Web Service Patterns

95-702 Distributed Systems

Web Service Patterns

Based largely on the book “Service Design Patterns” by Robert Daigneau

Ideas also taken from the Coulouris text on Distributed Systems and “Restful Java with Jax-RS” by Burke

Introduction

- Almost all enterprise applications need to be integrated with other applications.
- How is this done?
- Primarily with Shared files, Shared Database, RPC/RMI, Messaging, and sockets
- Some of these approaches would use separate components operating over a network and communicating only by passing messages. These would be considered distributed systems.

Why are distributed systems hard?

- Heterogeneous networks, operating systems, middleware, languages, developers - all required to work together
- The heterogeneity may hinder interoperability and performance
- Security becomes of greater concern. We need to consider the behaviors of Eve and Mallory.
- Failures can be partial and the concurrency of components adds complexity
- Increased latency
- Time differs on different systems
- Communicating parties will change
- Communicating parties may change location
- Moral? Don't distribute unless you must

But, we must distribute!

- Business capabilities are scattered across organizational boundaries and so are the systems that automate them
- Several binary schemes are widely used. These include CORBA, DCOM, Java RMI, .NET Remoting, and Protocol Buffers. For raw speed you are here.
- This discussion focuses on Web Services
- The web has been hugely successful and interoperable. The web is based on three core ideas: HTTP, XML (HTML), and URI's.

System Building Goal: Reduce Coupling

- Coupling is the degree to which one entity depends on another.
- Examples:
  - if two systems are coupled in time, they must both be ready to interact at a certain moment.
  - if the client knows the location of the service handling its request then the system is coupled in space.
  - if a client must provide an ordered list of typed parameters this is more tightly coupled than one that does not.
  - Web services can eliminate the client's coupling to the underlying technologies used by a service.
  - The web service client, however, is still dependent on the correct functioning of the service.
- Some coupling always exists.

System Building Goal: Separation of Concerns

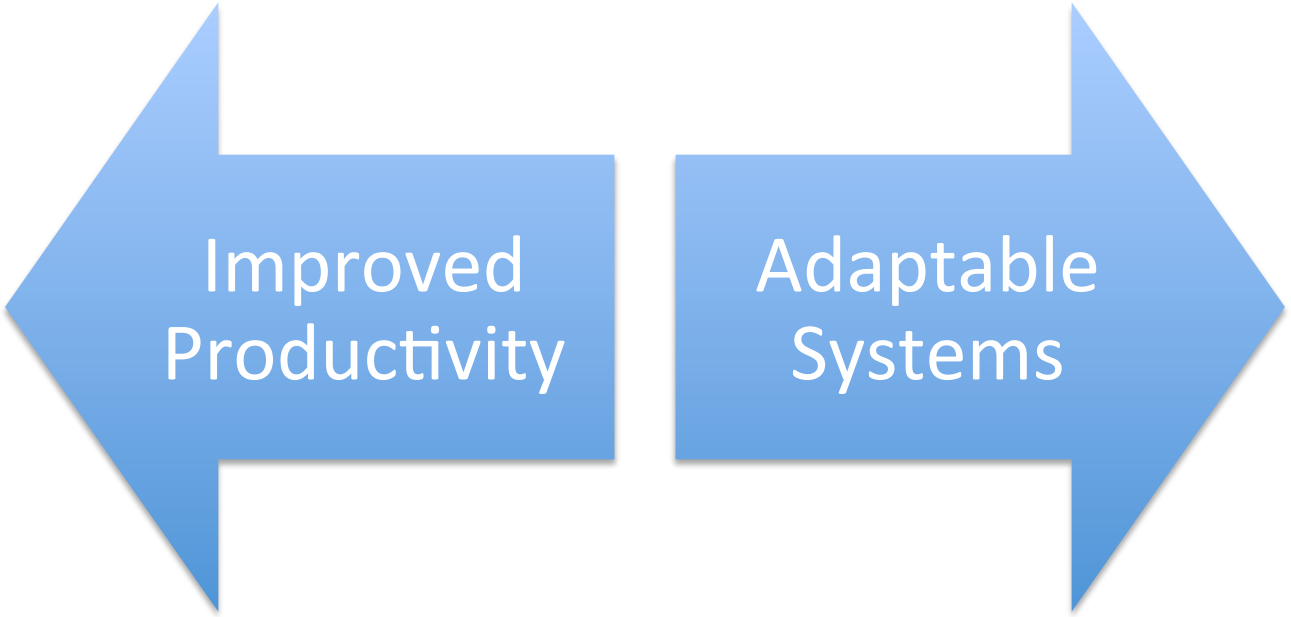
- Separate systems into distinct sections
- Separation of Concerns increases modularity
- Separation of concerns is promoted by encapsulation and Information hiding
- Application tiers separates concerns, e.g., a web site may be based on model view controller design
- Layered architectures separate concerns, e.g., the layers of HTTP/TCP/IP/Ethernet

Robustness Principle

- Jon Postel's Law:
- Be liberal in what you accept.

  Quiz: Are browsers liberal in what they accept?
- Be conservative in what you send.
- This conflicts with how we usually think of programming.
- Design services to read only what is needed from a request. Design clients to read only what is needed in a reply. Build tolerant readers.

System building issue: Often at odds



Categories of Patterns We Will Review

1) Web Service API Styles
2) Client-Server Interaction Styles
3) Request and Response Management
4) Web Service Implementation Styles

