

[Send a Simple Message to a Service - WSO2 API Manager Documentation 4.2.0](https://apim.docs.wso2.com/en/4.2.0/tutorials/integration-tutorials/sending-a-simple-message-to-a-service/)

**WSO2 Micro Integrator** is a lightweight, cloud-native integration engine designed to facilitate communication between microservices, APIs, and external systems in a distributed architecture. It enables seamless integration by handling tasks like message transformation, routing, service orchestration, and error handling. Essentially, it acts as middleware that connects and coordinates multiple services, making it easier to build complex workflows and connect to databases or legacy systems without tightly coupling components.

The diagram illustrates the role of **WSO2 Micro Integrator** in micro services architecture and shows how it works as a middleware to enable seamless communication between services and external systems. Here’s a breakdown of the main components and how they interact in the diagram:

**Key Components in the Diagram**

1. **API Gateway**: This serves as the entry point for external requests, usually coming from external users or systems. It exposes APIs that clients can interact with and handles tasks like request routing, rate limiting, and security.
2. **Micro Integrator**: WSO2 Micro Integrator is shown here as a mediator between different microservices and external systems. It helps orchestrate services, handles complex integration logic, and provides features like message transformation, routing, and error handling.
3. **Microservices**: These are independent, self-contained services that handle specific business capabilities. Each microservice performs a specific function (such as order processing, payment, or inventory management) and communicates with other services through the Micro Integrator when necessary.
4. **External Systems**: These can include databases, external applications, legacy systems, or any external resource that the microservices might need to interact with.

**Flow of Interaction**

1. **Request Flow**:
   * A request originates from the API Gateway and is directed toward the appropriate **Micro Integrator** instance.
   * The **Micro Integrator** acts as an intermediary that routes the request to the relevant microservice based on the request type or content.
2. **Service Orchestration**:
   * The **Micro Integrator** can orchestrate multiple microservices to fulfill a complex business request. For example, an order fulfillment request might require communication with inventory, payment, and shipping microservices. The Micro Integrator handles this by routing requests and coordinating responses among services.
3. **Communication with External Systems**:
   * The **Micro Integrator** connects to external systems such as databases, file servers, or external applications to retrieve or store data as part of a business process.
   * For example, when an order is placed, it might need to retrieve customer information from a database or store transaction details in a central database. The Micro Integrator handles this connection seamlessly.

**Example to Illustrate**

Let’s walk through a simple example of an **Order Processing System** that uses this architecture:

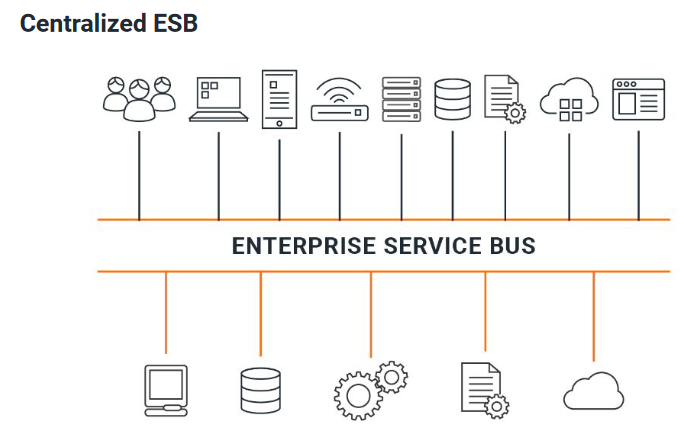
* **Step 1**: A user places an order through a frontend application. The request is sent to the **API Gateway**.
* **Step 2**: The API Gateway directs the request to the **Micro Integrator**, which is responsible for coordinating the order processing.
* **Step 3**: The Micro Integrator then does the following:
  + Routes the order details to the **Order Processing Microservice** to validate and process the order.
  + Calls the **Inventory Microservice** to check if the items are in stock.
  + Interacts with the **Payment Microservice** to charge the user’s payment method.
  + Updates the **Shipping Microservice** to arrange delivery of the items.
* **Step 4**: During this process, the **Micro Integrator** might also need to retrieve or store information in an **external database** or notify an external **CRM system** about the new order.
* **Step 5**: Once all steps are completed, the Micro Integrator returns a consolidated response back to the API Gateway, which then relays it to the frontend application.

**Summary**

In this architecture:

* The **API Gateway** manages external access.
* **Micro Integrator** handles complex workflows, service orchestration, and communication with external systems, which helps decouple microservices from knowing the details of other services or external systems.
* **Microservices** are isolated units, focusing on specific tasks and rely on the Micro Integrator for any inter-service communication or access to external resources.

This setup provides flexibility, scalability, and maintainability by separating the integration logic from the business logic of individual services, enabling each component to focus on its specific role.



The image shows a **Centralized ESB (Enterprise Service Bus)** model, where the ESB acts as a central hub connecting multiple systems, applications, databases, and services.

**Explanation:**

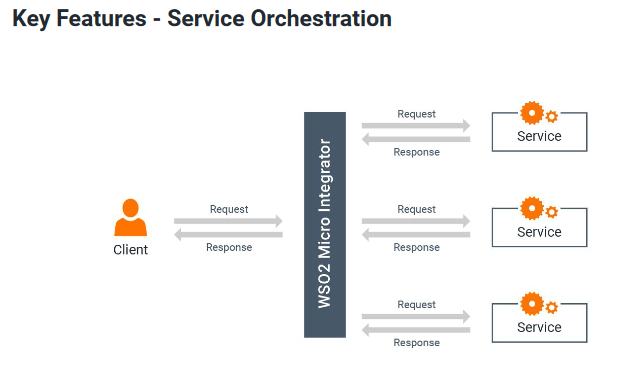
In this setup, the **WSO2 Micro Integrator** functions as the ESB, providing a central integration layer for handling communication between all connected systems. It routes messages, transforms data, and manages events, serving as a bridge between different technologies and data formats.

**Key Points:**

1. **Central Hub**: The ESB is a single, central platform that connects all systems, including users, databases, cloud services, and APIs.
2. **Message Mediation**: The ESB routes and transforms messages as they move between different systems, ensuring compatibility and smooth communication.
3. **Integration Layer**: By centralizing the integration logic, the ESB simplifies and streamlines interactions, making it easier to manage complex systems.

**Example:**

In a business with multiple departments using different software (e.g., CRM, ERP, databases), the ESB (WSO2 Micro Integrator) connects them all, allowing data to flow smoothly between systems without each one having to directly connect with the others.



**Service Orchestration** in WSO2 Micro Integrator is about combining multiple smaller services into a single, larger service that clients can interact with easily. Instead of clients calling each small service individually, they make one request to WSO2 Micro Integrator, which manages the entire workflow by calling each necessary service behind the scenes.

**Types of Service Orchestration**

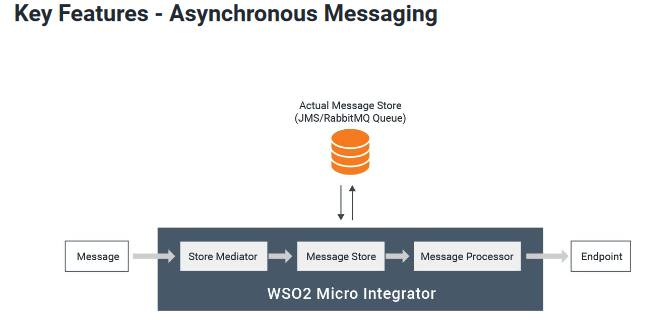
1. **Synchronous Service Orchestration (Service Chaining)**:
   * Services are called one after another in a set sequence.
   * Each service’s output becomes the input for the next one.
   * Example: An order processing workflow where inventory, payment, and shipping services are called in sequence.
2. **Asynchronous Service Orchestration (Parallel Invocations)**:
   * Multiple services are called simultaneously without waiting for one to finish before starting the next.
   * Useful when services don’t depend on each other’s outputs.
   * Example: Sending notifications via email, SMS, and push notifications all at the same time.

In both cases, WSO2 Micro Integrator handles the orchestration, simplifying complex workflows for the client.

* **Key Features of WSO2 Micro Integrator:**

1. **Service Orchestration**: Combines multiple services into a single workflow, supporting both synchronous (chained) and asynchronous (parallel) invocations.
2. **Message Transformation**: Converts messages between different formats (e.g., XML to JSON) to ensure compatibility between systems.
3. **Message Routing**: Directs requests to specific services based on content, headers, or custom logic.
4. **Error Handling**: Manages errors with custom fault sequences, retries, and fallback mechanisms for reliable integration.
5. **Protocol Support**: Supports multiple protocols like HTTP, HTTPS, JMS, SOAP, and more for versatile connectivity.
6. **Lightweight & Cloud-Native**: Designed for containerized environments with a small footprint, ideal for microservices.
7. **Connectors**: Provides pre-built connectors for popular applications and services (e.g., databases, CRMs).
8. **Monitoring & Observability**: Tracks service performance and errors with logging, metrics, and dashboards.
9. **Security**: Supports authentication, authorization, and encryption for secure communication.

These features make WSO2 Micro Integrator a powerful tool for integrating and managing complex workflows across diverse systems.



The image illustrates the asynchronous messaging process within the WSO2 Micro Integrator. Here’s a breakdown:

1. **Message Flow**: A message is sent into the system and goes through a series of steps for processing asynchronously.
2. **Store Mediator**: This component intercepts the message and stores it temporarily in a **Message Store** (like a JMS or RabbitMQ queue) instead of processing it immediately.

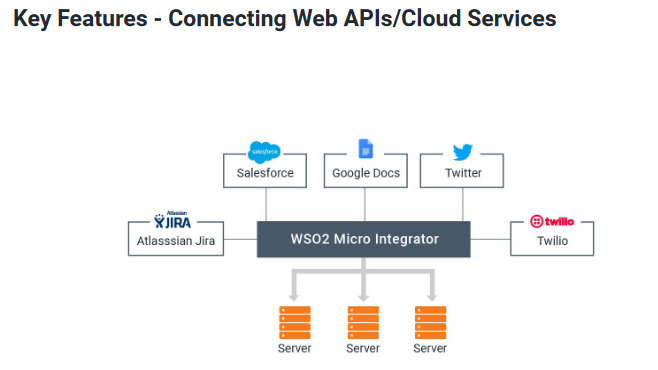
* **JMS (Java Message Service)**: A Java API for sending messages between two or more clients, enabling reliable communication in distributed systems.
* **RabbitMQ Queue**: A message broker that stores and forwards messages, often used to manage tasks asynchronously in scalable, distributed applications.

1. **Message Store**: Acts as a middle layer where messages are stored until they are ready to be processed. This queue enables reliable message delivery, even if the receiving system is temporarily unavailable.
2. **Message Processor**: Retrieves messages from the **Message Store** and sends them to the final **Endpoint** for processing.

**Key Points of Asynchronous Messaging:**

* **Advantages**: Allows the system to continue processing without waiting for a response, provides reliable message delivery, manages varying message rates, and enables batch processing.
* **Disadvantages**: Adds complexity by requiring a message broker or transfer agent, which may affect performance and reliability due to the additional steps for message storage and retrieval.

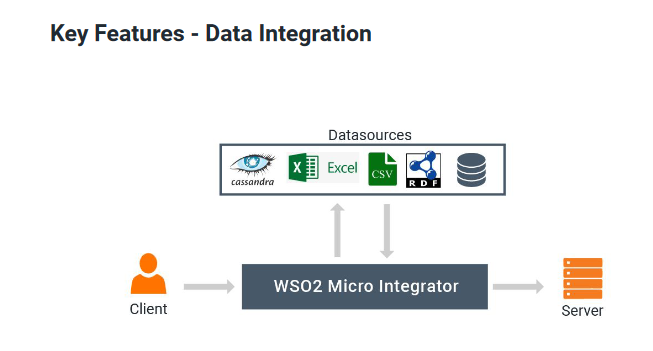
In summary, asynchronous messaging improves reliability and flexibility in message delivery, but it requires additional components like message brokers, which can impact system performance.



The image illustrates WSO2 Micro Integrator’s ability to connect with various Web APIs and cloud services, a feature often called **Hybrid-Cloud Integration**. Here’s a clear breakdown:

* **Hybrid-Cloud Integration**: WSO2 Micro Integrator can seamlessly connect with third-party applications via public APIs. These applications can either be cloud-based (e.g., SaaS) or hosted on-premise.
* **Supported Services**: The integrator connects with widely used services like Salesforce, Google Docs, Twitter, Jira, and Twilio, allowing communication and data exchange across diverse platforms.
* **Connectors**: WSO2 Micro Integrator supports over 150 connectors, downloadable from the WSO2 connector store. Developers can also create custom connectors to integrate new or specific systems not yet supported by default.

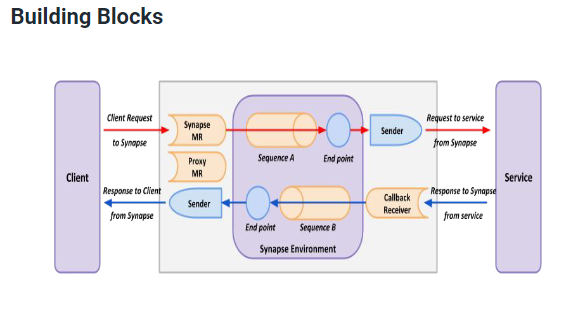
In summary, WSO2 Micro Integrator enables versatile integration across cloud and on-premise applications through a broad range of connectors, facilitating smooth interaction with external services.



The image illustrates WSO2 Micro Integrator’s **Data Integration** feature. Here’s a clear explanation:

* **Data Integration**: WSO2 Micro Integrator connects data from multiple, diverse data sources (such as Cassandra, Excel, CSV files, RDF, and databases) to enable unified data access.
* **Data Services**: The integrator abstracts data from these sources by exposing them as **data services**, allowing seamless interaction with the data without direct dependency on the underlying sources.
* **Data Manipulation**: Once exposed as data services, the integrator enables operations like **CRUD (Create, Read, Update, Delete)**, facilitating various data manipulations through the integration flow.

In short, WSO2 Micro Integrator unifies data from different sources, exposing them through data services for streamlined access and management in integration workflows.



The attached image represents the building blocks of a message processing environment in WSO2, showing how messages flow between a client and a service through Synapse.

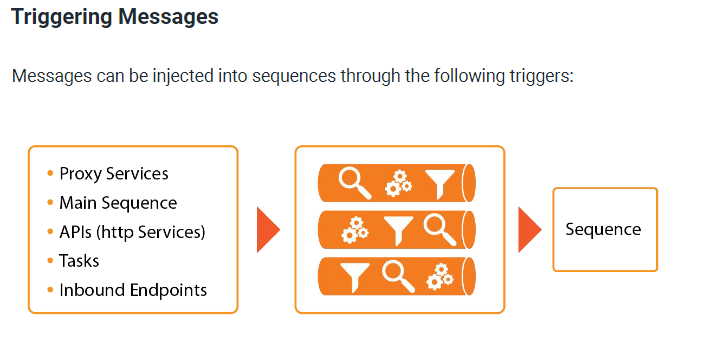
Here’s a simplified breakdown of each component based on the image and your description:

1. **Sequences**
   * Sequences organize the mediators, defining the steps and logic for processing requests and responses.
   * They define how the response will be handled and list mediators in the order they should be executed.
2. **Mediators**
   * Mediators are processing units that perform specific actions on messages, such as sending or filtering them.
   * The Synapse environment includes a library of mediators for common functions, and you can also create custom mediators using Java, scripting, or Spring.
3. **Endpoints**
   * Endpoints specify the external destination of a message, such as a service.
   * They are defined independently of transports, meaning the same endpoint can be used across multiple transports.
   * Common types include address endpoints, WSDL endpoints, and load balancing endpoints.
4. **Transports**
   * Transports are protocols that carry messages in specific formats between components.
   * WSO2 supports popular transports like HTTP, JMS, and VFS, along with domain-specific ones like FIX.
   * WSO2 transports are based on the Apache Axis2 framework.

**Flow in the Image**

* The **Client** sends a request to **Synapse** via the **Synapse Message Receiver (MR)**.
* The message passes through a sequence of mediators (e.g., **Sequence A** and **Sequence B**) within the **Synapse Environment**, where specific actions are taken.
* If needed, the message is sent to an **Endpoint** (an external service) through a **Sender**.
* The service response follows a similar path in reverse, from the **Service** back through Synapse to the **Client**.

