# API Resource design guidelines

**DEPRECATED - Please use:**API Design Recommendations

## Resource Naming Conventions

The key principles of REST support separating an API into logical resource and manipulating these using HTTP methods, where each method has a specific meaning. A resource represents object types within your domain.  For example, /employees would present employees with your organization.

## Use Nouns, Not Verbs

A URI should refer to a resource that is a “thing” instead of referring to an action. Therefore, avoid using “actions” within your resource name. For example, getOrders or deleteOrder should be avoided. The action prefix should be implied by the HTTP method i.e. GET or DELETE.

Examples of resources would be:

/orders

/payments

/shipments

## Use Plural Nouns

It is common practice to standardise on using plural nouns over a mixture of both singular and plural nouns in URIs. The makes it consistent and predictable for developers.

## Use Concrete Nouns

Avoid making a resource name too abstract. Tunneling a number of objects through an abstract resource name makes it difficult to understand what the resource actually represents or how the API resource should be used. For example, consider the different types of services – order, payment and shipment. Representing all these services as a /services resource is too abstract.

## Identify Actions

The next step in the design process is to identify the actions that can be performed on the resource model. When designing RESTful API the API designer should try to use as much as possible CRUD actions, that is, Create, Retrieve, Update and Delete. Similar to the “nouns” rule, the API designer should consider the verbs and try to identify candidate actions.  For example, the customer should be able to create a purchase order, to add products to that order (that is, to create Order lines that refer to a specific product and belongs to a Purchase order), and to retrieve his/her orders. Further, the customer wants to be able to change (that, is to update) the shipping address of an order. It could be the case that the customer might want to completely delete a purchase order that has not been shipped yet.

## Map Actions to HTTP Verbs

On the first view REST using standard HTTP verbs are a matching implementation of the CRUD pattern. The main difference is that CRUD is a pattern to be applied to entities in systems and data stores while REST using HTTP commands implements system behaviour. Nevertheless, in many cases the CRUD-REST mapping can be used to explain the behaviour of the RESTful API. Closer inspection shows that there are differences which need to be taken into account. In the majority of the cases REST services can be mapped as shown in the following table:

|  |  |  |
| --- | --- | --- |
| **CRUD** | **Description** | **HTTP mapping in REST services** |
| Create | Create an Entity on the service side | POST |
| Read | Retrieve an Entity from the service | GET |
| Update | Implements a persistent change of the Entity on the service | PUT (replace) / PATCH (partial update) |
| Delete | Remove an Entity from the service | DELETE |

Problem in the mapping is the use of the POST and PUT verbs. While PUT is commonly used to update an entity on the service side (the entity does exist and some of the attributes are modified), the POST verb is commonly used to create a new entity (as a subordinate of a special resource). Some implementations are using the PUT verb for “create” and “update”, with the difference that the implementation creates an entity on the service side if it does not exist during an “update” function. Important for the particular implementation is the requirement that the behaviour of PUT and POST are different: the PUT verb is idempotent (i.e. if the verb is repeated, the outcome on the service side is identical) while POST is not.

In some cases, the action do not map to CRUD actions. There are several approaches to deal with this:

* Restructure the action to operate on a field of a resource. This works if the action does not use parameters. For example, ship order could be mapped to PATCH that updates only the filed “shipped”.
* Use a sub-resource to capture the state that results from the execution of the action. For example, the state of an offer could be changed by creating a sub-resource Shipped Order. Then, an order state could be changed by POST /orders/{id}/shipped-order.
* In some cases, it is impossible to map an action to a reasonable resource structure. For example, a search action over a collection of multiple, different resources. In such cases, it makes sense to define a “fake” resource “Search” but this has to be documented clearly to avoid confusion.

## Resource States

Analysing the user stories, the API designer finds out that the user should be able to perform some of the action only if certain preconditions are met. For example, it should be possible to change the shipping address or cancel an order only if the order has not been shipped yet. This implies that the API resources could have different states, e.g., order could be created, shipped or cancelled. It is a good practice to capture the states of the resources and the respective transitions that change the state of a resource in a state diagram as shown below: