# How to Create Shared Components

## Vendor-agnostic Shared Component Creation

Shared API components are generally created to externalize and package shared reusable processes and routines across more than one API. For example, a reusable validation routine with complex business rules can be packaged into a shared component to be used by multiple APIs that rely on the same business rules and validation. In addition, to enable standard API response messaging, it is recommended to standardize exception and validation messages across all APIs for a given application. Error messages can be packaged into a shared component and for all API developers of an application to embed this component into their API projects. That way, whenever an API exception or validation error is raised within any API of the same application, a standard message is returned to the API consumer.

There are common best practices to create shared components, package them and integrate as dependent modules or libraries into API development projects. In Java a shared component is a .jar dependency library while in Microsoft a .NET assembly. Both language platforms have a similar approach to packaging and integrating shared components.

In Java, using Maven, Gradle or Apache Ant, a shared component is packaged into a .jar file. That .jar file is then included as a dependency into any project that requires it. Maven, Gradle or Ant have specific instructions on how to add a shared component into a project. For example, with Maven, it is simply listing the shared component as a dependency in the pom.xml file.

In Microsoft.NET, packaging a project as an assembly and then adding that assembly file via NuGet or directly as a dependency reference in Visual Studio.NET is the best practice approach.

In NodeJS projects, the best practice is to package reusable shared components as modules, with package.json file listing the module name and version. That module can then be deployed into a dependency management platform (public or private NodeJS module library) and integrated into any NodeJS microservice project as a dependency.

To properly manage shared components across projects, there is a need for a dependency management platform. As a best practice, it is a function of the organization’s IT department to maintain a private dependency management repository of all approved and certified third-party and in-house-developed dependencies and components. There are various dependency management platforms on the market, but commonly used are JFrog Artifactory or Maven private repository for Java and NodeJS projects. For Microsoft .NET projects, Team Foundation Server provides the ability to manage shared components and integrate them into any Visual Studio.NET project via NuGet or similar package management add-ons.

## MuleSoft-specific Shared Component Creation

### Creating Shared Components for Mule

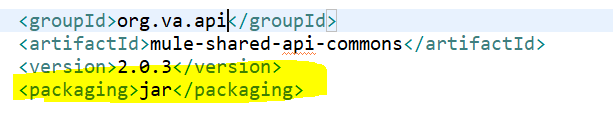
#### Use Case

There are several needs for shared components that arise when developing Mule applications. A typical use case is to package those flows and sub-flows that are reusable across multiple Mule process applications into a shared component. Also, with Mule 4.x, reusable error handlers with standardized exception messages can and should also be packaged into a shared library or component. Finally, with the new Mule 4.x DataWeave 2.0, Mulesoft provides a capability to create reusable DataWeave functions that can be moved into a .dw file. These shared reusable transformation functions can also be packaged into a shared component.

#### Shared API Commons Library

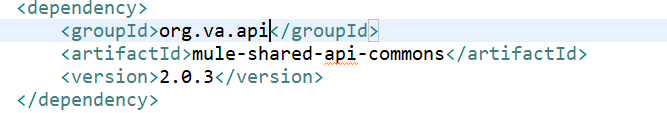
Mule with version 3.x and, now, with version 4.x provides the ability to package shared reusable flows, functions, and error handlers into a separate Mule application. However, instead of instructing Mule to package this application as a standard runtime application (in a form of a .zip file), there is a need to update the Maven pom.xml file to instruct Mule to package this application as a .jar file. A JAR is a Java Archive file which can then be added to any Mule project as a dependency, similarly to how any third-party external dependency is added in a Maven pom.xml file.

Therefore, the first step is to update the shared library pom.xml to instruct Maven to package and build this application as a .jar file:

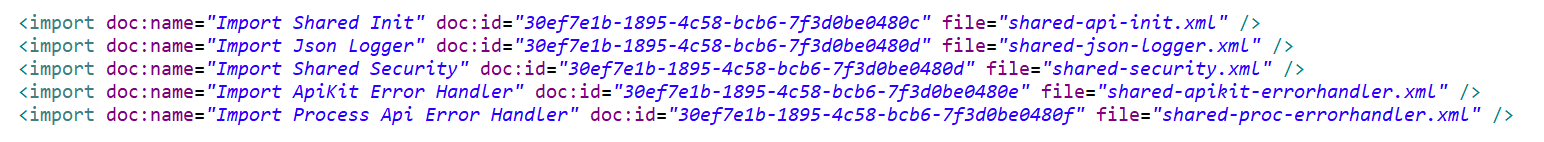


From the example above, the target file will be “mule-shared-api-commons.jar.” This file should be imported into an organization shared library repository. Typically, it can be a private Maven repository, a JFrog Artifactory, Mulesoft Exchange or similar. It’s recommended to manage all JAR dependencies in a private Maven repository or Artifactory. Mulesoft Exchange is not typically used to manage dependencies for an organization. However, Exchange can be a stop-gap dependency management platform until the organization enables a private dependency management solution such as Artifactory.

Once this JAR file is placed into the dependency management platform, all Mule applications can now use this shared component library by updating their Maven pom.xml file to add it as a dependency. In the example above, a sample Mule application pom.xml will have these lines added in the <dependencies></dependencies> section of the file:



The last step is for any Mule application that requires a shared flow or error handler to add this flow or error handler XML file via Mule 4.x Import statement. For example, in an application’s common.xml, add these lines to enable the shared flows and error handling:



With Mule 4.x Anypoint Studio 7, there is currently a defect where the Visual Editor shows compilation errors, complaining that the above shared XML files cannot be found. However, once a Mule developer runs their application in the local runtime, the errors are gone. Mulesoft is planning to update the current version of Anypoint Studio to address these and other compilation error bugs.

#### Shared Connectors and Modules

In addition to creating shared libraries of reusable Mule flows and error handlers, there may be a need to create a shared custom connector to integrate with the backend platform. First, it is recommended to review all existing connectors already developed by Mulesoft and the community. If there are no connectors on Exchange to meet the needs, a custom connector may be required. The connector is a Java component that is packaged as a connector and published to the organization’s private Exchange repository. Once the connector is created and published to Exchange, any Mule project that needs to use this connector can add it to the project by reviewing the list of connectors on the organization’s Exchange dashboard.

Custom connectors are modules that have a direct connection to a target backend source system. However, in some cases, there may be a need to create a custom module with reusable logic such as custom validation or security integration, without a need to connect to the backend source system. In that case, instead of creating a custom connector, a custom Mule 4.x module should be created. A module and connector are packaged the same way, as a Java custom component with specific configuration to instruct Mule that this is a custom module or connector. Once the shared module or connector is packaged, it should be added to the organization’s Exchange repository.

Custom modules and connectors will be visible in Anypoint Studio 7.x here:

