Service API and Contract Design with Web Services

hungdn@ptit.edu.vn

1. Service Model Design Considerations

Entity Services vs. Utility Services

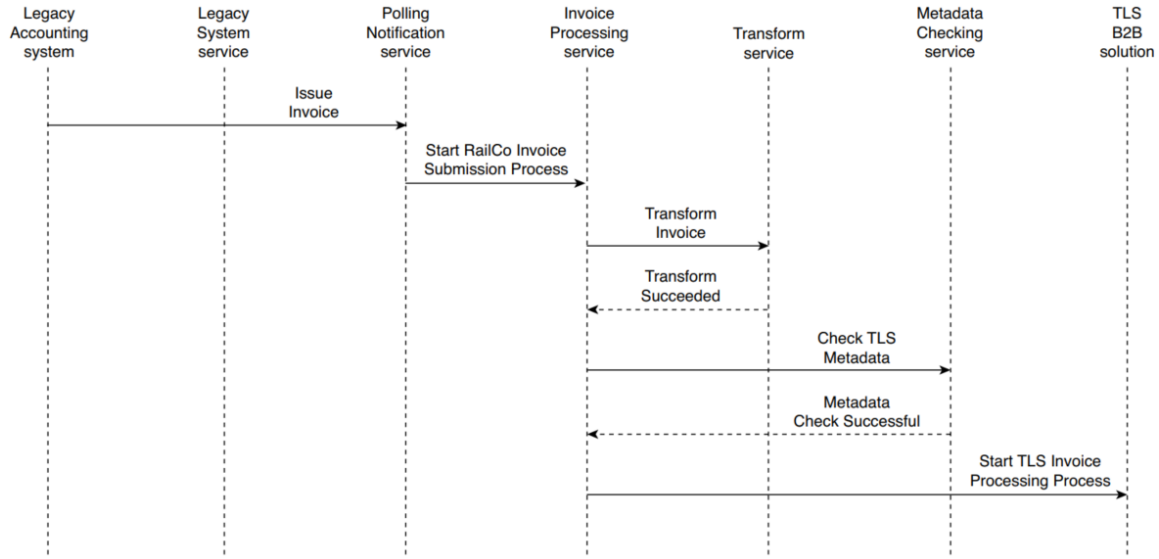
Aspect	Entity Services	Utility Services
Purpose	Represent business entities (e.g., invoices, customers)	Handle low-level processing functions
Business Relation	Business-centric, independent of processes	Technology-focused, often abstract legacy systems
Reusability	High – reusable across business processes	High – provides generic functionality
Design Considerations	Standardized XML schemas, metadata for discoverability	Flexible APIs, avoid redundancy, consider third-party options
Implementation	Often SOAP-based with WSDL & XML schemas	Can use SOAP, REST, or other protocols based on need

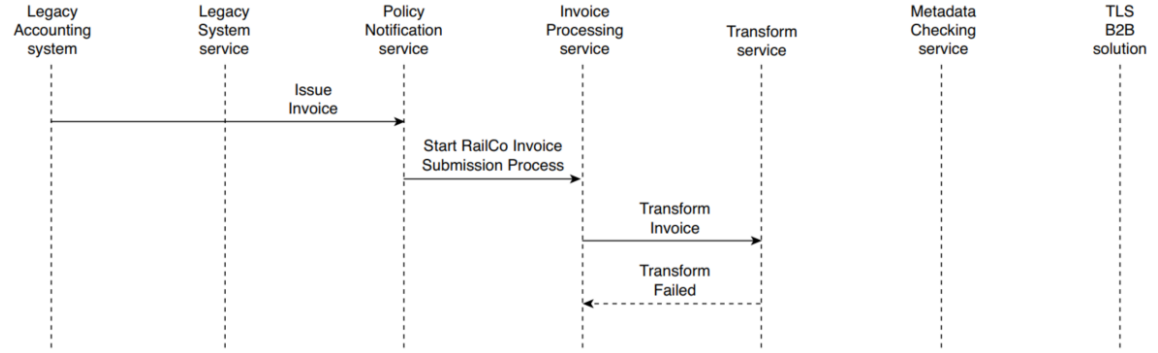
Microservices vs. Task Services

Aspect	Microservices	Task Services
Purpose	Small, independent services for high performance & scalability	Coordinate workflows & service compositions
Business Relation	Typically business-agnostic, focuses on specific functionalities	Business process-driven, manages execution logic
Reusability	High – lightweight, designed for modular reuse	Medium – specific to workflow but can be reused
Design Considerations	REST-based, avoid SOAP overhead, optimized for performance	Requires workflow logic, state management, and exception handling
Implementation	Uses REST APIs, often containerized (e.g., Docker, Kubernetes)	May use SOAP with document-style messages for state persistence