1) Web Service API Styles

- Recommendation: Pick one style
  - RPC API
  - Message API
  - Resource API

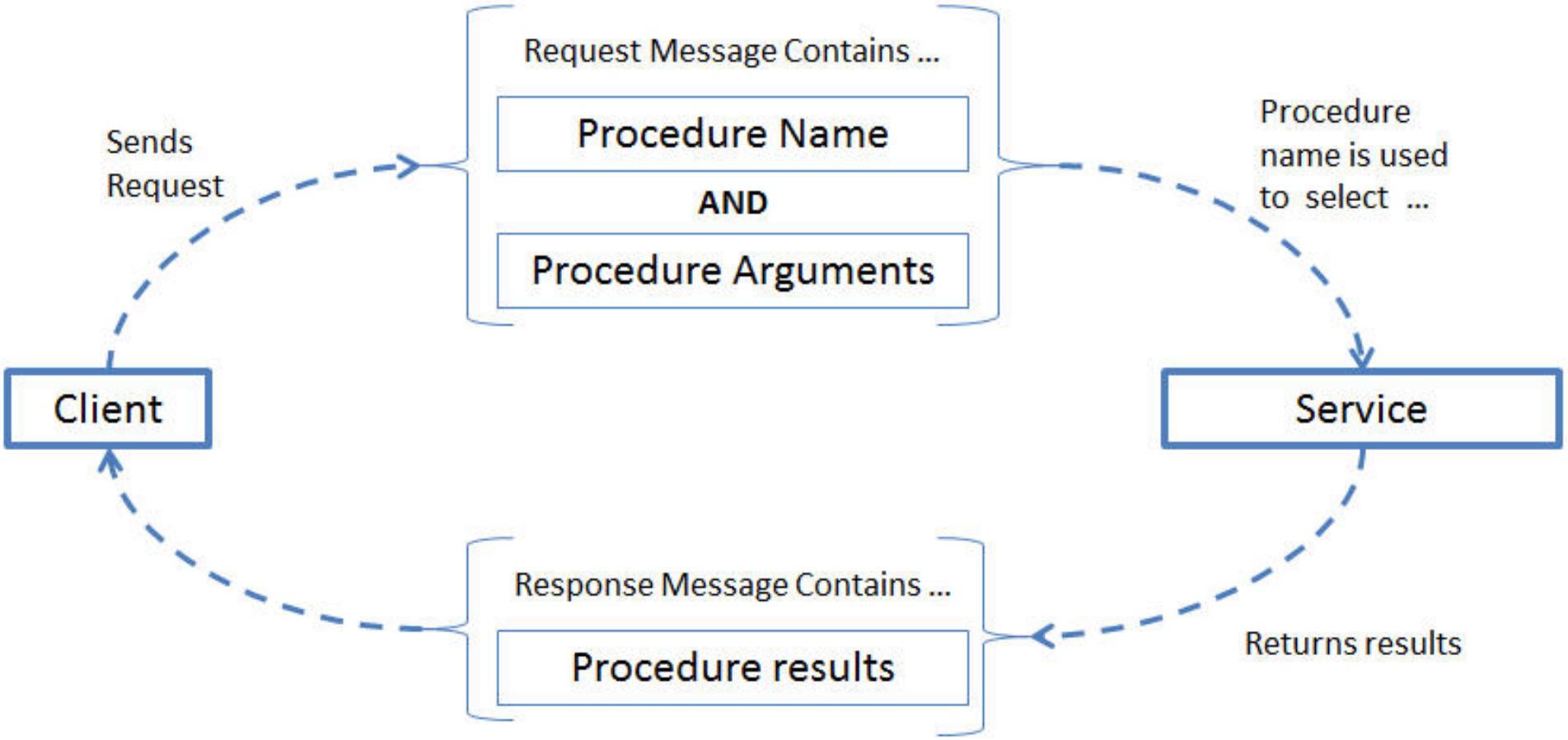
RPC API Style (1)

- How can clients execute remote procedures over HTTP?
- Define messages that identify the remote procedures to execute and also include a fixed set of elements that map directly into the parameters of remote procedures. Have the client send the message to a URI designated for the procedure.

RPC API Style (2)

- A service makes available a Service Descriptor.
- The Service Descriptor is used to generate a Service Connector (proxy) on the client.
- The client calls operations on the Service Connector as if it were the service.
- The descriptor might be coded with WSDL, XSDL or a non-XML approach (JSON-RPC)
- Frameworks such as JAX-WS and WCF makes all of this easy.

RPC API Style



From: http://www.servicedesignpatterns.com/WebServiceAPIStyles/RemoteProcedureCallAPI

RPC API Style Considerations (1)

- Methods or procedures may contain a list of typed parameters.
  - This is a tightly coupled system. If the parameter list changes this approach breaks clients.
  - A Descriptor change forces the Connector to be regenerated on the clients
  - A less tightly coupled system would contain a Single Message Argument

RPC API Style Considerations (2)

- Request/Response is the default but it may be replaced by Request/Acknowledge.
- Request/Acknowledge is less tightly coupled in time. (Separation of concerns) The request can be queued for later processing. This may improve scalability.
- The response may still be received with Request/Acknowledge/Poll or Request/Acknowledge/Callback.
- Clients may use an Asynchronous Response Handler if they don't want to block while waiting. (Think Javascript's XHR object)
- Request/Acknowledge/Callback and the Asynchronous Response Handler are quite different.

1) Web Service API Styles

- Recommendation: Pick one style
  - RPC API
  - Message API
  - Resource API

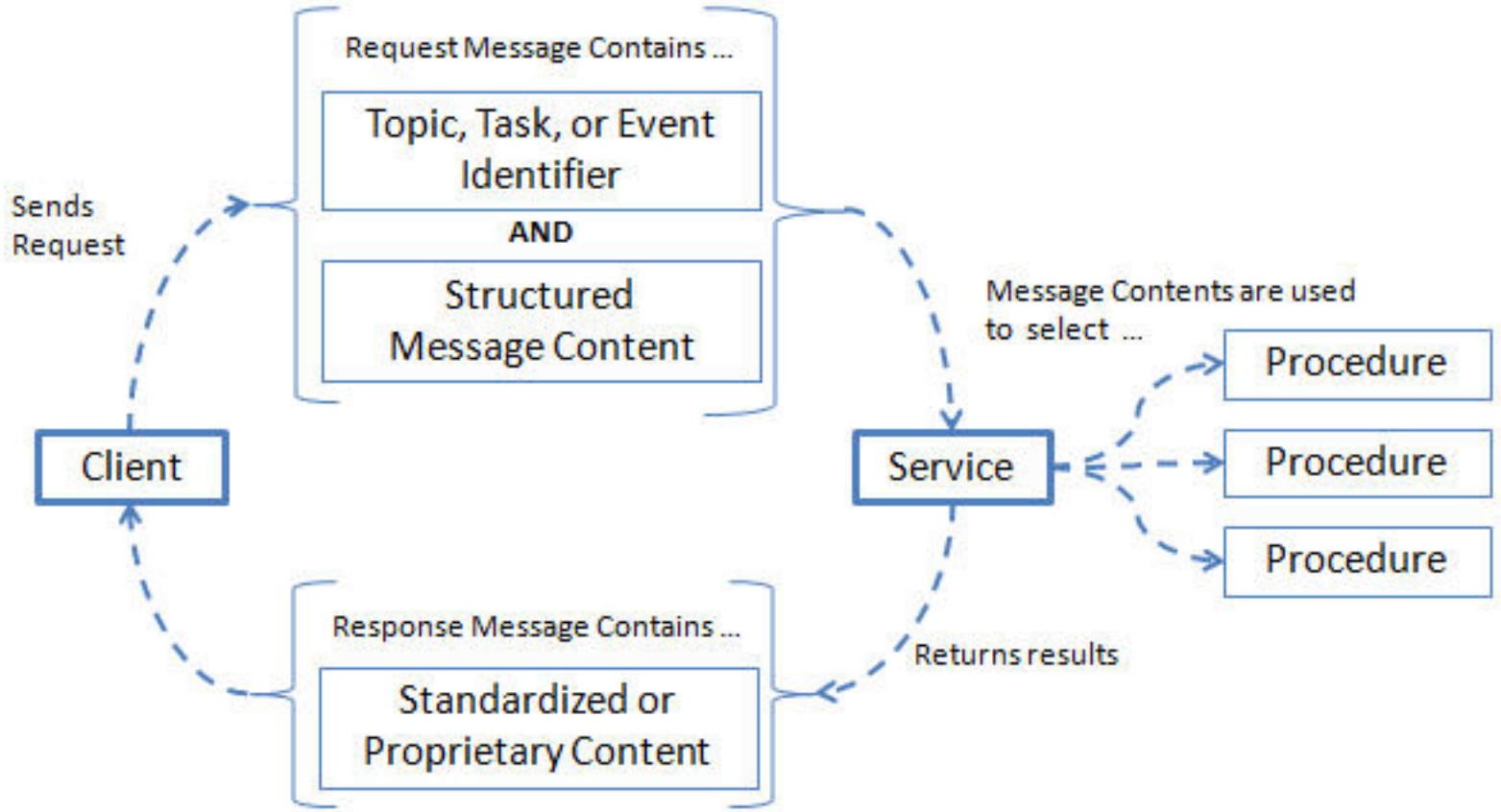
Message API's (1)

- How can clients send commands, notifications, or other information to remote systems over HTTP while avoiding direct coupling to remote procedures?
- Define messages that are not derived from signatures of remote procedures.
- When the message is received, the server examines its contents to determine the correct procedure to execute.
- The web service is used as a layer of indirection by insulating the client from the actual handler.

Message API's (2)

- The web service serves as a dispatcher and is usually built after the message is designed.
- No procedure name or parameter list is in the message.
- The service descriptor is often WSDL and XSDL.
- The services descriptor is used to generate the service connector (proxy).
- SOAP, WS-Policy, WS-Security may all be used.

Message or Document API Style



From http://www.servicedesignpatterns.com/WebServiceAPIStyles/MessageAPI

Message Style Considerations

- Message API's typically request/acknowledge rather than request/response.
- Responses may contain addresses of related services using the linked services pattern.
- It should be a simple matter to add additional message types to the service – simply dispatch new message to the correct handler.
- RPC API's may have 1 or more parameters.
- Message style API's contain exactly one.
- A standards body may define the message first.

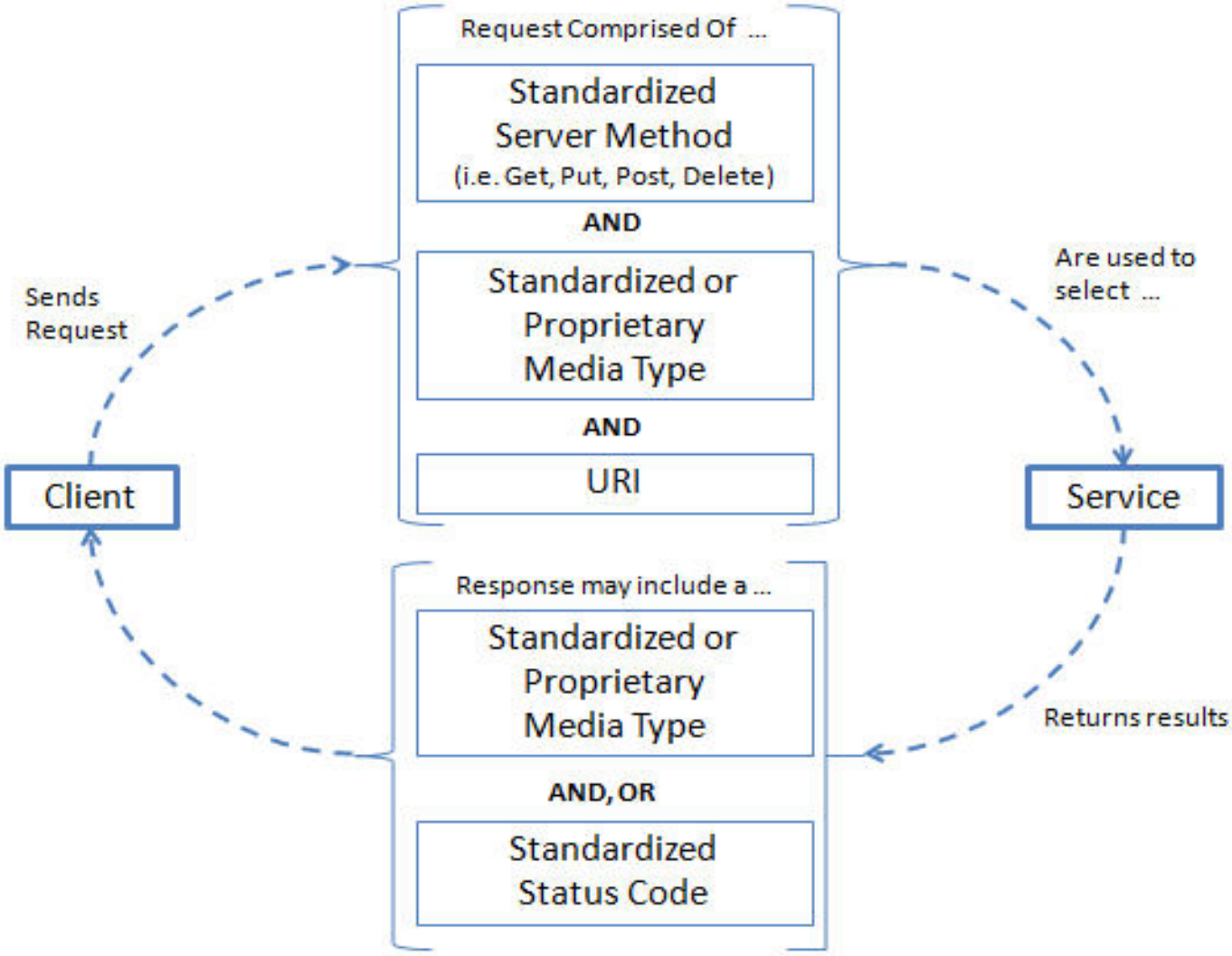
1) Web Service API Styles

- Recommendation: Pick one style
  - RPC API
  - Message API
  - Resource API

Resource API's

- A client application consumes or manipulates text, images, documents, or other media files managed by a remote system.
- How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimize the need for domain specific API's?
- Assign all procedures, instances of domain data, and files a URI.
- Leverage HTTP as a complete application protocol to define standard service behaviors.
- Exchange information by taking advantage of standardized media types and status codes when possible.
- The client's intent is determined by a) the HTTP method used b) the URI and c) the requested or submitted media type.
- These services often adhere to the principles of Representational State Transfer (REST).
- Not every Resource API would be considered RESTful.

