Practical SOA for the Solution Architect

Ganesh Prasad
Consultant Architect, WSO2

Australia, October 2011
About the Presenter

- Consultant Architect with WSO2
- 24+ years in IT
- 10+ years of SOA-related experience
- Extensive experience working at SOA end-user organisations
- On design review committees for many solution designs
- Ideas for lightweight method came out of practical experience
Target Audience

- **The Solution Architect**
  - Understands business requirements as provided by the Business Analyst (BA)
  - Produces High-Level Solution Designs to be fleshed out and implemented by development teams
  - Identifies existing (legacy) components that can be reused
  - Identifies new components that need to be procured or developed
  - Specifies integration of existing and new components to meet business need

- Anyone else with an interest in end-to-end solution design involving integration of applications and systems
Why “Practical SOA”?  

- Some IT practitioners think SOA is a buzzword that is now passé  
- Some understand the concept of “loose coupling” in theory but don't know how to apply it in practical situations  
- Some think that just using a “SOA Product” like an ESB is sufficient to “do SOA”.  
- In practice, many ostensibly “SOA-based” designs running on SOA Products are tightly-coupled, violating the spirit of SOA.  
- We need to re-educate Solution Architects about SOA  
- We need an approach that is simple to understand and easy to apply – hence “Practical SOA”. 

Oct 25, 2011
The Two Layers of a Solution Design

- Solution Architects must choose the right tool for the job at the Technology Layer.
- They must ensure loose coupling at the Data Layer to avoid negating the investments at the Technology Layer.

**The Data Layer**
- Implicit assumptions and dependencies made explicit
- Required for loosely-coupled interoperability
- Decoupling of Message Data from Domain Data
- Common vocabulary (within practical limits)

**The Technology Layer**
- Standardised Wire Protocol (e.g., SOAP, HTTP)
- Standardised Data Encoding (e.g., XML, JSON)
- Standardised Message Exchange Patterns (e.g., Request/response, One-way)
- Standardised Qualities of Service (e.g., Security, Reliability)

Solution Architects are responsible for designing this layer. SOA products support, but do not guide, this design.

SOA products support loose coupling at this layer, but Solution Architects must choose the right tools and patterns from the offering.
The Three Core Components at the Technology Layer

- The Service Container
- The Broker
- The Process Coordinator
The Service Container

The business requirement demands a component:

- that does not yet exist
- is not available off the shelf
- must be developed in-house

Desirable to expose this new component's functionality in a reusable way

A *Service Container* is what you need to host this logic
The Broker

In practice, solution must be able to exploit pieces of logic or data that *already exist* somewhere in the organisation, within some legacy system or application.

Possible complications:

- Legacy component speaks a specialised or proprietary *protocol*. Use an *adapter* to translate proprietary protocols into more open ones.

- The *data* is not in the right format for you to use. Use a *transformer* to enrich data and massage it into a more usable form.

- Need to hide, proxy or throttle access to legacy data or app. Use a *mediator*. 
The Broker, cont'd

A **Broker** component performs the functions of an adapter, a transformer and a mediator.
The Process Coordinator

Sometimes you don't know what you need in advance. To pull components together *dynamically*, based on a run-time status, you need a **Process Coordinator**.

It can decide which automated and human tasks to invoke at each step of the process, and make these decisions dynamically.
Supporting Components

The Service Container, Broker and Process Coordinator are often sufficient for a basic solution design. But some components can refine a solution with specialised functionality.

- Rules
- Data Access
- Registry/Repository
- Governance Support
- Activity Monitoring
- Complex Event Processing
- Presentation Support
- Identity and Access Management
SOA Technology Components from WSO2
Common Broker-related Mistake #1

“When All You Have is a Hammer, the Whole World Looks Like a Nail”

Brokers are very powerful and have many extensions, but...

- Do not use a Broker in place of a Service Container by hosting business logic
- Do not use a Broker in place of a Process Coordinator by hosting coordination logic

Use the right tools for the job - ensure that developers are equally proficient in the use of all three core SOA components
Common Broker-related Mistake #2

A Broker is not a Singleton, Centralised Component

Some commercial Brokers are expensive. Temptation for a hub-and-spokes architecture – i.e., install just one Broker instance for the entire organisation and push all traffic through it

Implications: Performance (scalability) and Availability (single point of failure)

A decentralised or “federated” architecture is better, and ties in very well with a pragmatic data model.
A Data Design Example

A business application needs to store documents (files) alongside structured data. It can hold references to the locations of these documents but not the document files themselves.

An Enterprise Content Management (ECM) system can store the documents themselves.

When a document is checked into the ECM, it returns a document ID.

If a document ID is presented to the ECM, the corresponding document is retrieved.

How can the business application interface with the ECM?

Would the design below suffice?
Data Design Example – Analysis

The simplistic design fails for many reasons:

- Document IDs are unique only within an ECM instance, not globally.
- There are many instances of ECM – which one is implied?
- Need for business app to store implicit information – ECM instance ID
- ECM instance ID is assumed to be a surrogate for the application – metadata transformation is done based on the ECM instance ID.