This flow illustrates how messages are processed, directed, and handled using sequences, mediators, and endpoints with the help of various transport protocols in Synapse.



S**equence** is a predefined series of steps or instructions in WSO2 Integration Studio that determines how incoming messages are processed, transformed, routed, or logged. It acts as a workflow that takes a message from a trigger (like a Proxy Service or API), applies various processing rules, and directs the message to its final destination.

**Key Components of Message Triggering**

1. **Proxy Services**
   * **Definition**: A Proxy Service acts as an intermediary or virtual service between a client and the actual backend service.
   * **Purpose**: Proxy services are useful when you want to add security, routing, or transformation logic between clients and backend services. They let you control how messages interact with backend services without exposing the actual backend directly.
   * **Example**: Suppose you have a backend service that provides weather data, but you only want to expose a subset of that data to clients. You can create a Proxy Service that accepts client requests, retrieves the data from the backend, filters the results, and then sends only the necessary information to the client.

**How it works in a flow**:

* + Client sends a request to the Proxy Service (e.g., a request for weather information).
  + Proxy Service processes the message, possibly transforming or filtering it.
  + Proxy Service forwards the message to the backend or responds to the client.

1. **Main Sequence**
   * **Definition**: The Main Sequence is the default sequence in WSO2 Integration that processes any messages that are not directed through specific proxy services.
   * **Purpose**: It provides a general processing path for messages, especially when they don’t match any specific route or when other conditions are not met.
   * **Example**: Imagine you have multiple types of messages entering your system, but some messages don’t have specific routes defined for them. The Main Sequence would handle these messages by default. For instance, any message coming into the system without a designated Proxy Service or API would go to the **Main Sequence**.

**How it works in a flow**:

* + An unhandled message enters the system.
  + The message is directed to the Main Sequence.
  + The Main Sequence performs predefined actions (e.g., logging or generic transformation) and routes the message further.

1. **APIs (HTTP Services)**
   * **Definition**: APIs in WSO2 allow you to define HTTP services that accept RESTful requests, providing a flexible way for clients to interact with the system.
   * **Purpose**: APIs are used when you want to expose endpoints that allow more dynamic interactions. Unlike Proxy Services, APIs can provide RESTful access and support multiple operations like GET, POST, PUT, etc.
   * **Example**: Suppose you want to expose a REST API that clients can use to get information about different products. You could create an API with paths like /products/{id}, allowing clients to specify the exact product ID in their request. The API can then look up the relevant product information and return it to the client.

**How it works in a flow**:

* + Client sends a REST request (e.g., GET /products/123).
  + The API processes the request and decides how to route it or transform it based on the HTTP method and path.
  + The message is routed to a sequence for further processing.

1. **Scheduled Tasks**
   * **Definition**: Scheduled Tasks are automated tasks that run at specified intervals or times.
   * **Purpose**: Scheduled tasks allow for actions to be performed automatically, without a client triggering them. They are ideal for periodic tasks like data syncing, cleanup, or report generation.
   * **Example**: Suppose you have a task to fetch data from an external API every night at midnight. You can set up a scheduled task that automatically triggers a sequence to retrieve and process this data without needing any manual intervention.

**How it works in a flow**:

* + The Scheduled Task runs at the specified time (e.g., midnight every day).
  + It triggers a sequence to send a message to an external API, retrieve data, and process it.
  + The sequence then handles the message (e.g., saving data to a database).

1. **Inbound Endpoints**
   * **Definition**: Inbound Endpoints allow the integration of messages from various protocols (like HTTP, JMS, MQTT, etc.) into WSO2 EI without needing a Proxy Service.
   * **Purpose**: They enable the system to listen for and receive messages on specific protocols directly, allowing integration with external systems that use different protocols.
   * **Example**: Suppose you need to process messages from an IoT device using the MQTT protocol. You can set up an MQTT Inbound Endpoint to listen for messages from the device and direct them into a sequence for processing.

**How it works in a flow**:

* + An external device (e.g., IoT sensor) sends a message using a specific protocol (e.g., MQTT).
  + The Inbound Endpoint receives the message and triggers a sequence.
  + The sequence processes the message and routes it as necessary.

**Putting It All Together: Message Flow**

In WSO2 Integration, these triggers can inject messages into **sequences**, which are sets of instructions that define how the message should be processed, routed, transformed, or logged. The steps in a sequence often include **mediators** (components that handle specific processing tasks) and **endpoints** (destination addresses for messages).

Here's an example of a message flow in action:

1. **A client sends an HTTP request** to get weather data.
2. The **API** trigger (configured with an HTTP path for weather data) receives the request and decides which sequence to forward it to.
3. The **sequence** processes the message by calling an external weather service, transforming the data, and preparing the response.
4. The message is returned to the client, providing only the relevant weather information.

In this flow, **Proxy Services** could similarly handle different kinds of messages, **Scheduled Tasks** could periodically fetch weather updates, and **Inbound Endpoints** could be configured for non-HTTP protocols.

**Summary**

These triggers serve as entry points for messages in WSO2 Enterprise Integrator:

* **Proxy Services**: Act as virtual services for backend service protection and control.
* **Main Sequence**: Handles general messages that don’t go through a specific trigger.
* **APIs**: Provide RESTful access to sequences with more granularity and path-based routing.
* **Scheduled Tasks**: Run at specified intervals to automate routine tasks.
* **Inbound Endpoints**: Accept messages from various protocols directly into sequences.