State Management in Task Services

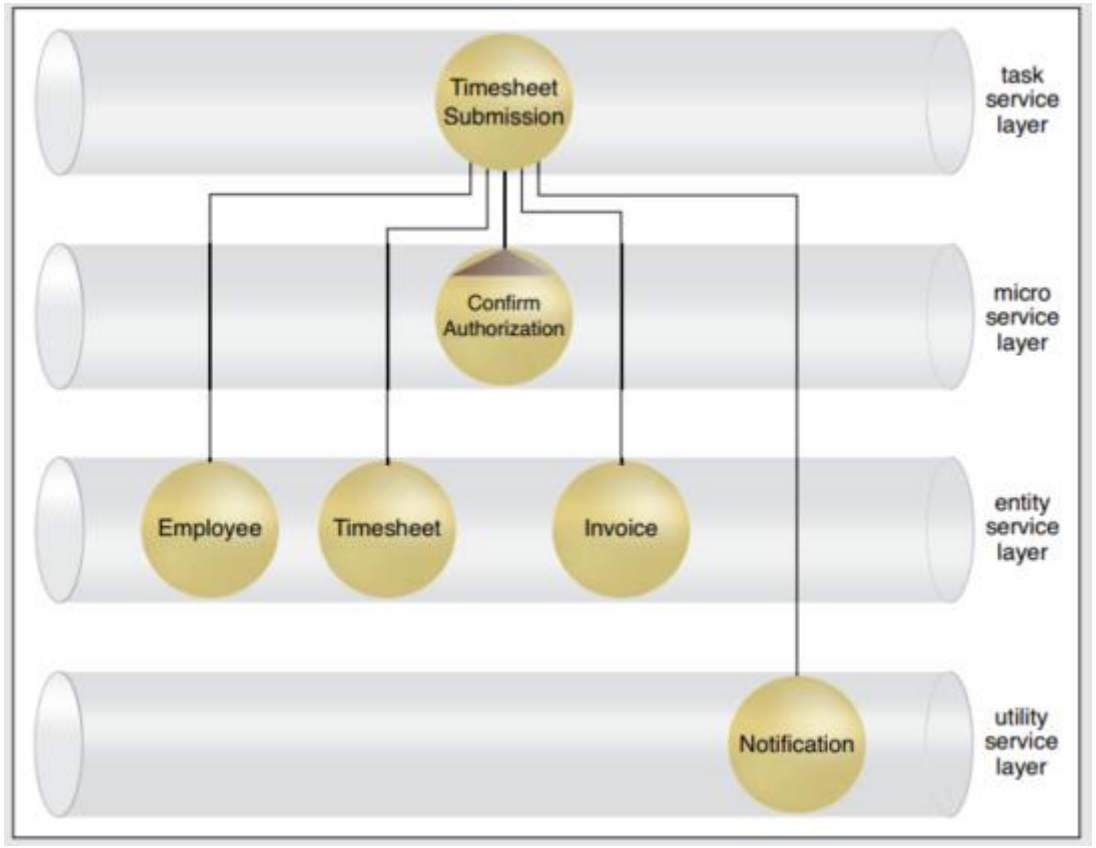
- **Task services often require state management to handle workflow logic effectively.**
  - SOAP messages can be used to store and transfer state information
- **State Management Patterns**
  - State Repository
  - Partial State Deferral
  - State Messaging
- **Transaction Handling**
  - Atomic Transactions
  - Compensating Transactions





CASE STUDY EXAMPLE

- The service modeling process performed by TLS produced a number of Web service candidates in new TLS Timesheet Submission solution



