API:

It provides info for how to communicate with a s/w component, defining the

i)operation, what to call,

ii)i/ps(what to send)

iii)o/ps(what u get back)

VM is Mule's internal transport for messaging/queueing. The VM transport is for intra-JVM communication between Mule flows. So, that means when you use a VM in your flow, you can communicate between different flows in the application. It can only be used by Mule applications. When creaing a VM queue it can only be accessed by the Mule application that creates it (Cloudhub for example) OR it can be reused by same Mule apps running in a domain project or cluster. No existing broker infrastructure needs to be setup. Supports persistent and transient.

Anypoint MQ is Mulesoft's Cloud Messaging platform. This can be used by other applications - not just Mule. It can also be used across multiple Mule apps regardless of domains or cluster, well suited for Cloudhub applications. No infra setup, all in the cloud. Think Amazon SQS but a lot better and great integration with Anypoint Platform

JMS uses the Java Messaging Service protocol and requires an external JMS broker such as ActiveMQ. Can be used by any application that supports JMS connectivity.

**For reliable we use JMS, there isno chance of losing data.**

**VM –adv queue**

**JMs- more robust**

API Autodiscovery is a mechanism that manages an API from API Manager by pairing the deployed application to an API created on the platform. API Management includes tracking, enforcing policies if you apply any, and reporting API analytics. Critical to the Autodiscovery process is identifying the API by providing the API name and version.

Map : Returns an array that is the result of applying a transformation function (lambda) to each of the elements. ([https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators) map)

2. MapObject: Similar to Map, but instead of processing only the values of an object, it processes both keys and values as a tuple. (

### Using Map to Return an Array

(':array', ':function') ⇒ :array

Returns an array that is the result of applying a transformation function (lambda) to each of the elements. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map upper $

Output

{

"users": [

"JOHN",

"PETER",

"MATT"

]

}

In the following example, custom names are defined for the index and value parameters of the map operation, and then both are used to construct the returned value. In this case, value is defined as **firstName** and its index in the array is defined as **position**.

Transform

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map ((firstName, position) -> position ++ ":" ++ upper firstName)

Output

{

"users": [

"0:JOHN",

"1:PETER",

"2:MATT"

]

}

### Using Map on an Object

(':object', ':function') ⇒ ':array'

Returns an array with the values that result out of applying a transformation function (lambda) to each of the values in the object. The keys of the original object are all ignored by this operation and the object is treated as an array. To have access to the keys, you can use the operation **mapObject** instead. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**. The index refers to the position of a key:value pair when the object is treated as an array.

|  |
| --- |
| See [Map Object](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#map-object) if what you want is to process both keys and values instead of just values. |

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map (

'$$':{

dollars: $,

localCurrency: $ \* conversionRate

}

)

### Input: XML

Input

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

### Output: JSON

Output

{

"priceList": [

{

"0": {

"dollars": "9.99",

"localCurrency": 134.3655

}

},

{

"1": {

"dollars": "53",

"localCurrency": 712.85

}

},

{

"2": {

"dollars": "398.99",

"localCurrency": 5366.4155

}

}

]

}

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of in this example, you must either enclose it in quote marks or brackets. '' or ($$) are both equally valid. |

In the example above, as key and value are not defined, they’re identified by the placeholders **$$** and **$**. For each key:value pair in the input, an object is created and placed in an array of objects. Each of these objects contains two properties: one of these directly uses the value, the other multiplies this value by a constant that is defined as a directive in the header.

The mapping below performs exactly the same transform, but it defines custom names for the properties of the operation, instead of using $ and $$. Here, position is defined as referring to the array index, and money to the value in that index. The reference to the array index (named position, in this case) is optional.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map ((money, position) ->

'$position':{

dollars: money,

localCurrency: money \* conversionRate

}

)

The reference to the array index is optional. This is also a valid example:

.Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map ((money) ->

{

dollars: money,

localCurrency: money \* conversionRate

}

)

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of position in this example, you must either enclose it in brackets or enclose it in quote marks adding a $ to it, otherwise the name of the property is taken as a literal string. '$position' or (position) are both equally valid. |

## Map Object

(':object', ':function') ⇒ ':object'

Similar to Map, but instead of processing only the values of an object, it processes both keys and values as a tuple. Also instead of returning an array with the results of processing these values through the lambda, it returns an object, which consists of a list of the key:value pairs that result from processing both key and value of the object through the lambda.

