# Feedback:

Need content on using an API Catalog/Exchange/Marketplace. How to Providers put APIs in Catalog. How do Consumers get API’s in Catalog. What is the governance mechanism between the Provider, Consumer, and the API Custodian (i.e. Administrator) during the “Reuse process”. How do Consumers find and browse API’s that are reusable?

How to leverage an API Contract during “Reuse process”

Need content on using Shared Libraries and Common Re-Use. Examples, such as Error Handling, Logging, and other utilities.

I killed the section on 3 API Layers because we have content already defining the 3 API Layers. But, more importantly, how are API’s at each layer made reusable and how are they typically reused?

# INTRODUCTION

Introducing software reuse at a corporate level represents one of the most promising means of addressing the rising costs that are plaguing the software industry. A series of mechanisms are needed for shortening development cycles and providing reliable software of high quality which will be more maintainable and flexible for future extensions. API driven solutions can leverage these approaches, so that the VA solutions built on API’s can follow a “Re-Use First” approach to solution design.

This section describes guidelines for implementing an API reuse methodology focusing on a reusable API artifact library system and its role inside the reuse program. The goal for reuse program is to make API reuse a significant and systematic part of the software development process. To institutionalize an API reuse process that incorporates reuse-specific activities all along the object oriented software development process, drawing heavily on a reusable software artifact library system which has been designed to support the classification, management and search for artifacts to be employed in reuse efforts.

The API implementation patterns are mainstream and engineers can effectively take advantage of reusable components without having to tightly couple or make a duplicate copy of code for basic reuse, the prime question shifts from “what can we do to enable software reuse?” to “how can we instrument and measure our software reuse?”

# Definition of Reusability

Artifacts can be reused in multiple contexts in a project, including collections and modules. One can edit artifact content in a module or by opening the artifact outside the module. Content changes are reflected in all contexts; however, links, tags, and comments are specific to the context in which they are created.

Accelerated project delivery driven by reusable assets and shared services. To address the growing IT project delivery gap in government, agencies must stop “reinventing the wheel” with each incremental project, and instead, seek to build and leverage shared services. Traditional architectural approaches centered around point-to-point integration are unable to fulfill these needs. Without reusing integration work (such as API’s) across projects, or across agencies, governments are unable to increase the speed at which they can deliver projects without adding incremental staffing. Limited budgets typically rule this out as a tenable option. Furthermore, point-to point integration creates architectural brittleness over time, making the agencies less flexible and adaptive to change. It also creates a dependence on legacy systems, increasing maintenance costs and reducing budget for innovation and net new projects.

# Reusable Services for a Micro-services Architecture

The Javascript world has, for a recent while, been adopting the culture of actively building applications with reusable code, which has led to the numerous component-based frameworks. The prominent of these are ReactJs, VueJs and Angular. Since a component could contain its own markup, style and behavior, this component can then be reused across the web app or sometimes even in a separate app altogether. Furthermore, different developers could work on different components which can then be integrated to produce one app. This inherently led to improved productivity on the developer’s end as everything just became plug-and-play and they needed not to repeat themselves.

Micro-services are built with the same narrative of separating concerns, combining small parts of a system to form the bigger picture as it were (not producing a distributed monolith). Again different developers work on different micro-services of the same application/back-end and integrate these independent services with a message bus or via HTTP APIs. This separation brings about independent governance on the state and future of a service, leaving all the power to the developer/team responsible for that service.

There are similarities in different projects, (for example user management & messaging) or similar tasks that developers easily overlook. Building a bulk SMS API (like Frello) from the scratch for a single project. There is some necessary code that needs to be rewritten with every project to connect the relevant APIs, and depending on the nature of the projects.

# Reusable Micro-services approach

1. Building micro-services which are non-project specific (i.e. reusable)
2. Developers clone a repository
3. Set a few configurations
4. Add project specific logic

This will make an independent service is up and running and ready to be deployed. This approach will not only improve productivity but also starts a culture of sharing work with other developers, projects, and ultimately the Open Source Community itself

* Microservice capabilities are expressed formally with [business-oriented APIs](https://www.mulesoft.com/platform/api). They encapsulate a core business capability, and as such are valuable assets to the business. The implementation of the service, which may involve integrations with systems of record, is completely hidden as the interface is defined purely in business terms. The positioning of services as valuable assets to the business implicitly promotes them as adaptable for use in multiple contexts. The same service can be reused in more than one business process or over different business channels or digital touchpoints, depending on need. Dependencies between services and their consumer are minimized by applying the principle of loose coupling. By standardizing on contracts expressed through business oriented APIs, consumers are not impacted by changes in the implementation of the service. This allows service owners to change the implementation and modify the systems of record or service compositions which may lie behind the interface and replace them without any downstream impact. If microservices’ interfaces are exposed with a standard protocol, such as a REST-ful API, they can be consumed and reused by other services and applications without direct coupling through language bindings or shared libraries. Project teams need to easily discover services as potential reuse candidates. These services should provide documentation, test consoles, etc so re-using is significantly easier than building from scratch.

# Software Process for Reuse

Software process for reuse includes the set of activities that allow, during the course

of a project, the early identification of artifacts to be developed that might exhibit a high reuse

potential either within or in other iterations of projects. Besides, added costs and time (incurred for reuse addressed specific activities) estimation techniques will be defined.

We will discuss two types of reuse here:

* Vertical reuse (i.e., the reuse of software artifacts within a specific domain or application area)
* Horizontal reuse (i.e., the reuse of software artifacts across domains or application area).

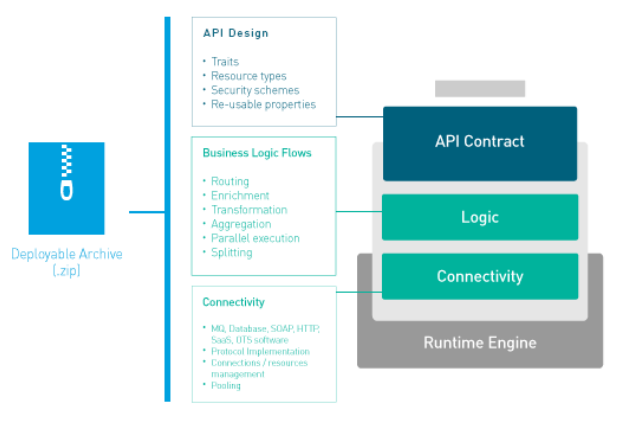
However, it puts a major emphasis on the latter, since horizontal reuse provides the highest payoff. The reusable components can be organized into a hierarchy, in a number of product layers, each playing a well-defined technical and market role. This reusability framework goal is to set the strategic direction for structuring organization’s software products to support reuse activities. The four layers are defined as below:

* *Operating Environment (layer 1):* it addresses the technological foundation of the software products. The architectural components of this layer reflect the selection of basic database, graphics, and communications enabling technologies for the Operating Environment.
* *Application (layer 2):* it characterizes software components specifically designed to

solve application problems for a broad telecommunication market (e.g., Billing, Service Pro-visioning, Customer Network Management, etc.). This layer is likely to contain a large number of reusable software components.

* *Market Segment Specific (layer 3):* it reflects the specificity of a market segment related to a specific problem-domain (e.g., the Service Provisioning Application can be customized for small PTfs, large end-users, and mobile telephony market segments).
* *Customer Specific (layer 4):* it represents those software components which are highly specialized to address application features specific to a particular customer. Components at this layer have typically a low reuse potential.

As we will show below, moving beyond point-to-point connectivity requires an approach centered around the development of connectivity assets that are both reusable across different projects and contexts, and discoverable by the teams that need them. In our experience, enabling that approach requires that monolithic services be broken down into smaller, constituent microservices. Agencies that employ this approach simultaneously realize greater project delivery speed and security, but only if they do so in such a way that drives reuse of these microservices across the enterprise.



