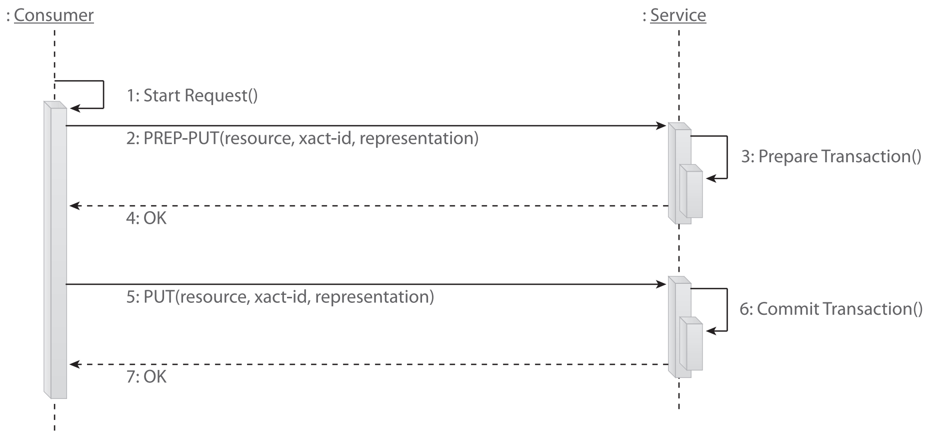


Figure 10.17
An example of a Trans complex method, using a custom primitive method called PREP-PUT.

understand how to participate in the Trans complex method, it then rejects the PREP-PUT method and allows the consumer to abort the transaction.

To carry out the logic behind a typical Trans complex method will usually require the involvement of a transaction controller to ensure that the commit and rollback functions are truly and reliably carried out with atomicity.

Alternative transaction models that have varying degrees of compliance with Stateless {395} are further explored in Chapter 12.

PubSub Method

A variety of publish-subscribe options are available once it is decided to intentionally breach the Stateless {395} constraint. As explained in the Event-Driven Messaging [465] pattern, these types of mechanisms are designed to support real-time interactions where a service consumer must act immediately when some pre-determined event at a given resource occurs. The Event-Driven Messaging [465] pattern is applied as an alternative to the repeated polling of the resource, which can negatively impact performance if the polling frequency is increased to detect changes with minimal delay.

There are various ways that this complex method can be designed. Figure 10.18 illustrates an approach that treats publish-subscribe messaging as a "cache-invalidation" mechanism.

This form of publish-subscribe interaction is considered "lightweight" because it does not require services to send out the actual changes to the subscribers. Instead, it informs them that a resource has changed by pushing out the resource identifier, and then reuses an existing, cacheable Fetch method as the service consumers pull the new representations of the changed resource.

The amount of state required to manage these subscriptions is bound to one fixed-sized record for each service consumer. If multiple invalidations queue up for a particular subscribed event, they can be folded together into a single notification. Regardless of whether the consumer receives one or multiple invalidation messages, it will still only need to invoke one Fetch method to bring itself up-to-date with the state of its resources each time it sees one or more new invalidation messages.

The PubSub method can be further adjusted to distribute subscription load and session state storage to different places around the network. This technique can be particularly effective within cloud-based environments that naturally provide multiple, distributed storage resources.