The lambda is invoked with two parameters: **key** and the **value**. If these parameters are not named, the key is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices mapObject (

'$$':{

dollars: $,

localCurrency: $ \* conversionRate

}

)

### Input: XML

Input

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

### Output: JSON

Output

{

"priceList": {

"basic": {

"dollars": "9.99",

"localCurrency": 134.3655

},

"premium": {

"dollars": "53",

"localCurrency": 712.85

},

"vip": {

"dollars": "398.99",

"localCurrency": 5366.4155

}

}

}

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of in this example, you must either enclose it in quote marks or brackets. '' or ($$) are both equally valid. |

In the example above, as key and value are not defined, they’re identified by the placeholders **$$** and **$**. For each key:value pair in the input, the key is preserved and the value becomes an object with two properties: one of these is the original value, the other is the result of multiplying this value by a constant that is defined as a directive in the header.

The mapping below performs exactly the same transform, but it defines custom names for the properties of the operation, instead of using $ and $$. Here, 'category' is defined as referring to the original key in the object, and 'money' to the value in that key.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices mapObject ((money, category) ->

'$category':{

dollars: money,

localCurrency: money \* conversionRate

}

)

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of **category** in this example, you must either enclose it in brackets or enclose it in quote marks adding a $ to it, otherwise the name of the property is taken as a literal string. '$category' or (category) are both equally valid. |

# About Design Center

Anypoint Design Center is a development environment that consists of two tools:

* [API Designer](https://docs.mulesoft.com/design-center/design-create-publish-api-specs)

In API Designer, you can create API specifications in RESTful API Modeling Language (RAML) 0.8 or 1.0, or according to OpenAPI specification (OAS) 2.0 or 3.0. You can also create API fragments in RAML. After you create a specification or fragment, you can publish it to Anypoint Exchange, so that it can be used by anyone in your MuleSoft organization.

* [Flow Designer](https://docs.mulesoft.com/design-center/about-designing-a-mule-application)

In Flow Designer, you create Mule applications to integrate systems into workflows.

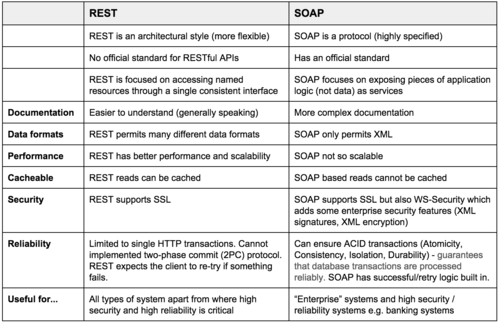
# Flow Designer

In Flow Designer, you create Mule applications to integrate systems into workflows. You create them as projects that have one or more flows. A project contains the flows for one Mule application. A flow consists of a sequence of cards, with each card representing a core component, connector, module, or API. Each card receives input data, carries out a specific task that makes use of the input data, and then sends the output data from that task to the next card in the flow. The final card in a flow is usually one that sends a notification that the Mule application has completed its task or that sends data.

To start creating Mule applications:

* Try the easiest option, which is to import a template into Flow Designer from Anypoint Exchange. See [Create a Mule Application Quickly from a Template](https://docs.mulesoft.com/design-center/import-template).
* Create a new Mule-app project in Design Center and use the wizard that automatically opens to walk you through scaffolding a flow with a trigger card and a target card.
* Create a new Mule-app project and click **Go straight to the canvas** in the wizard to start building a flow from a blank canvas.

## WebServices > SOAP VS REST and JSON VS XML: WebServices : SOAP vs REST & JSON vs XML



## Soap ws are more complex, restful r simpler type

## What is WSDL?

**Web Services Description Language** (WSDL) is an XML-based file that basically tells the client application what the web service does. The WSDL file is used to describe in a nutshell what the web service does and gives the client all the information required to connect to the web service and use all the functionality provided by the web service.

A WSDL document is used to describe a web service. This description is required, so that client applications are able to understand what the web service actually does.

* The WSDL file contains the location of the web service and
* The methods which are exposed by the web service.

The WSDL file itself can look very complex to any user, but it contains all the necessary information that any client application would require to use the relevant web service.