Microservices promote reuse by exposing their functionality through a standardized API contract that any project team can leverage without needing to understand the underlying business logic of the microservice. In addition, as a function of their smaller scope, microservices can be used across a much larger variety of projects and business contexts. Furthermore, by decoupling services from their end consumers, multiple project teams from different domains can implement microservices with their own choice of technology, yet remain aligned with the broader mission of the agency, encouraging project teams to reuse existing microservices instead of building their own.

# Training and Education

The framework defines and enacts the policies to disseminate the reuse culture

and to encourage the adoption of reuse practices among project teams. It is principally

through this mechanism that the organization learns how to "reuse" effectively.

* *Provide expertise:* whenever possible, engineers work with the project teams to better support them in achieving both development "for reuse" and "with reuse".
* *Development of reusable components:* dedicated teams are in charge of producing reusable components by either re-engineering the "normal" software components or by developing new components whose reuse potential has been assessed during domain analysis. The act of developing reusable components asynchronously with project team activities is called an "a-posteriori (or backroom)"approach.
* *Maintenance of reusable components:* reusable components may change as flaws are corrected and enhancements are made.
* *Maintenance and administration of the framework*: the central repository is managed by a single team which controls the nature and quality of the information stored in the repository. This activity also deals with the population of the library of artifact descriptors, including setting policies for artifact acquisition and maintenance (e.g., evaluation, certification, classification, and weeding). In addition, it has to define and maintain a coherent classification scheme for the stored artifacts, weeding old artifacts, announcing incoming ones, user management (e.g., add/delete users, set user privileges), and reporting on artifact usage.

# Classification of an artifact

The three classification methodologies implemented in the reusability framework are different in the way the artifacts are classified and retrieved.

1. When a new artifact is inserted, it has to be classified in all of the methodologies supported by the system. In fact, if this is not done, the artifact will not be found when a search based on the missing methodology is performed.
2. For the free text classification, the artifacts are classified associating them with a text-description of arbitrary length which describes the artifacts themselves.
3. In the keywords classification, each artifact in the system is associated with a set of keywords that characterize the artifact itself and are chosen from a well-defined dictionary. The dictionary can be expanded by the system administrator, who has the capability to remove unused keywords

# Retrieval of an artifact

1. An artifact stored using the reusability can be retrieved by navigating through the relations established between the artifacts or performing a search based upon one of the classification methodologies implemented in the system.
2. The search is fundamental even when navigating through the relations in fact, it is necessary to find an initial artifact from which to start the navigation. The search is done in different ways depending on the classification it uses:
   * The user has to do a query in a format which depends on the different kinds of classification. If the results of the search are not satisfactory, the query can be relaxed or modified in some way and the search

# APPROACHES FOR REUSE

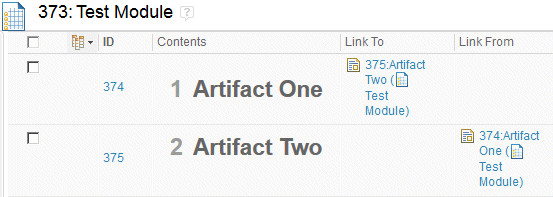
Reuse approach can be of two types:

* + Generation-based: The generation-based approach reuses the process of previous software development efforts, often embodied in computer tools that automate a part of the development life cycle.
  + Composition-based: The composition-based approach reuses existing software products in a new system to avoid repetitive work. As mentioned in the previous section, many types of software products can be reused. However, because this research focuses on the reuse of components, the discussion here is limited to component reuse, although many problems and solutions discussed can be extrapolated to the reuse of other software products. Component reuse is also known as *component-based development*. Based on the role that components contribute to the programming process, component reuse is further divided into three categories.
* **Black-Box Reuse:** In black-box reuse, a component is directly reused without modification. A component can be reused as it is or reused through inheritance if the programmer creates a specialized subclass of an existing class component.
* **White-Box Reuse:** In white-box reuse, programmers reuse the component after they have modified the components to their needs. White-box reuse does not contribute as much to the easier maintenance and evolution of software systems as black-box reuse does, but it can reduce development time.
* **Glass-Box Reuse:** In glass-box reuse, programmers do not directly reuse the component; instead, they use it as an example for their own development. For instance, programmers can look at examples to find out how a program plan is realized and build their own system through analogy.

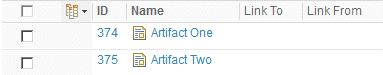
[API-led transformations](https://blogs.mulesoft.com/dev/api-dev/what-is-api-led-connectivity/) have demonstrably unlocked both code and component reuse for enterprises and small shops alike, a set of easily understandable KPIs have begun to emerge to measure how well an IT shop has integrated the ethic of software reuse.

# STORING ARTIFACTS IN MODULES

* To reuse artifacts in a module, existing artifacts should be added to a module or create new artifacts in the module. The artifact can be reused by copying it from one module and pasting into another module.
* After addition or creation of artifacts in modules, work can be done with the artifacts in the module or on the Artifacts page. When you work with an artifact outside a module, you are working with the *base* artifact. Changes to the artifact name, content, and attributes are reflected in all contexts: in the base artifact and in all modules that include the artifact.
* Creation of links, tags, and comments for artifacts in the context of a module, those contextual elements are applicable and visible only in the module where are they created. For example, in the following module, artifact 374 is linked to artifact 375 and artifact 375 is linked from artifact 374:



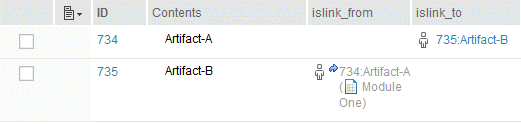
* However, the base artifacts 374 and 375 on the Artifacts page, are not linked:



## Using bi-directional links in a module

The user should be able to link one artifact in a module to another artifact in a module using a bi-directional link type, the link will appear with a blue icon next to it when you select Include links, tags and comments from base artifacts to indicate that the link is from a base artifact.

If user Include links, tags and comments from base artifacts, an indication must be shown as below:



# GENERAL GUIDELINE:

* When user creates links, tags, and comments for base artifacts, those elements apply to the artifact in all contexts in the project. However, those elements should not be shown by default in modules.
* The user should be able to override the default display settings in a module and show the links, tags, and comments for the base artifacts in addition to the links, tags, and comments that are created in the module.
* If you create links between *base* artifacts, those links are available in every module that the base artifacts are used in.
* If you create links between artifacts in the context of a module, the links exist only in that module and in any copies you make of the module.
* If you want to link to an artifact that is reused, when you create the link, you can access the artifact by selecting from the **Rows in the module** list.
* To help new developers get started and to make sure that they are productive as soon as possible, you can assist them with a few useful code templates. These are not meant to hamper innovation by helping them get started and quickly get over potential initial problems that would be counter productive. Code templates can also be used for various simple functions that needs to be performed on each Microservice, which would not only help developers to avoid doing repeatable work, but also make sure that the most efficient function is used and reused for this functionality.
* Measure the software reuse across a portfolio of assets, but with the proliferation of API-led transformations, enterprise IT shops have never been as poised as they are right now to enact and measure the concept of software reuse across an enterprise.
* Must have requirements: Tracking and driving reuse through instrumented sources is necessary to help drive the spread between productivity and labor curves for a technology shop; this is because reuse is a prime method of growing revenue at lower incremental costs. This widening of the spread between these curves serves to fend off [digital disruption](http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/perpetual-evolution-the-management-approach-required-for-digital-transformation) from competitors who are no longer bound by historical barriers to entry.
* Tracking and driving technology health metrics through instrumented sources including, but not limited to, speed, quality, reuse, cost per feature and others is necessary to build and maintain trust between organizations with different near-term goals and objectives.

# REUSABILITY FRAMEWORK

\

Analysis

Architectural Design

Component Engineering

Testing

Component Qualification

Component Adaptation

Component Composition

Component Update

Component-Based

Development