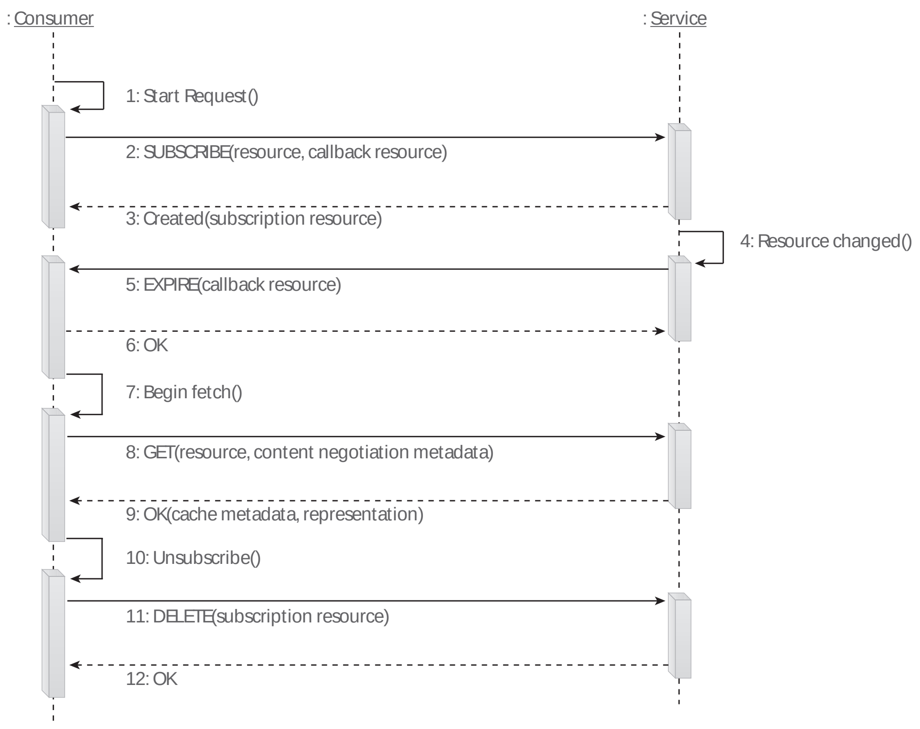

: Consumer
: Service
1: Start Request()
2: SUBSCRIBE(resource, callback resource)
3: Created(subscription resource)
4: Resource changed()
5: EXPIRE(callback resource)
6: OK
7: Begin fetch()
8: GET(resource, content negotiation metadata)
9: OK(cache metadata, representation)
10: Unsubscribe()
11: DELETE(subscription resource)
12: OK

Figure 10.18

An example of a PubSub complex method based on cache invalidation. When the service determines that something has changed on one or more resources, it issues cache expiry notifications to its subscribers. Each subscriber can then use a Fetch complex method (or something equivalent) to bring the subscriber up-to-date with respect to the changes.

SUMMARY OF KEY POINTS

- When designing both the uniform contract and individual service contracts, we can consider creating complex methods as part of the functions offered by the contracts.
- Complex methods encompass the aggregation of multiple primitive HTTP methods or the repeated execution of a single primitive HTTP method, along with other functional features that are part of predefined message interactions.
- Complex methods are ideally standardized so that the interaction behavior is consistent across all services and consumers that use them.
- Both stateless and stateful complex methods can be designed, although the latter variation is not REST-compliant.

CASE STUDY EXAMPLE

The MUA team responsible for service design encounters a number of requirements for accessing and updating resource state.

For example:

- One service consumer needs to atomically read the state of the resource, perform processing, and store the updated state back to the resource.
- Another service consumer needs to support concurrent user actions that modify the same resource. These actions update certain resource properties while others need to remain the same.

Allowing individual service consumers to contain different custom logic that performs these types of functions will inadvertently lead to problems and runtime exceptions when any two service consumers attempt updates to the same resource at the same time.

MUA architects conclude that the simplest way to avoid this is to introduce a new complex method that ensures that a resource is locked while being updated by a given consumer. Using the rules of optimistic locking, an approach commonly traditionally used with database updates, they are able to create a complex method that is stateless and takes advantage of existing standard features of the HTTP protocol. They name the method "OptLock" and write up an official description that is made part of the uniform contract profile:

OptLock Complex Method

If two separate service consumers attempt to update the state of a resource at the same time, their actions will clearly conflict with each other as the outcome depends on the order in which their requests reach the service. The OptLock method (Figure 10.19) addresses this problem by providing a means by which a service consumer can determine whether the state of a resource has changed since it was last read by the consumer before attempting an update.

Specifically, a consumer will first retrieve the current state associated with a resource identifier using the Fetch method. Along with the data the consumer receives an "ETag." ETag is a concept from HTTP that uniquely identifies the version of a resource

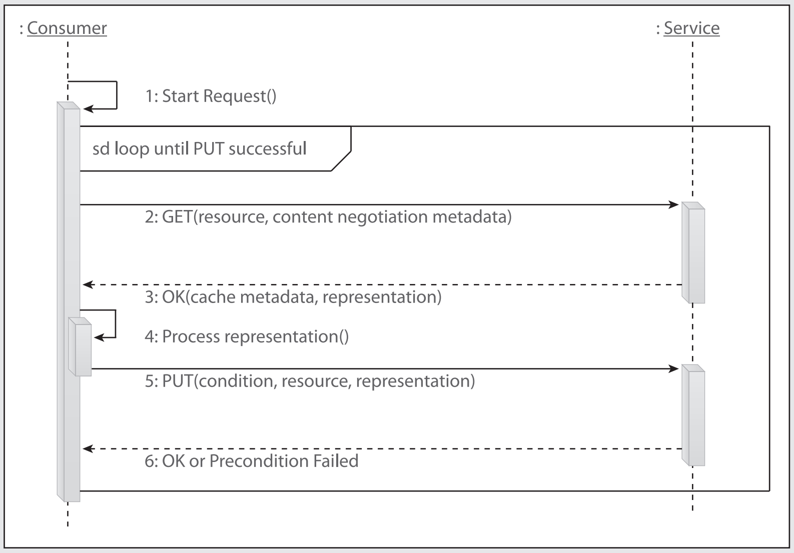

: Consumer
: Service
1: Start Request()
sd loop until PUT successful
2: GET(resource, content negotiation metadata)
3: OK(cache metadata, representation)
4: Process representation()
5: PUT(condition, resource, representation)
6: OK or Precondition Failed

Figure 10.19
An example of an OptLock complex method.

in an opaque way. Whenever the resource changes state its ETag is guaranteed to be different. When the service consumer initiates a Store, it does so conditionally by requesting the service to only honor the Store interaction if the resource's ETag still matches the one that it had when fetched. This is done with the `If-Match` header. The service can use the ETag value in the condition to detect whether the resource state has been changed in the meantime.