Below is the general structure of a WSDL file

* Definition
* TargetNamespace
* DataTypes
* Messages
* Porttype
* Bindings
* service

One key thing to note here is that definition of messages, which is what is passed by the SOAP protocol is actually defined in the WSDL document.

The WSDL document actually tells a client application what are the types of SOAP messages which are sent and accepted by the Web service.

In other words, the WSDL is just like a postcard which has the address of a particular location. The address provides the details of the person who delivered the postcard. Hence, in the same way, the WSDL file is the postcard, which has the address of the web service which can deliver all the functionality that the client wants.

<!-- WSDL definition structure -->

<definitions

name="Guru99Service"

targetNamespace=http://example.org/math/

xmlns=http://schemas.xmlsoap.org/wsdl/>

<!-- abstract definitions -->

<types> ...

<message> ...

<portType> ...

<!-- concrete definitions -->

<binding> ...

<service> ...

</definition>

1. The **<types>** tag is used to define all the complex datatypes, which will be used in the message exchanged between the client application and the web service. For example, there could be a data type called EmployeeDataType which could have 2 elements called "EmployeeName" of type string and "EmployeeID" of type number or integer. Together they form a data structure which then becomes a complex data type.
2. The **<messages>** tag is used to define the message which is exchanged between the client application and the web server. These messages will explain the input and output operations which can be performed by the web service. An example of a message can be a message which accepts the EmployeeID of an employee, and the output message can be the name of the employee based on the EmpoyeeID provided.
3. The **<portType>** tag is used to encapsulate every input and output message into one logical operation. So there could be an operation called "GetEmployee" which combines the input message of accepting the EmployeeID from a client application and then sending the EmployeeName as the output message.
4. The **<binding>** tag is used to bind the operation to the particular port type. This is so that when the client application calls the relevant port type, it will then be able to access the operations which are bound to this port type. Port types are just like interfaces. So if a client application needs to use a web service they need to use the binding information to ensure that they can connect to the interface provided by that web service.
5. The **<service>**tag is a name given to the web service itself. Initially, when a client application makes a call to the web service, it will do by calling the name of the web service. For example, a web service can be located at an address such as **http://localhost/Guru99/Tutorial.asmx** . The service tag will actually have the URL defined as **http://localhost/Guru99/Tutorial.asmx**, which will actually tell the client application that there is a web service available at this location.

## Why WSDL

A web service is an important component in building modern day web applications. Their main purpose is to allow multiple applications built on various programming languages to talk to each other. For instance, we can have a .Net web application talks to a[Java](https://www.guru99.com/java-tutorial.html)application via a Web service.

A web service has the following key features

* It is built using the XML programming language. Almost all modern day technologies such as .Net and Java have corresponding commands that have the ability to work with XML. Hence, XML was taken as the most appropriate language for building web services.
* Web services communicate over HTTP. HTTP is a protocol used by all web-based applications. Hence, it just made sense to ensure that Web services also had the ability to work over the HTTP protocol.
* Web services conform to a particular language specification. This specification is set by the W3C, which is the governing body for all web standards.
* Web services have a description language known as WSDL, which is used to describe the web service.

The WSDL file is written in plain old XML. The reason that it is in XML is so that the file can be read by any programming language.

## **What is HTTP?**

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.

HTTP works as a request-response protocol between a client and server.

Example: A client (browser) sends an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.

## **The GET Method**

**GET is used to request data from a specified resource.**

**GET is one of the most common HTTP methods.**

Note that the query string (name/value pairs) is sent in the URL of a GET request:

/test/demo\_form.php?name1=value1&name2=value2

**Some other notes on GET requests:**

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests are only used to request data (not modify)

## **The POST Method**

**POST is used to send data to a server to create/update a resource.**

The data sent to the server with POST is stored in the request body of the HTTP request:

**Some other notes on POST requests:**

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

## **The PUT Method**

**PUT is used to send data to a server to create/update a resource.**

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

## **The HEAD Method**

**HEAD is almost identical to GET, but without the response body.**

In other words, if GET /users returns a list of users, then HEAD /users will make the same request but will not return the list of users.

HEAD requests are useful for checking what a GET request will return before actually making a GET request - like before downloading a large file or response body.