Resource API's



From: http://www.servicedesignpatterns.com/WebServiceAPIStyles/ResourceAPI

Review of REST

Representational State Transfer

- Roy Fielding's doctoral dissertation (2000)
- Fielding (along with Tim Berners-Lee) designed HTTP and URI's.
- The question he tried to answer in his thesis was "Why is the web so viral"? What is its architecture? What are its principles?

REST Architectural Principles

- The web has addressable resources.

  Each resource has a URI.
- The web has a uniform and constrained interface.

  HTTP, for example, has a small number of methods. Use these to manipulate resources.
- The web is representation oriented – providing diverse formats.
- The web may be used to communicate statelessly – providing scalability
- Hypermedia is used as the engine of application state.

Understanding REST

- REST is not protocol specific.
- SOAP and WS-* use HTTP strictly as a transport protocol.
- HTTP may be used as a rich application protocol.
- Browsers usually use only a small part of HTTP (GET and POST).
- HTTP is a synchronous request/response network protocol used for distributed, collaborative, document based systems.
- Various message formats may be used – XML, JSON,..
- Binary data may be included in the message body.

Principle: Addressability

- Addressability (not restricted to HTTP)
  Each HTTP request uses a URI.
  The format of a URI is well defined:

  scheme://host:port/path?queryString#fragment

  The scheme need not be HTTP. May be FTP or HTTPS.
  The host field may be a DNS name or a IP address.
  The port may be derived from the scheme. Using HTTP implies port 80.
  The path is a set of text segments delimited by the "/".
  The queryString is a list of parameters represented as
  name=value pairs. Each pair is delimited by an "&".
  The fragment is used to point to a particular place in a document.

  A space is represented with the '+' characters. Other characters use %
  followed by two hex digits.

Principle: Uniform Interface (1)

- A uniform constrained interface

  - No action parameter in the URI

  - HTTP GET
    - read only operation
    - idempotent (once same as many)
    - safe (no important change to server's state)
    - may include parameters in the URI
      http://www.example.com/products?pid=123

Principle: Uniform Interface (2)

HTTP PUT

- store the message body
- insert or update
- idempotent
- not safe

Principle: Uniform Interface (3)

HTTP POST

- Not idempotent
- Not safe
- Each method call may modify the resource in a unique way
- The request may or may not contain additional information
- The response may or may not contain additional information
- The parameters are found within the request body (not within the URI)

Principle: Uniform Interface (4)

HTTP DELETE

- remove the resource
- idempotent
- Not safe
- Each method call may modify the resource in a unique way
- The request may or may not contain additional information
- The response may or may not contain additional information

HTTP HEAD, OPTIONS, TRACE and CONNECT are less important.

Principle: Uniform Interface (5)

Does HTTP provide too few operations?

Note that SQL has only four operations: SELECT, INSERT, UPDATE and DELETE

JMS and MOM have, essentially, two operations: SEND and RECEIVE

A lot gets done with SQL and JMS

What does a uniform interface buy?

Familiarity

We do not need a general IDL that describes a variety of method signatures.

We already know the methods and their semantics.

Interoperability

WS-* has been a moving target

HTTP is widely supported

Scalability

Since GET is idempotent and safe, results may be cached by clients or proxy servers.

Since PUT and DELETE are both idempotent neither the client or the server need worry about handling duplicate message delivery

Principle: Representation Oriented(1)

- Representations of resources are exchanged.
- GET returns a representation.
- PUT and POST passes representations to the server so that underlying resources may change.
- Representations may be in many formats: XML, JSON, YAML, etc., …

Principle: Representation Oriented(2)

- HTTP uses the CONTENT-TYPE header to specify the message format the server is sending.
- The value of the CONTENT-TYPE is a MIME typed string. Versioning information may be included.
- Examples:

  text/plain

  text/html

  application/vnd+xml;version=1.1
- "vnd" implies a vendor specific MIME type

Principle: Representation Oriented(3)

- The ACCEPT header in content negotiation.
- An AJAX request might include a request for JSON.
- A Java request might include a request for XML.
- Ruby might ask for YAML

Principle: Communicate Statelessly

- The application may have state but there is no client session data stored on the server.
- If there is any session-specific data it should be held and maintained by the client and transferred to the server with each request as needed.
- The server is easier to scale. No replication of session data concerns.

Principle: HATEOAS (1)

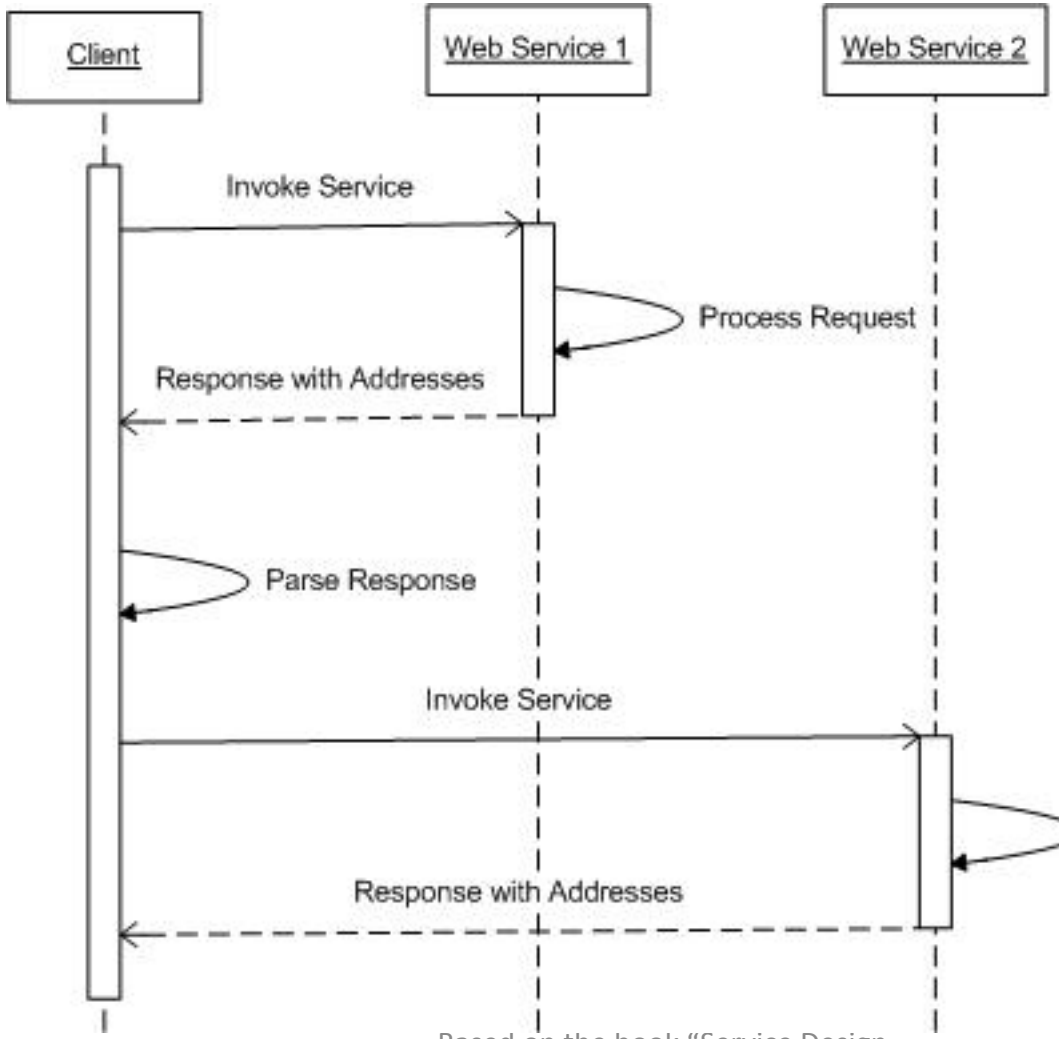
- Hypermedia is document centric but with the additional feature of links.
- With each request returned from a server it tells you what interactions you can do next as well as where you can go to transition the state of your application.
- Example:

```
<order id = “111”>
   <customer>http://…/customers/3214
   <order-entries>
      <order-entry>
         <qty>5
         <product>http://…/products/111
```