The contract design of the Employee service

- **The Employee Service's function:**
  - Query employee's **weekly work hour limit**
  - Update **employee history** when a timesheet is rejected

- **TLS invested in creating a standardized XML Schema data representation architecture, but accounting environment only.**

Employee Service - Querying Work Hours

- **Schema Considerations**
  - Existing accounting schema is too large and complex
  - Created a lightweight, compliant schema version
- **Solution:** Created new Employee.xsd schema by define two complex types
  - Search criteria (request parameters)
  - Query results (returned data)

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/accounting/">
  <xml:element name="EmployeeHoursRequestType">
    <xml:complexType>
      <xml:sequence>

        <xml:element name="ID" type="xml:integer"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeHoursResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="WeeklyHoursLimit"
          type="xml:short"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

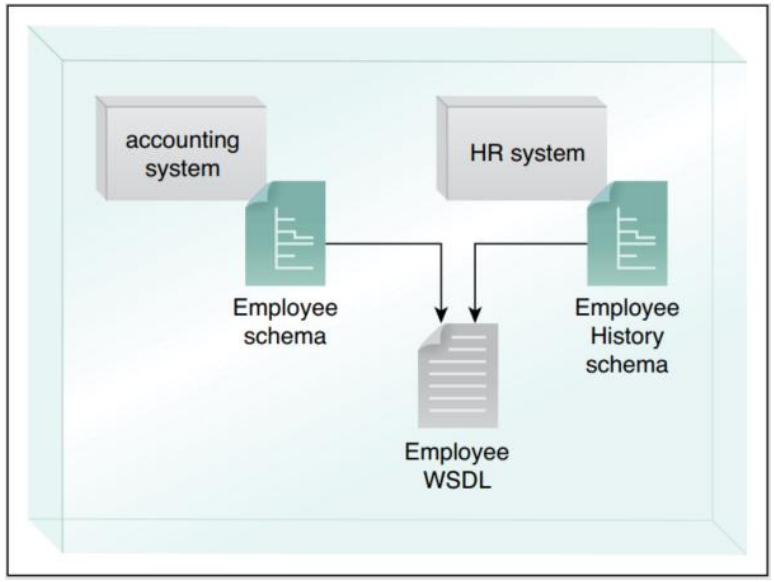
Employee Service - Updating Employee History

- **Problem**
  - Employee history not in the accounting schema and
  - A part of the HR environment
- **Solution:**
  - Created new **EmployeeHistory.xsd** schema, with a different **targetNamespace** to identify its distinct origin

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/hr/">
  <xml:element name="EmployeeUpdateHistoryRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="Comment" type="xml:string"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeUpdateHistoryResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ResponseCode"
          type="xml:byte"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

Employee service – XML Schema

- Used XML Schema **Import** to integrate both schemas into WSDL **types**
  - Ensures reusability & maintainability




```
<types>
  <xml:schema targetNamespace=
    "http://www.example.org/tls/employee/schema/">
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/accounting/"
      schemaLocation="Employee.xsd"/>
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/hr/"
      schemaLocation="EmployeeHistory.xsd"/>
  </xml:schema>
</types>
```

Two schemas originating from two different data sources.