Each trigger has unique use cases and enables flexible ways of integrating and managing message flows, making WSO2 Integration highly adaptable for different enterprise scenarios. Let me know if you need further clarification on any specific component!

The **Payload Factory Mediator** in WSO2 is used to create or modify the content (payload) of a message. It allows you to transform the message into a specific format (like JSON, XML, or plain text) by defining a custom template. This is useful when you need to reformat the incoming message before sending it to the next step or the final destination.

[Send a Simple Message to a Service - WSO2 API Manager Documentation 4.2.0](https://apim.docs.wso2.com/en/4.2.0/tutorials/integration-tutorials/sending-a-simple-message-to-a-service/)

In WSO2 (and many web services), *URI parameters* and *query parameters* are ways to pass data to a service. They help define and customize API requests, but they differ in how and where the information is included in the URL. Here’s a simple breakdown:

**1. URI Parameter**

* URI parameters (also called **path parameters**) are embedded directly within the URL path.
* They specify *specific resources* in the API and are usually used when accessing a particular resource by its ID or name.
* Example:

bash

Copy code

/api/users/{userId}

Here, {userId} is a URI parameter. When making a request, it will look something like /api/users/123, where 123 is the userId.

* **Purpose**: Used to access a specific resource or endpoint directly.

**2. Query Parameter**

* Query parameters are added to the end of a URL after a ?.
* They help filter, sort, or provide additional information to the API without changing the resource path.
* Multiple query parameters can be added, separated by &.
* Example:

bash

Copy code

/api/users?role=admin&status=active

Here, role=admin and status=active are query parameters, used to filter users with the role "admin" and status "active."

* **Purpose**: Used to refine, filter, or sort the results, not to specify the specific resource directly.

**Key Differences**

* **URI Parameters**: Part of the URL path, usually specify a resource (e.g., /api/users/123).
* **Query Parameters**: Appended to the URL after ?, used for filtering or providing extra information (e.g., /api/users?role=admin).

In WSO2, both types of parameters are common in defining RESTful APIs to enable flexible and detailed requests from clients.

**Types of URI Mapping**

In WSO2 and API development in general, **URI Mapping** is a way of defining how incoming requests (URLs) are routed to specific resources or functions in the backend. It allows the API to handle different types of requests and map them to the correct logic or data source.

1. **Direct Mapping**
   * Maps a specific URI directly to a particular resource or backend.
   * Example: /api/users always maps to a user-related function.
2. **Template Mapping (Parameterized Mapping)**
   * Uses placeholders within the URI to handle dynamic segments (variables) in the path.
   * Allows flexibility, as it can map multiple requests to a single endpoint by interpreting parameters.
   * Example: /api/users/{userId} where {userId} is a variable, allowing /api/users/123 and /api/users/456 to map to the same backend logic with different data.
3. **Wildcard Mapping**
   * Uses wildcards (like \*) to match a range of URIs with similar structures.
   * Allows mapping multiple paths without specifying each one explicitly.
   * Example: /api/\* maps all URIs that start with /api/ to a common backend or handler.

**In Short**

* **Direct Mapping**: Static, fixed paths to specific resources.
* **Template Mapping**: Dynamic paths with placeholders for variables.
* **Wildcard Mapping**: Flexible paths using wildcards to match a broader range of URIs.

These mapping types enable APIs to handle a variety of request patterns while keeping backend logic organized and accessible.

**What is LoopBack Mediators and In and Out Sequence Means**

In WSO2, the **LoopBack Mediator** is used to **send a message back to the calling sequence without routing it to any external backend**. Essentially, it allows the message flow to remain within WSO2, which can be useful for testing, debugging, or when you need to process data internally without invoking an external service.

**In Sequence and Out Sequence**

* **In Sequence**:
  + This is the sequence that **processes incoming requests** to the API.
  + It defines the mediators and logic that should be applied to the message before it’s sent to the backend or processed further.
  + Common tasks include validation, logging, transformation, or enrichment of the request.
* **Out Sequence**:
  + This sequence handles **outgoing responses** from the API, either from the backend or after processing is complete.
  + It defines the mediators and logic to apply to the response before it’s sent back to the client.
  + Common tasks in an out sequence include response transformation, logging, and adding headers.

**Summary**

* **LoopBack Mediator**: Keeps the message within WSO2 without calling an external backend.
* **In Sequence**: Processes and transforms incoming requests.
* **Out Sequence**: Processes and transforms outgoing responses.

These sequences allow WSO2 to control and manage the flow of messages through an API, both for incoming and outgoing data.

=======🡺 curl -v GET "http://localhost:8290/orders/list" -w "\n"

Here’s a step-by-step guide to creating and deploying a REST API project from scratch in **WSO2 Integration Studio** based on the setup found in the RESTAPI project.

**Step 1: Launch WSO2 Integration Studio**

1. **Open WSO2 Integration Studio** and ensure that you have a new workspace or an organized workspace structure.

**Step 2: Create a New ESB Config Project**

1. **Go to File > New > Project...**
2. **Select WSO2 > WSO2 Configuration Project**.
3. **Name the Project**: Enter a meaningful name, for example, RESTAPIConfigs.
4. **Click Finish** to create the project.

This project will hold your REST API configurations and related artifacts.

**Step 3: Create a New REST API**

1. **Right-click on the RESTAPIConfigs project**, then select **New > REST API**.
2. **Name the API**: Enter a suitable name like EcommerceAPI.
3. **Set Context Path**: In the **Context** field, specify the context path you want for the API, such as /ecommerce.
4. **Define Resources**:
   * Add **Resources** to the API configuration for specific endpoints, such as GET /products or POST /orders.
   * Each resource represents a specific URL pattern or path that this API will respond to.

