Detailed Notes on Entity-Centric Business Services

Definition

Entity-centric business services are specialized service layers focused on accurately representing individual business entities (e.g., Employee, Ledger, or Accounts Payable).

- **Purpose**: Provide a reusable interface for applications to access and manage entity-specific data.
- **Characteristics**:
  - **Business Process-Agnostic**: Independent of any specific workflows.
  - **Solution-Agnostic**: Can be used across multiple solutions without modification.
  - **Reusable**: Designed to be used by various applications requiring access to the underlying entity.

Example:
An *Employee Service* may provide operations to:

1. Retrieve an employee's weekly working hours limit.
2. Update an employee's historical records after a rejected timesheet.

These operations encapsulate data-related tasks while avoiding dependencies on a specific business process.

Design Process Steps

Step 1: Review Existing Services

Objective: Avoid duplication of functionality by analyzing existing services.

- Check whether:
  1. The required operations already exist in other services.
  2. Existing services can be extended to incorporate new functionalities.

Example:
In a company like TLS:

- Existing services included **Accounts Payable**, **Ledger**, and **Vendor Profile**. These were analyzed to ensure there was no overlap with the proposed **Employee Service**.
- Since these services dealt with distinct entities, the team confirmed that a new Employee Service was necessary.

Step 2: Define Message Schema Types

Objective: Specify the structure of data exchanged using schemas.

- **Approach**:
  - Use **XSD (XML Schema Definitions)** to define message formats carried in SOAP messages.
  - Create **entity-centric schemas** representing entity data.

Example:

1. TLS defined `Employee.xsd` for:
   - Request: Search criteria for employee data.
   - Response: Results of the employee query.
2. Another schema, `EmployeeHistory.xsd`, was created because historical employee data was stored in a different HR repository.

Implementation in WSDL:

Import schemas into the `types` section of WSDL:
xml
Copy code

```
<import namespace="http://example.com/tls/employee/schema/accounting/"
schemaLocation="Employee.xsd"/>
<import namespace="http://example.com/tls/employee/schema/hr/"
schemaLocation="EmployeeHistory.xsd"/>
```

-

Step 3: Derive Abstract Service Interface

Objective: Define generic, reusable service operations.

- Use the **portType** element in WSDL 1.1 to describe service operations abstractly.

Steps:

1. Define reusable operation candidates (e.g., `GetWeeklyHoursLimit`, `UpdateEmployeeHistory`).
2. Map inputs and outputs to message types defined in schemas.
3. Populate WSDL with `operation` and `message` constructs.

Example:

Abstract Service Operations for Employee Service:
xml
Copy code

```
<portType name="EmployeeService">
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message="tns:GetEmployeeRequest"/>
    <output message="tns:GetEmployeeResponse"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message="tns:UpdateHistoryRequest"/>
    <output message="tns:UpdateHistoryResponse"/>
  </operation>
</portType>
```

-

Step 4: Apply Principles of Service-Orientation

Key Principles:

1. **Reusability**:
   - Operations must be generic to allow usage across multiple scenarios.
   - Modular schemas encourage reusability by separating functionality into components.
2. **Autonomy**:
   - Services should function independently without dependencies.
3. **Statelessness**:
   - Avoid maintaining state in the service. State-related operations can be handled through document-style SOAP messages.
4. **Discoverability**:
   - Add metadata in the `documentation` element for easier service identification.

Example:

Metadata for the `GetEmployeeWeeklyHoursLimit` operation:
xml
Copy code

```
<documentation>
  Retrieves the weekly hours limit using Employee ID as input.
</documentation>
```

-

Step 5: Standardize and Refine the Service Interface

Objective: Improve consistency and usability by adhering to standards.

- Implement naming conventions, guidelines, and WS-I Basic Profile compliance.

Example:

- Renamed operations to ensure intrinsic interoperability:
  - `GetEmployeeWeeklyHoursLimit` -> `FetchWeeklyWorkLimit`.
  - `UpdateEmployeeHistory` -> `ModifyEmployeeRecord`.

Step 6: Extend the Service Design

Objective: Add functionality to meet broader requirements.

- **Approaches**:
  1. Add new operations (e.g., `AddEmployee`).
  2. Extend existing operations with new parameters.

Example:

- Extend Employee Service with CRUD operations:
  - `GetEmployee`
  - `UpdateEmployee`
  - `AddEmployee`
  - `DeleteEmployee`

Note: Balance reusability and complexity—too many parameters can confuse service consumers.

Step 7: Identify Required Processing

Objective: Determine application services needed to implement operations.

- Analyze the data and logic required for each operation.

Example:

- For the `GetWeeklyHoursLimit` operation:
  - Query the **Accounting Database** to retrieve hours using Employee ID.
- For the `UpdateHistory` operation:
  - Update the **HR Repository** to modify the employee's history.

Outcome:

- Identified the need for a **Human Resources Wrapper Service** to process employee-related data.

Key Concepts

1. **Granularity**:
   - Operations should balance simplicity and completeness. Avoid overly complex operations.
   - Example: `GetWeeklyHoursLimit` is fine-grained but meets the business need.
2. **Abstract vs. Concrete Definitions**:
   - **Abstract Definition**: Focuses on operations and data types.
   - **Concrete Definition**: Adds details like protocols and endpoints.

Case Study Highlights

- **TLS Implementation**:
  - Created schemas (`Employee.xsd`, `EmployeeHistory.xsd`) for employee and HR data.
  - Designed operations (`GetWeeklyHoursLimit`, `UpdateEmployeeHistory`).
  - Applied naming conventions and modular schemas for reusability.
  - Introduced a wrapper service for accessing multiple data repositories.

Summary

1. Entity-centric business services focus on specific data entities while remaining independent of processes and solutions.
2. They promote reusability, autonomy, and discoverability through modular design.
3. The design process involves careful planning, standardization, and possible extensions to ensure long-term usability.

These detailed notes provide comprehensive coverage, ensuring you're well-prepared for exams. Let me know if you need further clarification!