Principle: HATEOAS (2)

- For another example, after calling on an order creation service, the service would return URI's associated with Order Update, Order Cancel and Order Status.
- Consider the game of "Knock Knock". How could the protocol be controlled via HATEOS?
- HATEOS exemplifies "Late Binding". The methods an application may invoke are not known until runtime.
- HATEOS is a Linked Services Pattern

Linked Services Pattern



Only publish the addresses of a few root web services. Include the addresses of related services in each response. Let clients parse responses to discover subsequent service URIs.

Resource Style Considerations (1)

- Very appropriate for diverse clients - Browsers, feed readers, syndication services, web aggregators, microblogs, mashups, AJAX, and mobile applications
- Some may consider direct resource addressability a security risk – "hackable URI's".
- Authentication and authorization logic is needed here.
- Resource descriptors (and hence code generation tools) not used.
- Use Request/Response or Request/Acknowledge (HTTP 202 is an acknowledgement)
- Clients may use Asynchronous Response Handler to avoid blocking
- One logical resource may be represented at one URI. Clients choose the type of representation with Media Type Negotiation.

Categories of Patterns We Will Review

1) Web Service API Styles
2) Client-Server Interaction Styles
3) Request and Response Management
4) Web Service Implementation Styles

Regardless of which WS Style is chosen (RPC, Messaging, or Representational)

- We must decide on the client service interaction style.
- Request/Response….simplest
- Request/Acknowledge…not coupled in time, easier to scale
- Request/Acknowledge/Polling… simple, use ID to get response, increase use of network
- Request/Acknowledge/CallBack… harder, provide a service to handle the response
- Request/Acknowledge/Relay…notify others of request processing, foundation of publish/subscribe
- Media Type Negotiation…provide multiple representations of one logical resource while minimizing the number of distinct URI's. (Use HTTP Accept Headers not a new URL) Negotiation may be "server driven" or "client driven"
- Linked Services… less coupling in space, client insulated from changing locations, replaces need for registries or brokers (used in APP)

Categories of Patterns We Will Review

1) Web Service API Styles
2) Client-Server Interaction Styles
3) Request and Response Management
4) Web Service Implementation Styles
5) Web Service Infrastructures
6) Web Service Evolution

Request/Response Management

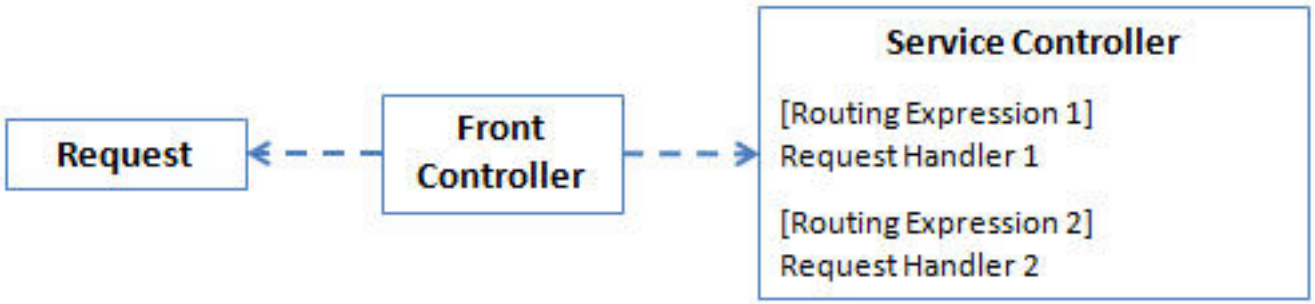
Decouple requests and responses from underlying system.
Useful patterns for all three web service API styles.

- Service Controller
- Data Transfer Object
- Request Mapper
- Response Mapper

Service Controller

- How can the correct web service be executed without having to maintain complex parsing and routing logic?
- Create a class (a Service Controller) that identifies a set of related services. Annotate each class method with routing expressions that can be interpreted by a Front Controller.
- The Front Controller selects the method in the Service Controller based on annotations on Service Controller methods (web methods) or in configuration files.
- The types of Routing Expressions used in the Service Controller depends on the service API style.