**Step 4: Configure the API Resources**

1. **Add Resources to Handle Requests**:
   * For each resource, specify the **HTTP Method** (e.g., GET, POST) and **URI Template** (e.g., /products, /orders).
2. **Set Up In and Out Sequences**:
   * In the **In Sequence** section, add logic for request handling.
   * In the **Out Sequence** section, define the logic for the response.
   * **Example**: To forward a request to a backend service, add a **Send Mediator** with an **Address Endpoint** in the In Sequence.

Example configuration for GET /products:

xml

Copy code

<resource methods="GET" uri-template="/products">

<inSequence>

<send>

<endpoint>

<address uri="http://localhost:3000/products"/>

</endpoint>

</send>

</inSequence>

<outSequence>

<send/>

</outSequence>

</resource>

**Step 5: Save the API Configuration**

1. **Save the API** after adding resources and sequences.
   * This will create an XML configuration file (e.g., EcommerceAPI.xml) under src/main/synapse-config/api/ in your RESTAPIConfigs project.

**Step 6: Create a Composite Application Project**

1. **Go to File > New > Project...**
2. Select **WSO2 > Composite Application Project** and click **Next**.
3. **Name the Project**: Name it something like RESTAPICompositeExporter.
4. **Add the RESTAPIConfigs Project as a Dependency**:
   * Right-click on the composite application project (RESTAPICompositeExporter) and go to **Add or Remove Projects**.
   * Select RESTAPIConfigs and add it to the dependencies.
5. **Finish Setup**: This Composite Application Project will package your REST API configurations into a deployable .car file.

**Step 7: Build the Composite Application**

1. **Right-click on the RESTAPICompositeExporter project** and select **Run As > Maven Install**.
   * This will generate a .car (Carbon Application Archive) file in the target folder.
   * The .car file is the deployable package containing your REST API and other configurations.

**Step 8: Deploy the CAR File to WSO2 EI Server**

1. **Start your WSO2 Enterprise Integrator (EI) server**.
2. **Open the Management Console**:
   * Access the management console at https://localhost:9443/carbon.
3. **Log In**: Use the default credentials (e.g., admin/admin).
4. **Go to Carbon Applications > Add**:
   * Upload the .car file from the target directory of your composite application project.
5. **Deploy**: This will deploy the API to the WSO2 EI server.

**Step 9: Test the REST API**

1. **Use Postman or Curl** to send requests to your deployed API.
   * Example: http://localhost:8280/ecommerce/products
2. **Verify the Response** to ensure your API is working as expected.

**Summary**

1. **Create a Configuration Project** to define the API (RESTAPIConfigs).
2. **Define API and Resources** with specific endpoints and request/response handling.
3. **Create a Composite Application Project** (RESTAPICompositeExporter) to package and deploy the API.
4. **Build and Deploy the CAR file** to the WSO2 server.
5. **Test the API** with tools like Postman.

This process will help you set up and deploy a REST API in WSO2 Integration Studio from scratch.

**Swagger API**

In WSO2, **Swagger API** (also known as OpenAPI) is used to define and document RESTful APIs in a standard, machine-readable format. This helps both developers and clients understand the API structure, test endpoints, and generate client libraries. Below are key concepts related to Swagger API in WSO2:

**1. Swagger Definition (OpenAPI Spec)**

* **What it is**: A JSON or YAML file that defines the API endpoints, request/response structures, parameters, authentication, and error codes.
* **Purpose**: Provides a complete blueprint of the API, allowing developers to understand and interact with it.
* **Example**:

yaml

Copy code

openapi: 3.0.1

info:

title: Sample API

version: 1.0.0

paths:

/products:

get:

summary: Get a list of products

responses:

'200':

description: Successful response

**2. Swagger in WSO2 API Manager**

* **API Creation**: WSO2 API Manager uses Swagger definitions to create and expose APIs.
* **API Import**: You can upload a Swagger definition file to automatically generate an API, including endpoints, parameters, and request/response types.
* **API Export**: WSO2 allows you to export an API’s Swagger definition for external documentation or for use with other API management tools.

**3. Interactive Documentation**

* **Swagger UI Integration**: WSO2 API Manager integrates Swagger UI to provide interactive, visual documentation of APIs.
* **Usage**: Through the WSO2 Developer Portal, users can view API details, try out endpoints, and see real-time responses from the API.
* **Benefits**: It enhances usability by allowing users to test and explore the API without writing any code.

**4. API Testing and Mocking**

* **Testing**: With Swagger’s interactive documentation in WSO2, users can test endpoints directly within the browser. This is especially useful for developers and testers.
* **Mocking**: For testing purposes, WSO2 API Manager can generate mock responses from the Swagger definition, allowing frontend and backend teams to work in parallel.

**5. Client Code Generation**

* **Purpose**: WSO2 supports Swagger, which can be used to generate client SDKs (software development kits) in various programming languages.
* **Benefit**: This allows clients to integrate with the API quickly by using generated client libraries instead of manually coding interactions.

**6. API Lifecycle Management**

* **Versioning**: Swagger definitions make it easy to version APIs as changes in structure are documented.
* **Governance**: Swagger defines rules and specifications for requests and responses, helping ensure consistency across API versions.
* **Monitoring**: WSO2 API Manager uses the structured Swagger data to monitor and enforce policies (e.g., rate limiting, security) on the API.