**The DELETE method deletes the specified resource.**

## **The OPTIONS Method**

**The OPTIONS method describes the communication options for the target resource.**

## **Compare GET vs. POST**

The following table compares the two HTTP methods: GET and POST.

|  |  |  |
| --- | --- | --- |
|  | **GET** | **POST** |
| BACK button/Reload | Harmless | Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted) |
| Bookmarked | Can be bookmarked | Cannot be bookmarked |
| Cached | Can be cached | Not cached |
| Encoding type | application/x-www-form-urlencoded | application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data |
| History | Parameters remain in browser history | Parameters are not saved in browser history |
| Restrictions on data length | Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters) | No restrictions |
| Restrictions on data type | Only ASCII characters allowed | No restrictions. Binary data is also allowed |
| Security | GET is less secure compared to POST because data sent is part of the URL  Never use GET when sending passwords or other sensitive information! | POST is a little safer than GET because the parameters are not stored in browser history or in web server logs |
| Visibility | Data is visible to everyone in the URL | Data is not displayed in the URL |

# [Status codes](https://www.w3.org/Protocols/HTTP/HTTP2.html)

The values of the numeric status code to [HTTP](https://www.w3.org/Protocols/HTTP/HTTP2.html) requests are as follows. The data sections of messages Error, Forward and redirection responses may be used to contain human-readable diagnostic information.

## Success 2xx

These codes indicate success. The body section if present is the object returned by the request. It is a MIME format object. It is in MIME format, and may only be in text/plain, text/html or one fo the formats specified as acceptable in the request.

### OK 200

The request was fulfilled.

### CREATED 201

Following a POST command, this indicates success, but the textual part of the response line indicates the URI by which the newly created document should be known.

## 304-not modified, nothing was modified by reqest

401-unauthorized

Does not have access to resource

404-resource not found, uri is not recognized by the server

500-server error

## Error 4xx, 5xx

The 4xx codes are intended for cases in which the client seems to have erred, and the 5xx codes for the cases in which the server is aware that the server has erred. It is impossible to distinguish these cases in general, so the difference is only informational.

The body section may contain a document describing the error in human readable form. The document is in [MIME](https://www.w3.org/Protocols/HTTP/References.html" \l "z1) format, and may only be in text/plain, text/html or one for the formats specified as acceptable in the request.

### Bad request 400

The request had bad syntax or was inherently impossible to be satisfied.

### Unauthorized 401

The parameter to this message gives a specification of authorization schemes which are acceptable. The client should retry the request with a suitable [Authorization](https://www.w3.org/Protocols/HTTP/HTRQ_Headers.html" \l "z9) header.

### PaymentRequired 402

The parameter to this message gives a specification of charging schemes acceptable. The client may retry the request with a suitable ChargeTo header.

### Forbidden 403

The request is for something forbidden. Authorization will not help.

### Not found 404

The server has not found anything matching the URI given

### Internal Error 500

The server encountered an unexpected condition which prevented it from fulfilling the request.

**Define Mule ESB**

It is an integration platform and a lightweight enterprise service bus (ESB) based on Java. It enables developers to easily and quickly connect applications and exchange data. With Mule ESB, developers can easily integrate existing systems irrespective of how different are the technologies that the application use such as Web Services, HTTP, JDBC, JMS and the like.

**What is the benefit of using Mule ESB?**

Mule ESB is an integration framework that is lightweight and highly scalable. It enables developers in starting small applications and also in connecting different applications. With Mule managing the exchanges between components and applications transparently and ESB taking care of a variety of applications, it is easy to integrate third-party applications with the help of Mule.

**What are the various types of Exception Handling?**

* Global Exception Handling
* Catch Exception Handling
* Choice Exception Handling
* Default Exception Handling
* Rollback Exception Handling

**What are the characteristics of Mule ESB?**

An ESB is used for the purpose of integration with an approach that is service-oriented. Its features include:

* Message Routing Service
* Message Transformation Service
* Set of Service Container
* Web Service Security

**In Mule, how do you develop and consume SOAP services?**

SOAP services can be created just like how we create a Mule project by using RAML. The difference here is that we need Concert WSDL importing rather than RAML. And SOAP services can be consumed by using our Mule flow CXF component or Web Service Consumer.