Service Controller



Request/Response Management

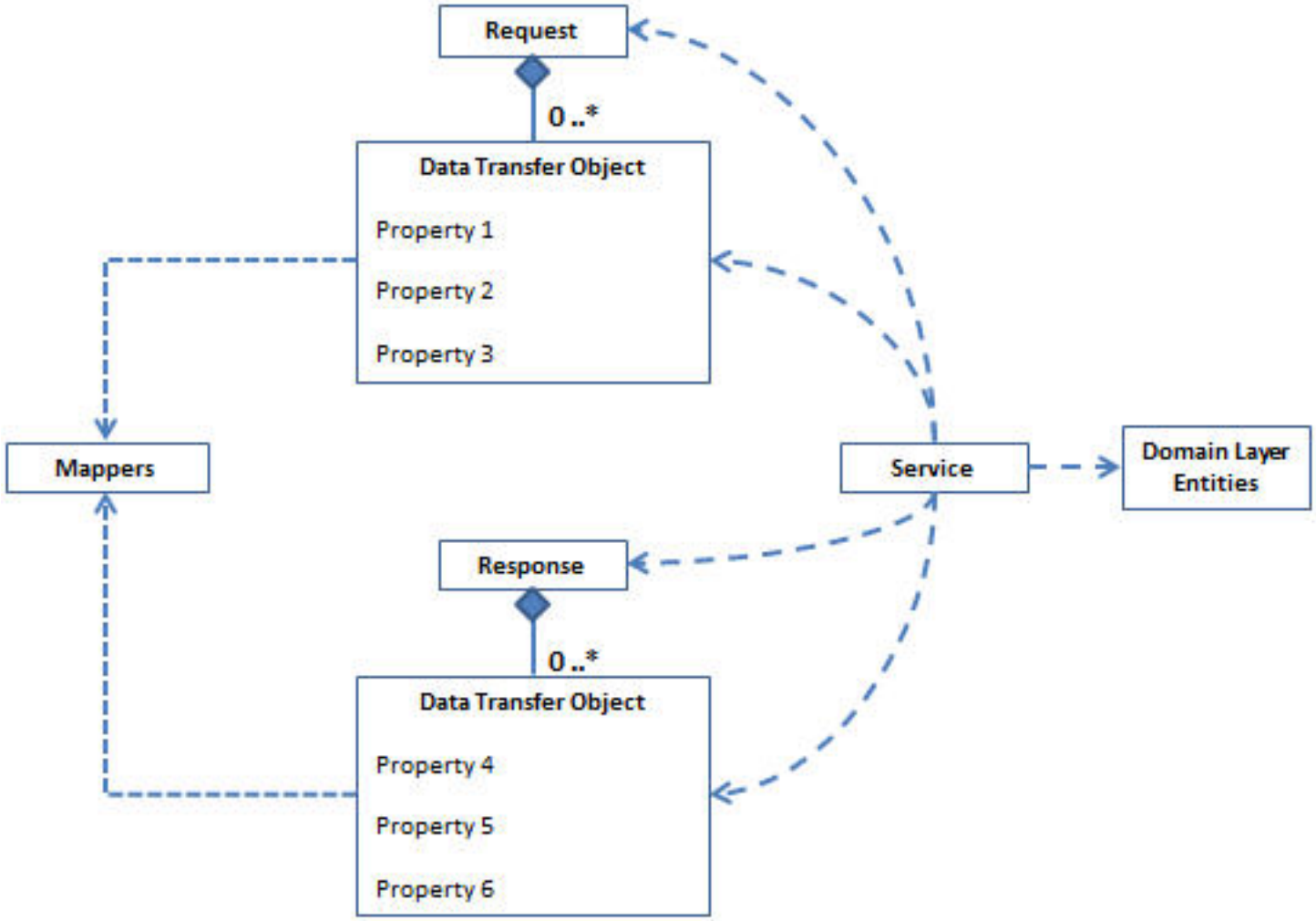
Decouple requests and responses from underlying system.
Useful patterns for all three web service API styles.

- Service Controller
- Data Transfer Object
- Request Mapper
- Response Mapper

Data Transfer Object

- Web Services typically use XML or JSON
- How can one simplify manipulation of request and response data, enable domain layer entities, requests, and responses to vary independently, and insulate services from wire-level message formats?
- DTO's are created as separate entities whose sole purpose is to define how data is received and returned from a service.

Data Transfer Objects



Request/Response Management

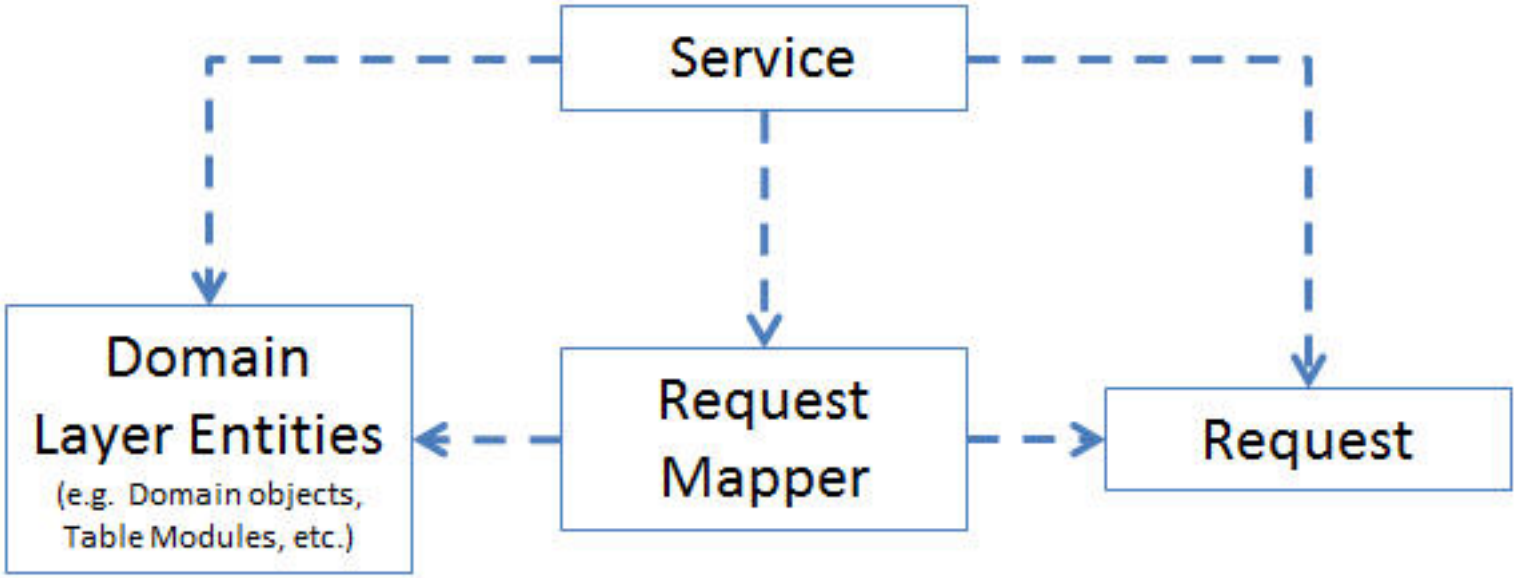
Decouple requests and responses from underlying system.
Useful patterns for all three web service API styles.

- Service Controller
- Data Transfer Object
- Request Mapper
- Response Mapper

Request Mapper

- A web service receives XML. The service owner has little to no control over the design of request structures.
- How can a service process data from requests that are structurally different yet semantically equivalent?
- Create specialized classes that leverage structure-specific APIs to target and move select portions of requests directly to domain layer entities or to a common set of intermediate objects that can be used as input arguments to such entities. Load a particular mapper based on key content found in the request.