**7. API Gateway Integration**

* **Routing and Policies**: WSO2 API Gateway uses Swagger definitions to understand and route API requests correctly and apply security and rate-limiting policies.
* **Simplifies Configuration**: Swagger definitions reduce the need for manual configuration as the API Manager understands the API structure and enforces policies accordingly.

**Summary**

Swagger in WSO2 provides a standardized way to define, document, test, and manage APIs. By using Swagger, WSO2 enables faster development, easier testing, and a more efficient API lifecycle management process.

**Message Entry point Thru Proxy Server**

**Understanding Proxy Service Concepts in WSO2 and the Project Structure**

**1. Proxy Service in WSO2**

* A Proxy Service in WSO2 acts as an intermediary that routes requests from clients to backend services.
* Key benefits include abstraction of backend service details from clients, logging, and enhanced security controls.
* Proxy Services support various transport protocols (like HTTP and HTTPS) and can be configured to handle different sequences for request and response processing (e.g., logging, transformations).
* First create Proxy Service 🡪 then create

**Inbound Endpoint**

* We can inject message from the transport layer directly to the mediation layer without going to Axis2 Engine
* It’s Advantage is the ability to create inbound messaging channels dynamically
* WSO2 provide two types
  1. The Listing Inbound Endpoints 🡪 that listen for a given port for the request that coming in when the request is available it’s injected to a given sequence for this case Http and Https inbound protocol are examples of this type
  2. Pulling Inbound Endpoints that pull periodically for data and when the data is available it’s injected in to a given sequence example for this is File, JMS, KAFKA inbound protocol

**What is an Inbound Endpoint?**

In WSO2, an **Inbound Endpoint** is a way for the server to "listen" for incoming messages or events from a specific source. Think of it as a special receiver that lets WSO2 Micro Integrator (MI) communicate with other systems over different protocols, not just HTTP.

**When to Use Inbound Endpoints?**

* **HTTP Inbound Endpoint**: For simple HTTP listeners without a full API setup.
* **JMS Inbound Endpoint**: When integrating with a message broker using JMS queues/topics.
* **File Inbound Endpoint**: For processing files in a folder automatically.
* **Kafka Inbound Endpoint**: For real-time event processing from a Kafka stream.

Inbound Endpoints let you easily expand the ways WSO2 can receive and process data, making it adaptable to a wide range of integration needs.

**Inbound Endpoint configuration**

The **dispatch.filter.pattern** in a WSO2 Inbound Endpoint configuration is used to specify which URL patterns the inbound endpoint should respond to. It acts as a filter, allowing the inbound endpoint to only process requests that match a certain URL path pattern.

**Example**

Suppose you configure an HTTP Inbound Endpoint like this:

xml

Copy code

<inboundEndpoint name="HTTPInboundEP" protocol="http" class="org.wso2.carbon.inbound.endpoint.protocol.http.HttpInboundEndpoint">

<parameters>

<parameter name="inbound.http.port">8081</parameter>

<parameter name="dispatch.filter.pattern">/api/\*</parameter>

</parameters>

<sequence>RequestHandlerSequence</sequence>

</inboundEndpoint>

In this example:

* The dispatch.filter.pattern is set to /api/\*.
* This means the inbound endpoint will only process HTTP requests that start with /api/ (e.g., /api/orders, /api/users).
* Requests that don't match this pattern (like /healthcheck or /status) will be ignored by this inbound endpoint.

**In Summary**

The dispatch.filter.pattern helps you control which requests get processed by the inbound endpoint, allowing you to filter requests based on their URL.

**What are Inbound Endpoints?**

An **Inbound Endpoint** in WSO2 is a configurable entry point that allows WSO2 to receive messages from various sources. It can handle different protocols, enabling WSO2 to "listen" for incoming data over HTTP, JMS, file directories, or message brokers (e.g., RabbitMQ).

**Why Use Inbound Endpoints?**

Inbound endpoints allow WSO2 to:

* Listen on custom ports for HTTP requests.
* Poll specific data sources (like a JMS queue or file directory).
* Consume events from event-based sources (like RabbitMQ or MQTT).

**Types of Inbound Endpoints and Examples**

Inbound Endpoints in WSO2 can behave in three main ways: **Listening**, **Polling**, and **Event-based**. Here’s what each type means with examples.

**1. Listening Inbound Endpoints**

* **Purpose**: Set up WSO2 to "listen" for incoming requests on a specific port.
* **Common Use Case**: HTTP or HTTPS requests.

**Example**: HTTP Inbound Endpoint

Suppose you want WSO2 to listen for HTTP requests on port **8085** instead of the standard HTTP port. You can configure an HTTP Inbound Endpoint with inbound.http.port set to **8085**.

xml

Copy code

<inboundEndpoint name="HTTPInboundEP" protocol="http" class="org.wso2.carbon.inbound.endpoint.protocol.http.HttpInboundEndpoint">

<parameters>

<parameter name="inbound.http.port">8085</parameter> <!-- Port to listen on -->

</parameters>

<sequence>RequestProcessingSequence</sequence>

</inboundEndpoint>

In this case:

* Clients can send HTTP requests directly to http://your-server:8085.
* WSO2 will forward those requests to RequestProcessingSequence, which can process the message and trigger the desired service.

**Use Case**: Ideal when you need to expose a service over HTTP without creating a full API, or when you want to listen on a specific port.