Defining the Initial Service Contract

- **Design considerations**
  - **Ensuring Reusability** – Validate operation granularity
  - **Defining Operations** – Create **portType** and operations
  - **Structuring Data** – Define input/output messages (.xsd) in WSDL
- **Operations Defined**
  - GetEmployeeWeeklyHoursLimit
  - UpdateEmployeeHistory

```
<message name="getEmployeeWeeklyHoursRequestMessage">
  <part name="RequestParameter"
    element="act:EmployeeHoursRequestType"/>
</message>
<message name="getEmployeeWeeklyHoursResponseMessage">
  <part name="ResponseParameter"
    element="act:EmployeeHoursResponseType"/>
</message>
<message name="updateEmployeeHistoryRequestMessage">
  <part name="RequestParameter"
    element="hr:EmployeeUpdateHistoryRequestType"/>
</message>
<message name="updateEmployeeHistoryResponseMessage">
  <part name="ResponseParameter"
    element="hr:EmployeeUpdateHistoryResponseType"/>
</message>
<portType name="EmployeeInterface">
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

Refining the Service Contract with Standardization

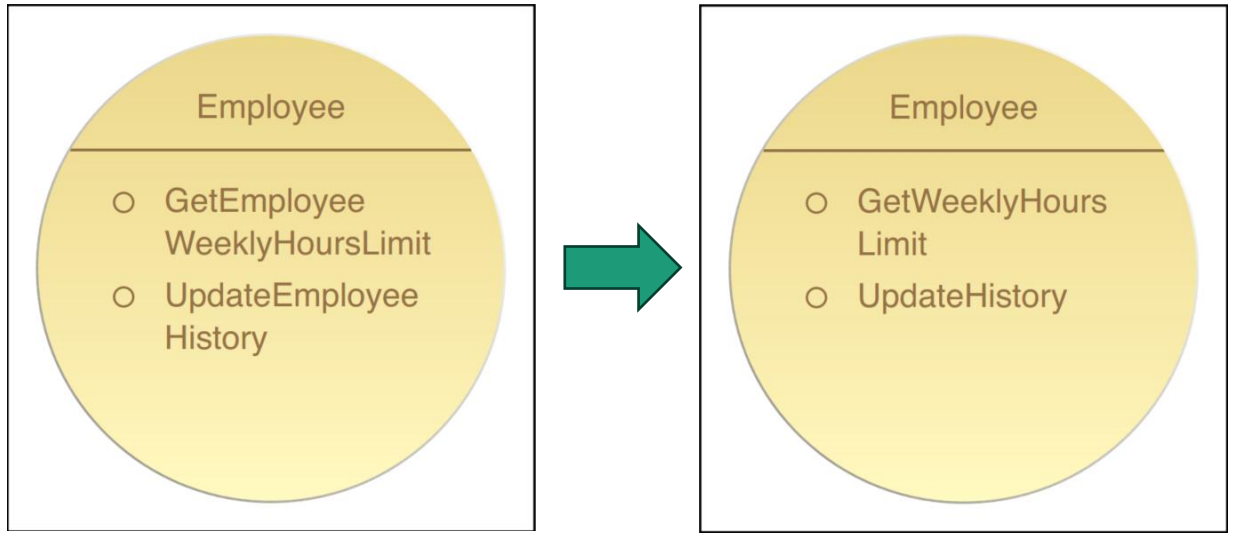
- **Reviewing & Refining the Contract**
  - Added **meta-information** to enhance service clarity
  - Applied **naming conventions** for consistency
- **Standardization Adjustments**
  - Improved operation and message naming
  - Revised abstract WSDL definition

```
<portType name="EmployeeInterface">
  <documentation>
    GetEmployeeWeeklyHoursLimit uses the Employee
    ID value to retrieve the WeeklyHoursLimit value.
    UpdateEmployeeHistory uses the Employee ID value
    to update the Comment value of the EmployeeHistory.
  </documentation>
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

Refining the Service Contract with Standardization

- **Reviewing & Refining the Contract**
  - Added **meta-information** to enhance service clarity
  - Applied **naming conventions** for consistency
- **Standardization Adjustments**
  - Improved operation and message naming
  - Revised abstract WSDL definition