Request Mapper



Request/Response Management

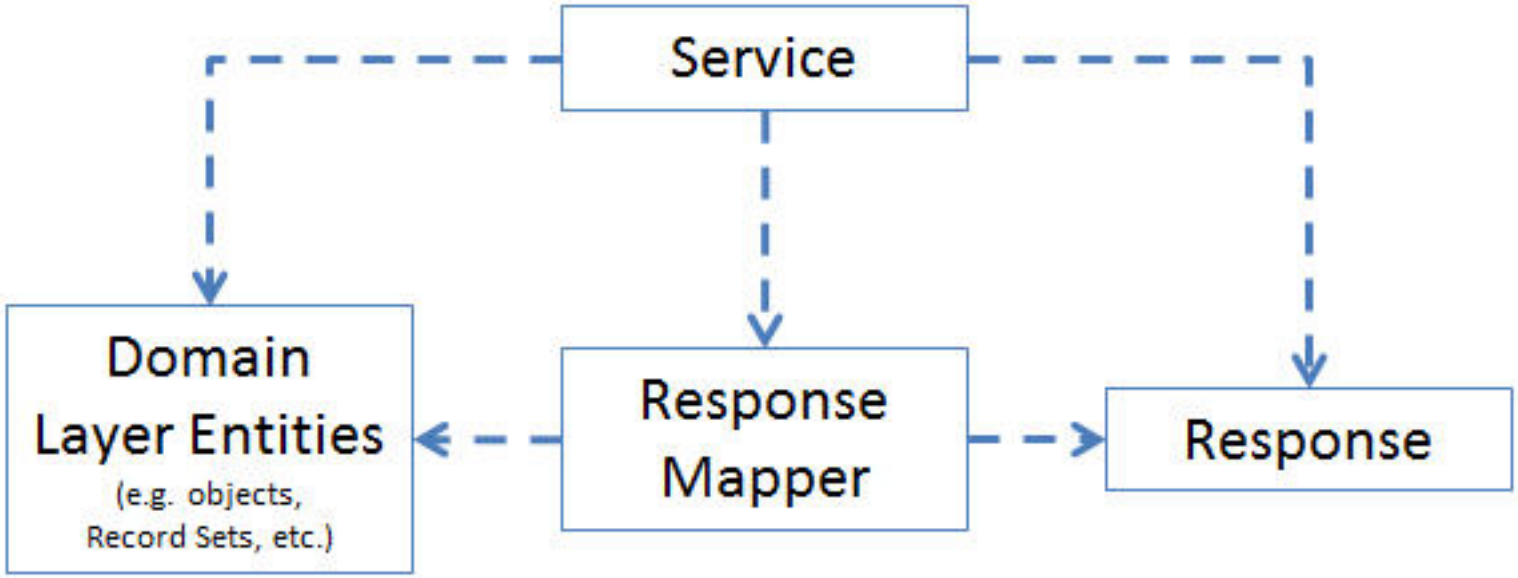
Decouple requests and responses from underlying system.
Useful patterns for all three web service API styles.

- Service Controller
- Data Transfer Object
- Request Mapper
- Response Mapper

Response Mapper

- A web service returns a text based response
- How can the logic required to construct a response be reused by multiple services?
- Once a mapper has been instantiated, the service calls methods on the mapper to pass domain objects, record sets, structs or primitive data. At some point, the service calls the mapper to acquire a final response that is returned to the client.

Response Mapper



Categories of Patterns We Will Review

1) Web Service API Styles
2) Client-Server Interaction Styles
3) Request and Response Management
4) Web Service Implementation Styles

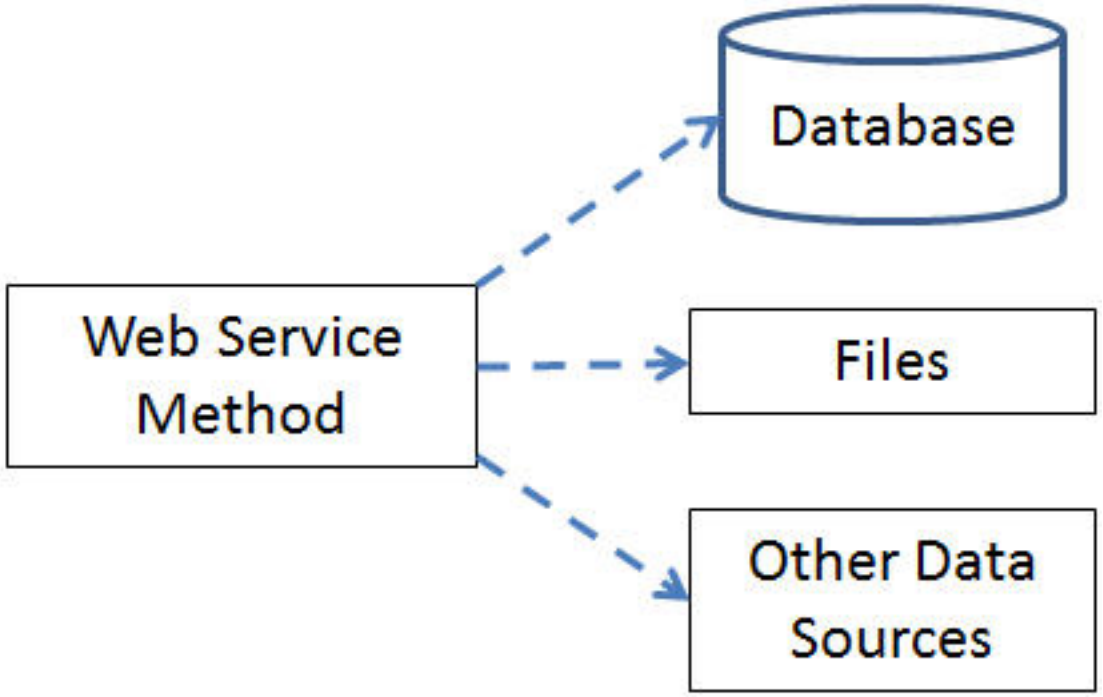
(4) Web Service Implementation Styles

- Transaction Script
- Datasource Adapter
- Operation Script
- Command Invoker
- Workflow Connector

Transaction Script

- How can developers quickly implement web service logic?
- Write custom logic for database access, file manipulation, or other purposes directly within the web service method.

Transaction Script



Transaction Script Considerations

- Simplicity
- Good if your in a hurry
- Tightly coupled to the underlying resources
- When the underlying resource changes, so too may the service have to change.
- Hard to maintain overtime.