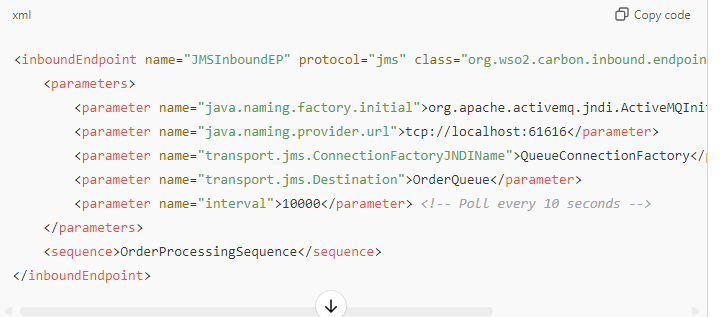
**2. Polling Inbound Endpoints**

* **Purpose**: Set up WSO2 to "poll" a data source at intervals and pick up new messages or data.
* **Common Use Case**: JMS queues, file directories.

**Example**: JMS Inbound Endpoint

Imagine you have a **JMS queue** (e.g., OrderQueue) where new order messages are placed. You want WSO2 to poll this queue every 10 seconds, retrieve any new messages, and process them.

In this example:



* WSO2 polls OrderQueue every 10 seconds.
* For each message in the queue, it triggers the OrderProcessingSequence for processing.

**Use Case**: Useful for message-driven integrations where messages are queued for processing, like JMS or file-based systems.

**3. Event-Based Inbound Endpoints**

* **Purpose**: Establish a connection to an event source once and then "subscribe" to receive events as they arrive.
* **Common Use Case**: Event brokers like RabbitMQ or MQTT.

**Example**: MQTT Inbound Endpoint

Imagine you have an **MQTT broker** sending IoT device data to a topic named DeviceStatus. You want WSO2 to connect to this broker, subscribe to DeviceStatus, and process events as they come in.



In this example:

* WSO2 establishes a connection to the MQTT broker on mqtt-broker.local:1883.
* It subscribes to the DeviceStatus topic.
* Every time a message is published to this topic, WSO2 triggers DeviceStatusProcessingSequence.

**Use Case**: Suitable for event-driven applications where real-time data needs to be processed immediately upon arrival, like IoT or sensor data streams.

**Summary of the Different Types**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Behavior | Example Protocols | Example Use Case |
| Listening | Listens on a specific port | HTTP, HTTPS | Expose a service on a custom HTTP port |
| Polling | Polls at intervals | JMS, File | Poll a JMS queue every X seconds to process new messages |
| Event-Based | Subscribes to events | RabbitMQ, MQTT | Subscribe to an MQTT topic to process messages as they arrive |

**Additional Notes**

1. **Dynamic Configuration**: Inbound Endpoints allow for dynamic configuration, meaning you can easily change the settings (like polling intervals or ports) without redeploying the service.
2. **Custom Inbound Endpoints**: For advanced needs, you can create custom inbound endpoints by extending the WSO2 classes (GenericInboundListener for listening or GenericPollingConsumer for polling).

**In Summary**

* **Inbound Endpoints** allow WSO2 to receive messages from different sources and protocols.
* They provide flexibility by supporting **Listening**, **Polling**, and **Event-Based** modes.
* This makes WSO2 capable of handling diverse integration patterns, such as HTTP services, JMS messaging, file polling, and real-time event processing.

By using the right type of inbound endpoint, you can easily tailor WSO2 to your specific integration needs.

First **Create Integration Project** 🡪 **Create Endpoint 🡪 Create A New API  🡪 and we have the backend service running and the endpoint we have created directly calling this backend running service and we are going to call this backend using Endpoint and in the API we creating we call the endpoint in the api and thru API we will call the Backend Service**

**When we are working in Inbound Endpoint refer this link** [Use Inbound Endpoints - WSO2 API Manager Documentation 4.2.0](https://apim.docs.wso2.com/en/4.2.0/tutorials/integration-tutorials/using-inbound-endpoints/)

Now we will create Inbound Endpoint In our previous Integration project and expose our Rest API using different port

**Scheduled Tasks**

Scheduled tasks in the Micro Integrator are tools that allow you to automate certain actions by setting them to occur at specific intervals. This is particularly useful when you need to perform repetitive jobs, like data extraction, processing, or sending automated messages. Here’s a simple breakdown and a real-world example to make it easier to understand.

**How Scheduled Tasks Work**

1. **Configure Tasks**: You set up scheduled tasks to execute either internal or external commands. These can be configured to send a message to specific parts of your system, such as a proxy service, main sequence, or any other named sequence.
2. **Set Timing Intervals**: Tasks can be set to run at fixed intervals (e.g., every 10 minutes) or at specific times using cron syntax (e.g., every day at midnight).
3. **Custom Tasks**: Beyond the default functions, you can create custom tasks by implementing the Task interface in a Java class. This flexibility lets you create tasks that are unique to your system’s requirements.

**Simple Real-World Example**

Suppose you manage an online store and want to automate a nightly job that checks the day’s sales, processes the data, and updates a sales report for the day. Here’s how you’d use a scheduled task for this:

1. **Data Extraction**: Set a task to run every night at midnight. The task connects to your store’s sales database and pulls the sales data from that day.
2. **Data Transformation**: Once the data is extracted, it’s processed (e.g., cleaned, summarized, or enriched with additional details like product category totals).
3. **Send to Service**: Finally, the task sends the processed data to a reporting service or a master data repository, where it’s stored or further analyzed.

**Example in Action**

Let’s say your task is scheduled to read a file containing stock order details every morning. The task:

* **Reads** the file for new orders.
* **Processes** the order data, such as verifying stock symbols and quantities.
* **Places** orders based on the file’s contents to a stock trading platform.

This setup automates the daily routine, saving time and ensuring consistency.