Implications:

- Too many instances of ECM – licence and operational costs
- Cannot load-balance multiple ECM instances for high-volume applications
- Cannot share an ECM instance between low-volume applications
- Cannot move a document from one ECM instance to another
- Cannot implement flexible archiving strategies

This tightly-coupled solution is not SOA-compliant even though it uses the right component (a Broker) at the technology layer.
Design Design Example – One Solution

- Remove implicit knowledge of ECM instance – get it from a rules engine.
- Remove unwarranted dependency of transformation logic on ECM instance – get it from a rules engine.
- Decouple business applications from instance-specific Document IDs – use a mapping to a global ID.
- Control the steps using a Process Coordinator in front of the Broker.

1. Invoke Document Checkin Process passing document and document metadata
2. Determine correct ECM instance and transformation logic required, passing document metadata
3. Check document into correct ECM instance specifying transformation logic, obtain instance-specific document ID
4. Generate globally unique ID and store mapping of this to the ECM instance and the instance-specific document ID
5. Return globally unique ID for application to store
Data Layer Principles

Lessons learnt:

- Identify Dependencies, Explicit and Implicit. Make all Implicit Dependencies Explicit
- Eliminate Unnecessary Dependencies
- Map Domain Data to Message Data (Do Not Derive or Generate)
- Develop a Sane Data Model for Message Data (Intermediate granularity)
**Impractical Approach – Boiling the Ocean in Search of a Canonical Data Model (and Suffering the Performance and Availability Drawbacks of a Centralised Broker)***

*All interactions between systems go through a centralised Broker.*

*All data is translated by the Broker to/from a single canonical data model and the various data models understood by individual systems.*
Pragmatic Approach – A Federated Data Model Based on Logical Domains, and a Correspondingly Federated Broker Configuration

* A Domain Broker enforces a Domain Data Model for a group of related systems.

* External systems see a consistent Data Model (but different from their own) when interacting with any service exposed through a Domain Broker and hosted on systems within that domain.

* It is the responsibility of systems to understand other data models when interaction crosses domain boundaries (which is relatively rare). Local Domain Brokers perform any required translations.
Industry Example 1 - Banking

Opening an account:

After verifying identity, check customer database to see if person is an existing customer. If not, generate a fresh customer number.

Generate a new account number on mainframe system.

If new customer, requisition the cards system to issue a new debit card and have it linked to the account. If existing customer, link new account to existing debit card.

The card system makes a request to a PIN mailer to have a PIN sent to the customer.
Industry Example 1 – Solution

Bank Account Opening – High-Level Solution Design Showing SOA Components

1. Check if existing customer,
2. Create customer,
3. Create account,
4. Link card to account,
5. Issue card

- Presentation support
- Branch officer
- Account Opening Front-end
- Account opening process
- Customer Data Services
- Data Access
- SQL
- Mainframe
- Card System
- SOAP/HTTP
- Card Services
- COBOL-XML transformation
- Queue adapter
- COBOL copybook over MQ
- Existing proprietary link
- PIN Mailer
- Transformation to bank schema
- Vendor-provided Web Services
- Transformation to bank schema
Industry Example 2 – Insurance

Providing a quote, converting a quote to a policy:

Insurance brokers ask for quotes from insurance companies on behalf of their customers. Customers may select the best quote and place an order to purchase a policy.

At an insurance company, a rules engine determines the quote based on the customer's and product's details. If it's a borderline case, it escalates to a human underwriter who makes a decision and communicates it back to the broker. Once a quote is provided, it is recorded on a mainframe system.

The broker's front-end application gathers quotes from insurance companies and presents them on a screen. When informed of the quotes, the customer may purchase one.

The broker places an order for the quote. The insurance company processes the payment through an external payment gateway. It then converts the quote recorded on the mainframe to a policy.
Industry Example 2 – Solution

Insurance Quotes – High-Level Solution Design Showing SOA Components

A. Get quote, B. Purchase policy

A1. Get quote decision
A2. Refer quote
A5. Get decisions

A6. Register quote, B2. Convert quote to policy
B1. Make payment

A7. Notify quote

A8. Get quotes

Broker

Inbox

Broker Data Services

Client

Presentation support

Broker Front-end Application

Quote Decisioning

Rules

Underwriter

Underwriter Data Services

Underwriter Inbox

Presentation support

Underwriter Front-end Application

Payment Gateway

Payment Service

Service Provider Interface

Queue adapter

COBOL copybook over MQ

COBOL-XML transformation

Policy Services

Mainframe

Oct 25, 2011
Summary and Conclusions

The success of SOA depends on the Solution Architect.

The Solution Architect must pay attention to the Technology Layer as well as the Data Layer of a solution design.

Technology Layer – Use the right tool for the job (Service Container, Broker and Process Coordinator), supplement with supporting components as appropriate. Steer clear of the common Broker-related design mistakes.

Data Layer – Remember 4 principles:

1. Make all implicit dependencies explicit
2. Remove unnecessary dependencies, put all legitimate ones into the contract
3. Loosely couple message data with domain data – do not derive or generate one from the other
4. Choose an intermediate level of granularity for data models – neither too coarse-grained nor fine-grained

Study industry examples to make these concepts real.