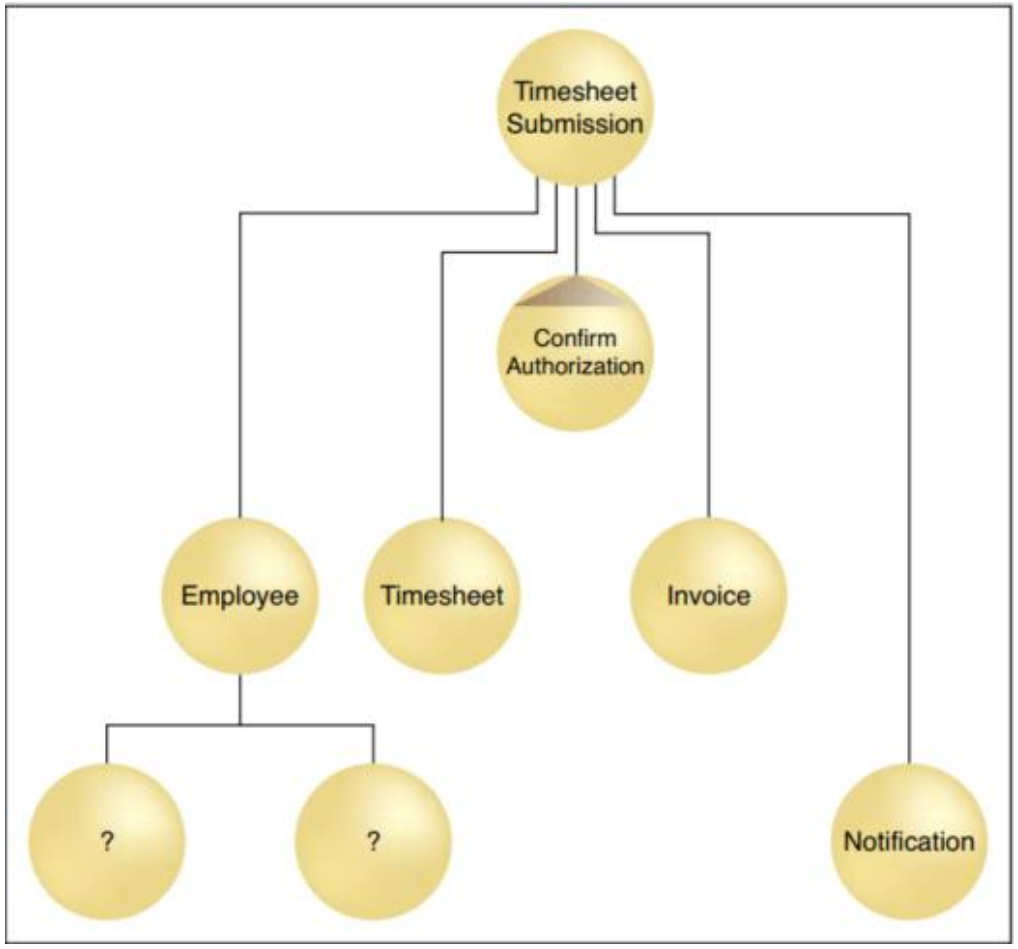
```
<operation name="GetWeeklyHoursLimit">
  <input message="tns:getWeeklyHoursRequestMessage"/>
  <output message="tns:getWeeklyHoursResponseMessage"/>
</operation>
<operation name="UpdateHistory">
  <input message="tns:updateHistoryRequestMessage"/>
  <output message="tns:updateHistoryResponseMessage"/>
</operation>
```



Finalizing the Employee Service Definition

- **Revise**
  - **Employee data** is stored in **two different repositories**
  - **Employee.xsd** schema **combines multiple data sources**, requiring alignment

- **Solution & Next Steps**
  - Introduce **HR wrapper utility service** to support Employee Service
  - Proceed with **concrete service definition & implementation**



2. Web Service Design Guidelines

Naming Standards

- **Importance of Naming Standards**
  - Ensures **clarity, consistency, and maintainability** of Web service contracts
  - Helps services be **self-descriptive** and **easier to reuse**