The OptLock complex method does not introduce any new features to HTTP, but instead introduces new requirements for handling GET and PUT requests. Specifically, the GET request must return an ETag value and the PUT request must process the `If-Match` header. And, if the resource has changed, the service must further guarantee not to carry out the PUT request.

There are several techniques for computing ETags. Some compute a hash value out of the state information associated with the resource, some simply keep a "last modified" timestamp for each resource, and others track the version of the resource state explicitly.

The OptLock method may not scale effectively for high concurrent access to a particular resource. If consumer update requests are denied with an HTTP `409 Conflict` response code, the OptLock method prescribes how the consumer can recover by fetching a newer version of the resource over which they have to re-compute the change and retry the Store method. However, this may fail again due to a conflicting update request. Service consumers that interact with a resource in this way rely on that particular resource having relatively low rates of write access.

The OptLock complex method becomes available as part of the uniform contract and is implemented by several services. However, scenarios emerge where a multiple consumers attempt to modify the resource at the same time, causing regular exceptions and failed updates. These situations occur during peak usage times and because concurrent usage volume is expected to increase further, it is determined that a more efficient means of serializing updates to the resource needs to be established.

It is proposed that the OptLock complex method be changed to perform pessimistic locking instead, as per the following PesLock complex method description:

PesLock Complex Method

Pessimistic locking provides greater flexibility and certainty than optimistic locking. From a REST perspective, this comes at the cost of introducing stateful interactions and limiting concurrent access while the pessimistic lock is held.

As shown in Figure 10.20, the WebDAV extensions to HTTP provide locking primitives that can be used within a composition architecture that intentionally breaches the Stateless {395} constraint. One consumer may lock out others from accessing a resource, so care must be taken that appropriate access control policies are in place. Consumers can also fail while the lock is held, which means that locks must be able to time out independently of the consumers that register them.

This way, the service consumer would be able to lock the resource for as long as it takes to read the state, modify it, and write it back again. Although other service consumers would still encounter exceptions while attempting to update the resource at the same time as the consumer that has locked it, it is deemed preferable to the unpredictability of managing the resource as part of an optimistic locking model.

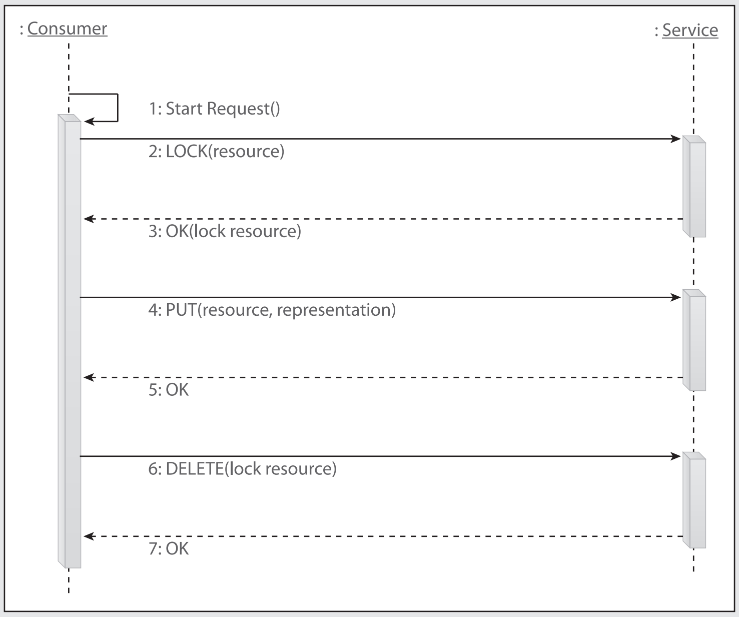


Figure 10.20
An example of a PesLock complex method.

This solution is not embraced by all of the MUA architects because retaining the lock on the resource requires that the Stateless {395} constraint be breached. It could further lead to the danger of stale locks starting to impact performance and scalability. In particular, unless proper measures are taken to ensure that only authorized consumers may lock a resource, this exposes the resources to denial of service attacks by malicious consumers that could lock out all other consumers.

After further discussion, a compromise is reached. The OptLock method will be attempted first. As a fallback, if the consumer tries three times and fails, it will attempt the stateful PesLock method to ensure it is able to complete the action.