**How can you find out whether your project requires ESB?**

As every project might not require an ESB, you should analyze first to see if your project might benefit from ESB implementation. Certain things that should be at the front of your mind while you analyze the need for ESB are:

* If the project requires integration of more than 3 applications or services and if communication between two application is needed, it would be enough to use point to point integration
* Sometimes there will be a need for you to scale the project in the future where there might arise a need to interact with multiple services. This is required only by a few projects that perform heavy tasks
* If the project requires message routing abilities such as aggregating and forking message flows. This feature is not necessary for all projects
* You should have clarity on the architecture of the thing that needs to be achieved. A simple POC integration of small parts to find out the benefits is much better
* As most of the ESBs are on the expensive side, first evaluate whether your project budget permits ESB use

**Name the various kinds of Primitives that are used in Mediation.**

The following are the various kinds of primitives in mediation

* Endpoint Lookup
* Service Invoke
* DB lookup
* Data Handler
* Type Filter
* Message Element Setter
* Custom MediationFan-out
* Fan-in
* Header Setters
* Message Logger
* Even Emitter
* XSLT
* BO MapMessage Filter
* Fail
* Stop
* Sub Flow

**Name the various ESBs that are in the market**

There are different ESBs in the market, both licensed and open source. They are:

* JBoss Fuse ESB
* Mule ESB
* Talend

Microservices:

The central idea behind microservices is that some types of applications become easier to build and maintain when they are broken down into smaller, composable pieces which work together. Each component is continuously developed and separately maintained, and the application is then simply the sum of its constituent components. This is in contrast to a traditional, "monolithic" application which is all developed all in one piece.

Applications built as a set of modular components are easier to understand, easier to test, and most importantly easier to maintain over the life of the application. It enables organizations to achieve much higher agility and be able to vastly improve the time it takes to get working improvements to production. This approach has proven to be superior, especially for large enterprise applications which are developed by teams of geographically and culturally diverse developers.

There are other benefits:

* **Developer independence**: Small teams work in parallel and can iterate faster than large teams.
* **Isolation and resilience**: If a component dies, you spin up another while and the rest of the application continues to function.
* **Scalability**: Smaller components take up fewer resources and can be scaled to meet increasing demand of that component only.
* **Lifecycle automation**: Individual components are easier to fit into continuous delivery pipelines and complex deployment scenarios not possible with monoliths.
* **Relationship to the business**: Microservice architectures are split along business domain boundaries, increasing independence and understanding across the organization.