- **Ensure uniformity across all services in the inventory**

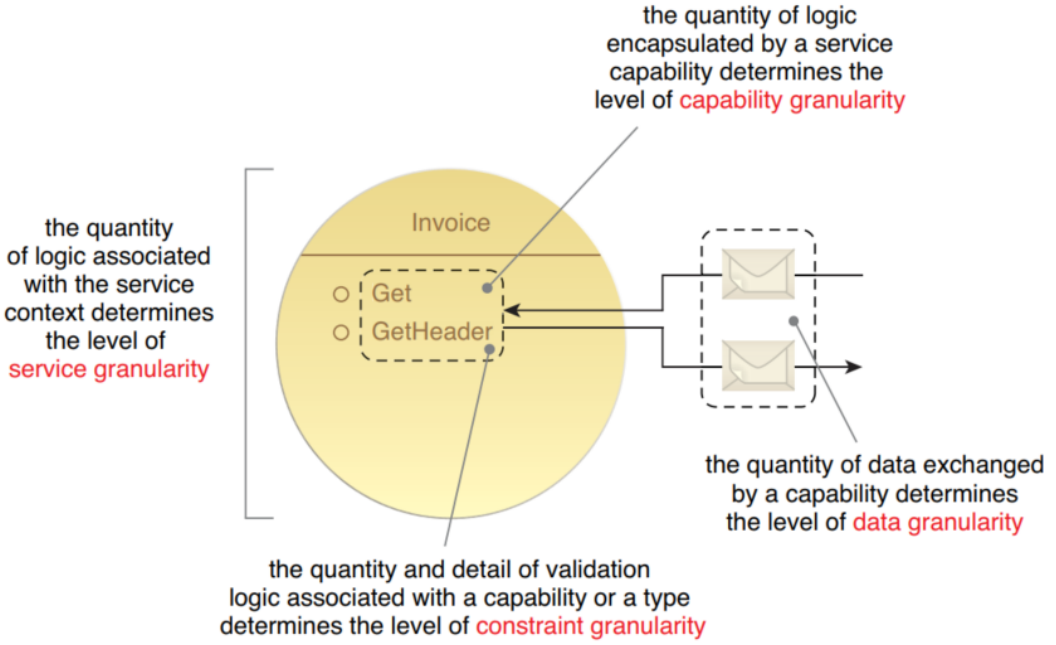
Guidelines for Naming Services & Operations

- **Service Names**
  - **Entity Services:** Use **noun-only** names (e.g., *Invoice, Customer, Employee*)
  - **Utility Services:** Use **verb+noun** or **noun-only** (e.g., *GetStatistics, SalesReporting*)
- **Operation Names**
  - **Entity Service Operations:** Use **verb-only** (e.g., *Create, Update, Delete*), **avoid repeating entity name**
  - **Utility Service Operations:** Clearly describe functionality (e.g., *GetReport, ConvertCurrency*)

Understanding API Granularity in Web Service Design

- **Fine-Grained vs. Coarse-Grained APIs**
  - **Fine-Grained:** Small, focused operations with limited data exchange
  - **Coarse-Grained:** Broad operations that handle multiple tasks at once

- **Choosing the right level of granularity is essential for optimizing both performance and reusability.**



Best Practices for API Granularity

- **Strategic considerations for API Granularity**
  - Coarse-grained APIs improve **performance but reduce reusability**
  - Fine-grained APIs increase **reusability but may impact performance**
- **Guidelines**
  - Assess Performance Needs
  - Offer Alternative APIs
  - Optimize for Consumers
  - Support Multiple Protocols
  - Maintain Consistency
- **Case Study: TLS Approach**
  - **External APIs**: Coarse-grained for **performance optimization**
  - **Internal APIs**: Less coarse-grained to **enhance reusability**
- **Balancing granularity is key to a scalable and efficient SOA architecture!**

Designing Extensible Web Service Operations

- **Why Extensibility?**
  - Business needs evolve, requiring **service updates**
  - Extensible design ensures **minimal contract changes**
  - Supports **future non-specific values & functions**
- **Guidelines**
  - Keep Operations Activity-Agnostic
  - Leverage Composition Over Modification
  - Use Schema Extensions Cautiously

Case Study: TLS Employee Profile Verification

- Employees **rarely update their profiles** after promotion
- **New requirement**: Verify profiles before processing timesheets
- **Solution**: Add a **separate utility service** for profile verification instead of modifying the Timesheet service contract

- **Extensible design reduces contract changes and ensures long-term flexibility!**

Best Practices for WSDL Design

- **Modular WSDL Documents for Flexibility**
  - Use **import statements** to dynamically assemble WSDL at runtime
  - Separate **types, operations, and bindings** for reuse across multiple services
  - Leverage **existing XML Schema modules** for efficient management
- **Careful Namespace Management**
  - WSDLs involve multiple namespaces for **specification-based elements**
  - Use **targetNamespace** to define a consistent structure
  - Organize namespaces to prevent **complexity & conflicts**
- **SOAP Message Formatting**
  - **style:** Defines message structure (**document** or **RPC**)
  - **use:** Determines data type system (**literal** or **encoded**)
  - **Best Practice:** Use **style:document + use:literal** for SOA compliance

Q & A