(4) Web Service Implementation Styles

- Transaction Script
- Datasource Adapter
- Operation Script
- Command Invoker
- Workflow Connector

Datasource Adapter

- Clients would like to use internal system resources, but access to these entities must be controlled.
- How can a web service provide access to internal resources like database tables, stored procedures, domain objects, or files with a minimum amount of custom code?
- Service frameworks that support this pattern intercept and translate requests into one or more actions against a specialized *Datasource Provider* that encapsulates the logic required to interact with a specific datasource type (e.g. Object-Relational Mapper, database, file, etc.).
- Vendor SOA Suites provide many adapters
- A WSDL document may be generated or the Datasource may return the URL of an APP Service Document

Datasource Adapter Considerations

- Simple to use.
- Less code needs written.
- Some tools impose an API style. Others allow you to choose (RPC, Message, or Resource)
- Tight coupling to backend data sources. Change the data source and the client may need to be regenerated.

(4) Web Service Implementation Styles

- Transaction Script
- Datasource Adapter
- Operation Script
- Command Invoker
- Workflow Connector

Operation Script

- How can web services reuse common domain logic without duplicating code?
- Common domain logic must be made available to a number of web services that may have different API styles or client/service interaction styles
- Encapsulate common business logic in domain layer entities that exist outside of the web service. Limit the logic within web services to algorithms that direct the activities of these entities.
- Often function as the top-most transaction manager for the entities that are used to fulfill the client's request.

Operation Script Considerations

- Usually manage local transactions
- Distributed transactions add complexity
- May use an Inversion of Control container that instantiates required objects by consulting configuration files.
- Business logic duplication is reduced.
- Still, duplication of validation, control flow, and exception handling code may continue.
- Use Command Invoker for less duplication

(4) Web Service Implementation Styles

- Transaction Script
- Datasource Adapter
- Operation Script
- Command Invoker
- Workflow Connector

Command Invoker

- How can web services with different APIs reuse common domain logic while enabling both synchronous and asynchronous request processing?
- All domain logic is extracted from the web service and moved to Command Objects.
- The code that is left over in the web service does very little. In its simplest form, the service selects, instantiates, and populates a command object with request data, then calls a method on the command to initiate request processing.
- The service might also instantiate a command and forward it to a background request processor for deferred processing.

Command Invoker Considerations

- The service may invoke immediately or may forward to a background process using request/acknowledge
- Request mappers may be used to translate the request data
- Commands may be implemented as transaction scripts or as operation scripts

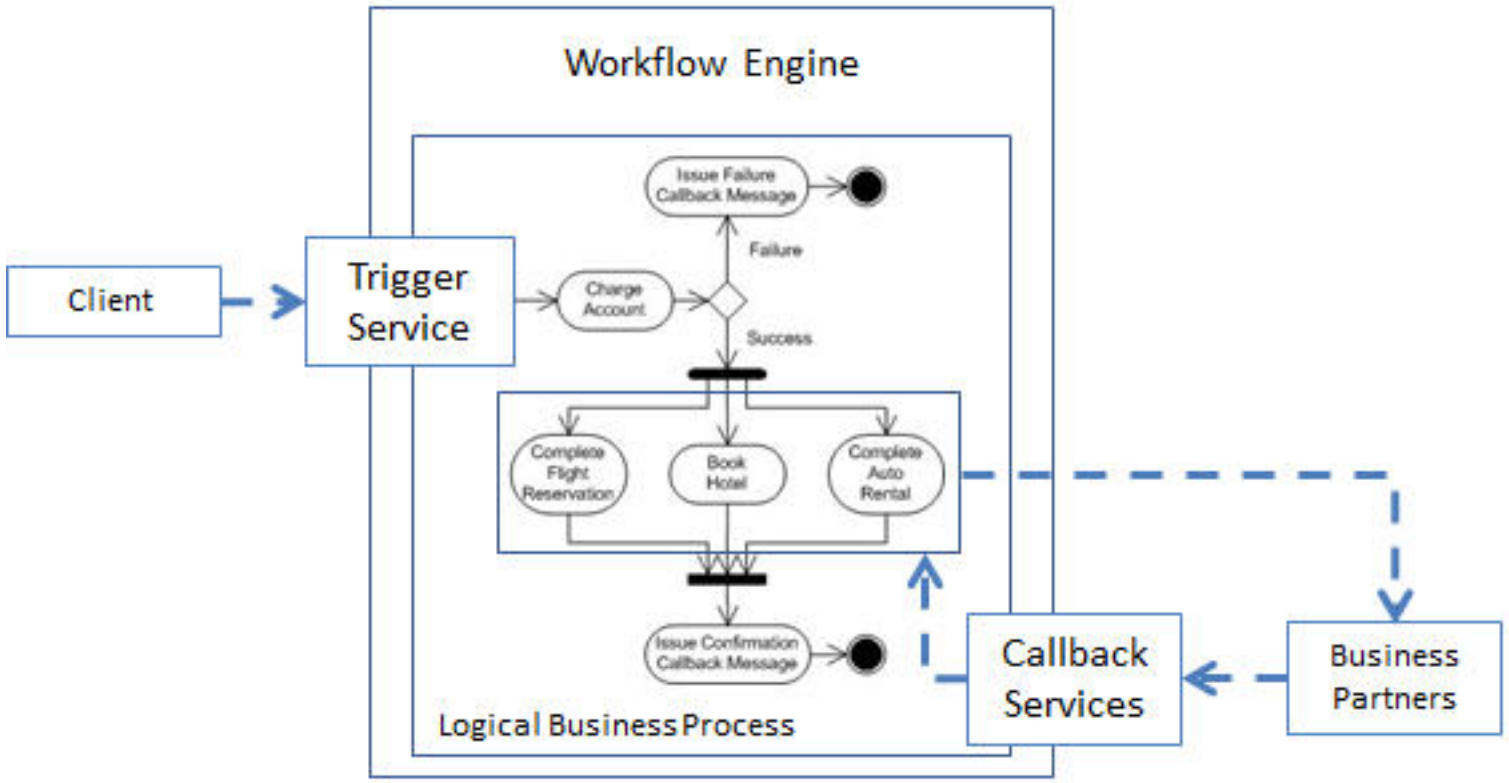
(4) Web Service Implementation Styles

- Transaction Script
- Datasource Adapter
- Operation Script
- Command Invoker
- Workflow Connector

Workflow Connector

- How can web services be used to support complex and long running business processes?
- These processes may run for minutes or hours or days.
- Use a workflow engine to manage the life cycle and execution of tasks within complex or long-running business processes. Identify a web service that will trigger each logical business process. Use callback services to receive additional data for these long-running processes, and forward messages from these callback services to the workflow engine.

Workflow Connector



Workflow Connector Considerations

- Rather than distributed transactions, use compensation handling.
- BPEL may be used to orchestrate business process.
- Overkill for many short running processes
- Some are lightweight while are others are heavy duty and expensive
- Business Activity Monitoring is often provided