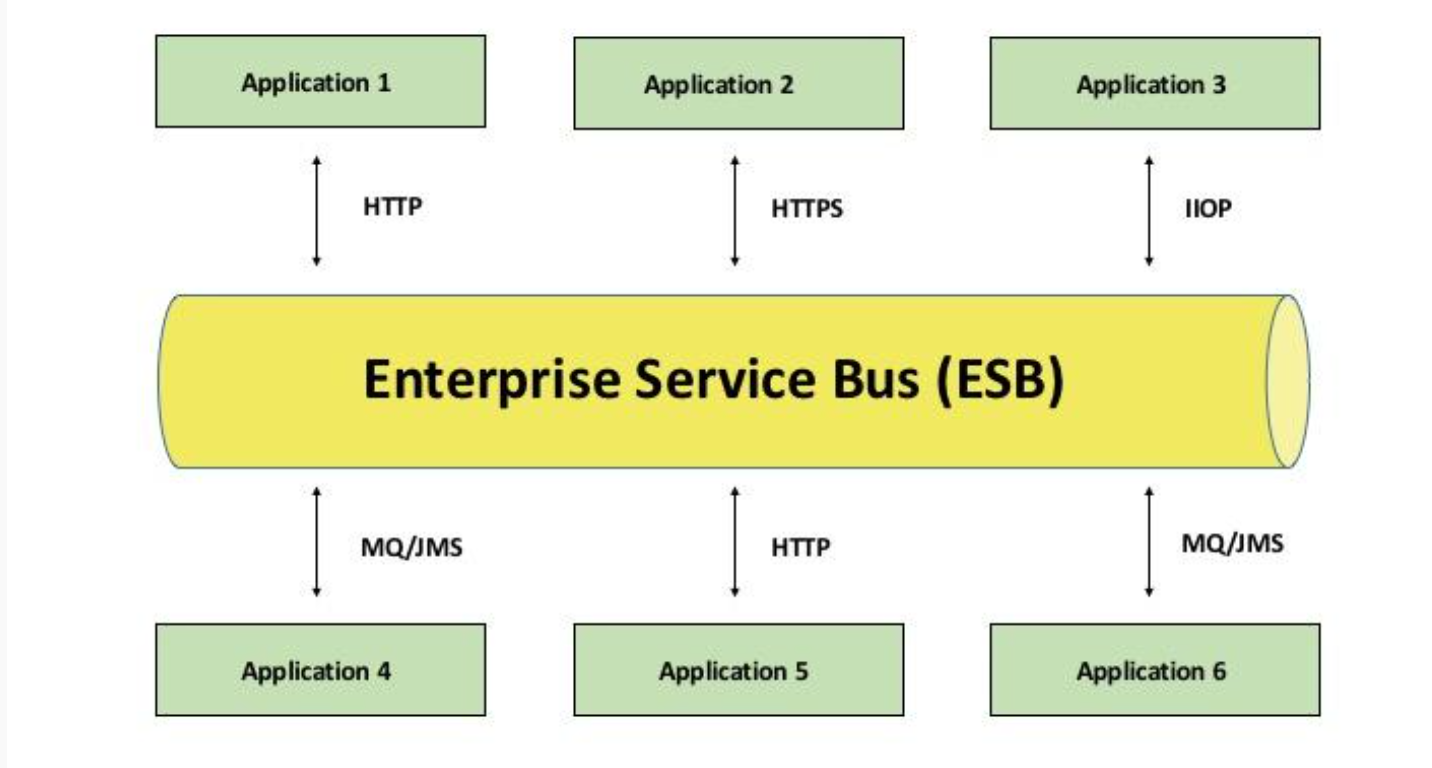
The common definition of microservices generally relies upon each microservice providing an API endpoint, often but not always a stateless REST API which can be accessed over HTTP(S) just like a standard web page. This method for accessing microservices make them easy for developers to consume as they only require tools and methods many developers are already familiar with.

## **What is ESB:**

An ESB provides a secure, scalable and cost-effective infrastructure that enables real-time data exchange among many systems.

An ESB can provide

Messaging, Web Services , Data Transformation, Routing



## An Enterprise Service Bus (ESB) is a

## 1. Middleware software platform

## 2. It used Application to Application communication.

## Why ESB:

## Lets Consider that there are two systems that need to exchange data.

## Team plan and implement a solution (point to point) that allows these systems to communicate.

## Over couple of years, team deploys several more systems that need to interact with each other as well as the existing two systems.

## 

## **What is Middleware?**

Middleware is the software that connects software components or enterprise applications. Middleware is the software layer that lies between the operating system and the applications on each side of a distributed computer network. **The function of middleware is to mediate interaction between the parts of an application, or between applications.**

Middleware is the infrastructure which facilitates creation of business applications, and provides core services like concurrency, transactions, threading, messaging, and the SCA framework for service-oriented architecture (SOA) applications. Middleware includes Web servers, application servers, content management systems, and similar tools that support application development and delivery.

We can write functions or create variables in dataweave to perform something complex or store values and reuse them later by calling them anywhere in dataweave body.

## **Service-Oriented Architecture**

SOA :

Service-oriented architecture is essentially a collection of services.

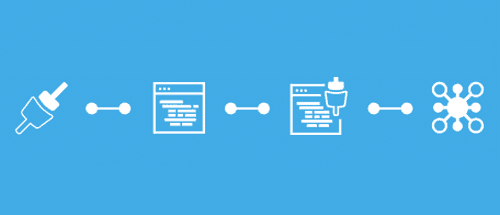
These services communicate with each other.

Services can range in size anywhere from small application services to very large enterprise services.



Service-Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among diverse interacting software applications, enabling organizations to take advantage of existing investments in applications and systems. SOA facilitates the development of modular business services that can be easily integrated and reused, thus creating a flexible and adaptable infrastructure. Using a SOA approach, an organization can focus more resources and budget on innovation and on delivering new business services. Systems that can successfully use SOA can minimize the disruption of planned or unplanned outages in an enterprise.

Some of the advantages of using SOA are:

* **Reduction in development time and cost**: SOA services are easily reused and can be rapidly assembled into new, composite applications.
* **Lower maintenance cost:** Reusable services reduce the number and internal complexity of enterprise services.
* **High-quality services**: Increased service reuse creates high-quality services through multiple testing cycles from different service consumers.
* **Lower integration costs**: Standardized services know how to work together, enabling disparate applications to quickly and easily connect.
* **Reduce risk**: Fewer, reusable services provide greater control over corporate and IT governance policies, and reduce the overall compliance risk to an enterprise.
* 

[API-led connectivity](https://www.mulesoft.com/lp/whitepaper/api/api-led-connectivity)

Mulesoft's methodical way of connecting data and applications through reusable APIs to decouple between the implementation and the API

Three layers

* Experience Layer
* Process Layer
* System Layer

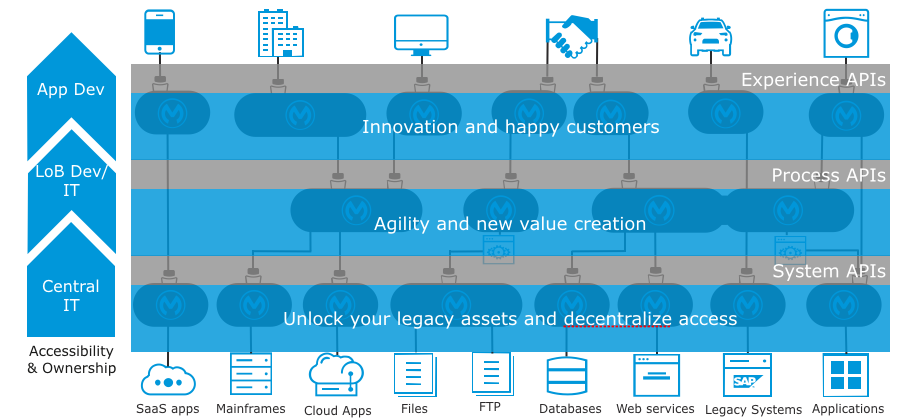
It is a methodical way to connect data to applications through reusable and purposeful [APIs](https://www.mulesoft.com/resources/api/what-is-an-api). **API-led connectivity that goes far beyond software integration.** The system is made up of three different layers of APIs with a single purpose – to make your business composable & agile:

* Governed by IT, **system APIs** are those APIs which extract sensitive information from your systems.
* **Process APIs** are designed specifically for processes in an organisation. For example, if you need to fetch your bank balance from a smartphone or laptop, the process is the same.
* **Experience APIs** would be developed according to how the information is displayed on any particular device.

This methodology allows organisations to easily **add more and more devices and solutions into the mix, while maintaining high performance of the whole system.** Instead of taking months to do so, it can be done within a matter of days or weeks because the processes are already there. This applies to the other end of the spectrum, even after merge or acquisition, systems, processes & devices of  the organisations can be interconnected through the APIs.

[**Read further: DRIVE YOUR BUSINESS SPEED OF CHANGE WITH API**](https://www.ricston.com/blog/drive-business-speed-change-apis/)

With API-led Connectivity, your IT infrastructure would look more like the below, creating a composable, transparent and decentralized structure. This allows different lines of business to take more control over their systems, while system data is still governed by the IT department.



No matter how many years your organisation has been around & how old your systems are, things don’t have to take an eternity to happen anymore. You can beat competition by doing things first, by transforming a business idea through technology within days, weeks instead of months, years.

Functions

Dataweave functions have to be defined with %function in the header part

%dw 1.0

%output application/json

%function getName() {

name: payload.user.name

}

---

getName()

With the introduction of datawaeve version 2 in Mule 4, the syntax for declaring functions will change.

Variables

Dataweave variables have to be defined with %var in the header part

%dw 1.0

%output application/json

%var name = payload.user.name

---

name: name

Lets have the below XML as input to both the scripts.

<user>

<name>Jane</name>

</user>

Output of both scripts for the above XML is same.

{

"name": "Jane"

}

**Serialization** is a mechanism of converting the **state of an object into a byte stream.** **Deserialization** is the reverse process where the byte stream is used to recreate the actual **Java** object in memory.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

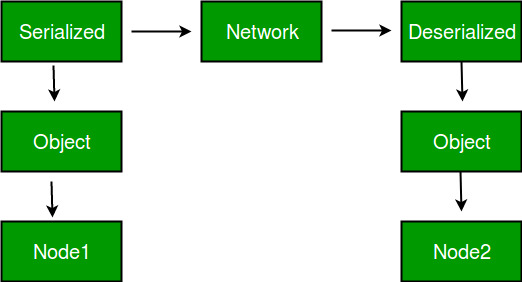
public final Object readObject()

throws IOException,

ClassNotFoundException

**Advantages of Serialization**

1. To save/persist state of an object.  
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface.

**Points to remember**  
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.  
Example :

class A implements Serializable{

// B also implements Serializable

// interface.

B ob=new B();

}

**Example 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code for serialization and deserialization  // of a Java object  import java.io.\*;    class Demo implements java.io.Serializable  {      public int a;      public String b;        // Default constructor      public Demo(int a, String b)      {          this.a = a;          this.b = b;      }    }    class Test  {      public static void main(String[] args)      {          Demo object = new Demo(1, "geeksforgeeks");          String filename = "file.ser";            // Serialization          try          {              //Saving of object in a file              FileOutputStream file = new FileOutputStream(filename);              ObjectOutputStream out = new ObjectOutputStream(file);                // Method for serialization of object              out.writeObject(object);                out.close();              file.close();                System.out.println("Object has been serialized");            }            catch(IOException ex)          {              System.out.println("IOException is caught");          }              Demo object1 = null;            // Deserialization          try          {              // Reading the object from a file              FileInputStream file = new FileInputStream(filename);              ObjectInputStream in = new ObjectInputStream(file);                // Method for deserialization of object              object1 = (Demo)in.readObject();                in.close();              file.close();                System.out.println("Object has been deserialized ");              System.out.println("a = " + object1.a);              System.out.println("b = " + object1.b);          }            catch(IOException ex)          {              System.out.println("IOException is caught");          }            catch(ClassNotFoundException ex)          {              System.out.println("ClassNotFoundException is caught");          }        }  } |

Output :

Object has been serialized

Object has been deserialized

a = 1

b = geeksforgeeks

**Example 2:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code for serialization and deserialization  // of a Java object  import java.io.\*;    class Emp implements Serializable {  private static final long serialversionUID =                                   129348938L;      transient int a;      static int b;      String name;      int age;        // Default constructor  public Emp(String name, int age, int a, int b)      {          this.name = name;          this.age = age;          this.a = a;          this.b = b;      }    }    public class SerialExample {  public static void printdata(Emp object1)      {            System.out.println("name = " + object1.name);          System.out.println("age = " + object1.age);          System.out.println("a = " + object1.a);          System.out.println("b = " + object1.b);      }    public static void main(String[] args)      {          Emp object = new Emp("ab", 20, 2, 1000);          String filename = "shubham.txt";            // Serialization          try {                // Saving of object in a file              FileOutputStream file = new FileOutputStream                                             (filename);              ObjectOutputStream out = new ObjectOutputStream                                             (file);                // Method for serialization of object              out.writeObject(object);                out.close();              file.close();                System.out.println("Object has been serialized\n"                                + "Data before Deserialization.");              printdata(object);                // value of static variable changed              object.b = 2000;          }            catch (IOException ex) {              System.out.println("IOException is caught");          }            object = null;            // Deserialization          try {                // Reading the object from a file              FileInputStream file = new FileInputStream                                           (filename);              ObjectInputStream in = new ObjectInputStream                                           (file);                // Method for deserialization of object              object = (Emp)in.readObject();                in.close();              file.close();              System.out.println("Object has been deserialized\n"                                  + "Data after Deserialization.");              printdata(object);                // System.out.println("z = " + object1.z);          }            catch (IOException ex) {              System.out.println("IOException is caught");          }            catch (ClassNotFoundException ex) {              System.out.println("ClassNotFoundException" +                                  " is caught");          }      }  } |

Output:

Object has been serialized

Data before Deserialization.

name = ab

age = 20

a = 2

b = 1000

Object has been deserialized

Data after Deserialization.

name = ab

age = 20

a = 0

b = 2000

Description for Output:  
You have seen while deserializing the object the values of a and b has changed. The reason being a was marked as transient and b was static.  
In case of **transient variables:-** A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).  
In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization