# 

API Patterns and Best Practices

**Review Log**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version Number | Created By | Date | Reviewed By | Date of next review | Notes |
| 0.1 | Stanislav & Andreas | 11-11-2015 |  |  | first draft |
| 0.2 | Stanislav Pokraev | 13-11-2015 |  |  | work in progress |
|  |  |  |  |  |  |

# 

# 

# 

## **Best Practices for Resources in APIs**

The following section details the Best Practices for Resources in the API context.

### **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 organisation.

#### **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. Tunnelling 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 / PATCH |
| **Delete** | Remove an Entity from the service | DELETE |

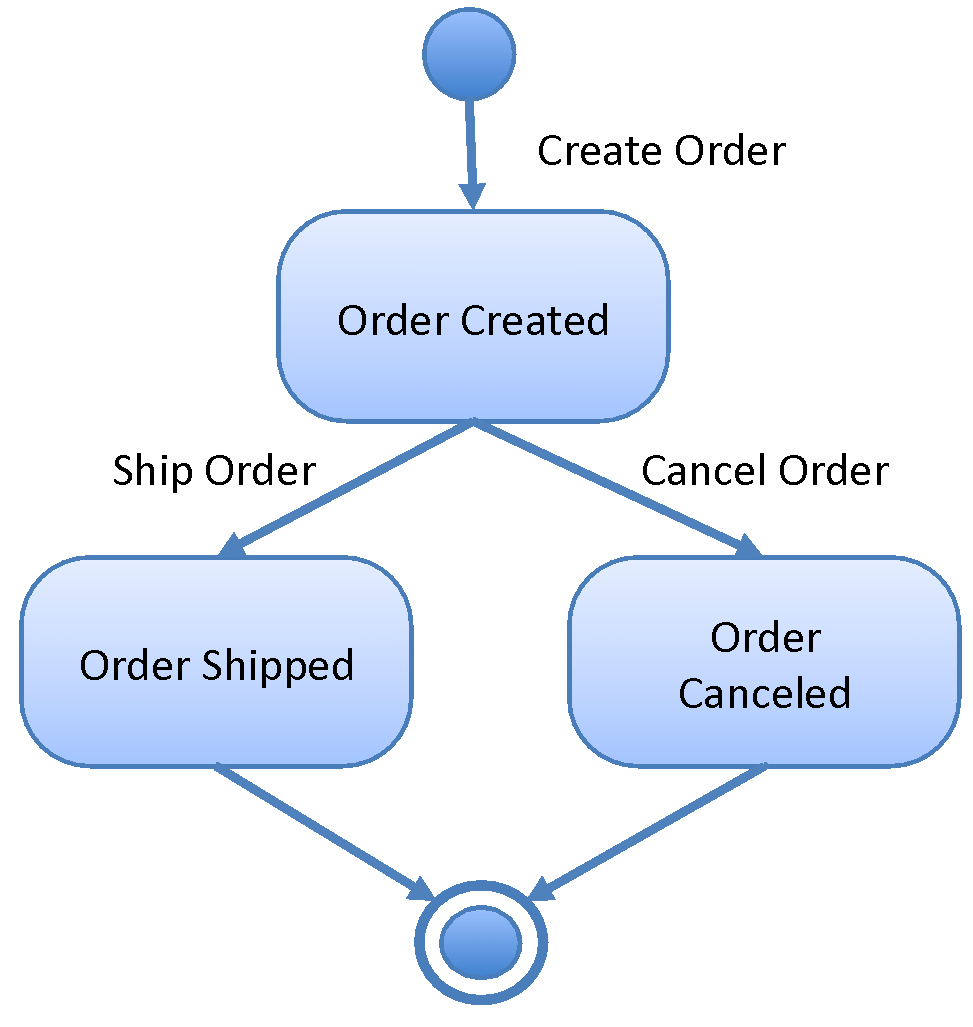
Problem in the mapping is the use of the POST and PUT verbs. While PUT is commonly used to update an entity on the service side (the entity does exist and some of the attributes are modified), the POST verb is commonly used to create a new entity (as a subordinate of a special resource). Some implementations 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:



### **Design API URLs**

The structure of a URI is central to how APIs are organised and categorised within your enterprise domain. A good URI taxonomy helps to categorise your APIs across functional domains, regions, access (public or private) and helps define relationships (hierarchical). A good URI also helps to govern the lifecycle of your API through versioning practices.

Recommended URI Structure:

|  |  |  |
| --- | --- | --- |
| Part | Description | Example |
| {env} | Optional. The API environment. An API could be available in a sandbox environment to enable developers to test that API. The {env} part is excluded for production APIs. | sandbox |
| {access} | Optional. The access level of the API. This could be public or private. By default the {access} part is excluded for public APIs or simply set to api. | api. |
| {company} | Required. The name of the company or business division for private services. | mytaxis. |
| {region} | Required. The region of the API | .co.uk |
| {context} | Required. The name of the API as defined in the API Manager. This typically presents the business service and should be a short but descriptive name. | quickbooker |
| {version} | Required. The version of the API. Depending on requirements, the version can reflect only major versions or include a more hierarchical convention to identify minor versions. | v1 |
| {resource} | Required. The name of the resource that represents the actual object. An API may contain multiple resources. The resource can also be referred to as the API endpoint. | bookings |
| {resource-id} | Optional. The id of the resource to be fetched/updates. The resource id is optional. | 1981927 |
| {queryparams} | Optional. The query string can define state transition parameters. | page=1&sort=+<field> |

### **Setting the Base URI**

The Base URI is defined within the RAML and takes the form:

**https://[env].[access].[company].[region]/[context]/[version]**

The API resources are defined relative to the base URI.

**Filtering**

In some cases, the API consumer might only need a subset of a collection of resources. This could be accomplished by using query parameters. For example, to get the list of all shipped orders, the API consumer could use:

GET /orders?state=shipped

Here, the state is query parameter that is used to implement filtering.

**Sorting**Similar to filtering, a generic query parameter sort could be used to describe sorting rules. To allow sorting on multiple fields, the query parameter could be design to take a list of fields instead of a single value. Next, to allow for ascending and descending sort order, the query parameter could take minus (“-“) as prefix of each field. For example, the following request will return all purchase orders sorted by data (descending) and then by product (ascending):

GET /orders?sort=-date,product

**Partial resources**

In some cases, the consumer might not need all the fields of a resource. To allow for obtaining only a partial resource the API URL could be design to take a list of fields as a query parameter, and return only the fields that are includes in that list. For example, the following request will return only the date and the total of the purchase order:

GET /orders/1?fields=date,total

**Aliases**

To make the experience of using an API more pleasant for the application developers, the API could package a set of conditions into an easily accessible URL. For example, to return the recently shipped orders, the API could provide the following endpoint:

GET /orders/most-recent

A resource name should remain short in order to avoid any size limitations. The base URL should also contain no more than 2-3 resources if possible. URIs can be limited in some HTTP stacks.

### **Design API Request and Response Representation(s)**

This step involves specifying the format of the API’s request and response messages. A good starting point would be to investigate whether there is an existing, standard format and media type that matches the API use cases and requirements. For that purpose use IANA[[1]](#footnote-1).

If there is no standard media type and format, try to use as much as possible extensible formats such as JSON (application/json) and XML (application/xml), preferably JSON.

Further, to increase the interoperability, use standards such as W3C XML Schema, ISO 3166, ISO 4217, RFC 3339 to represent dates, times, numbers, currencies, countries.

Last but not least, use multipart media types such as multipart/mixed, multipart/related, or multipart/alternative to encode request or response that contain mix of textual and binary data, that is, avoid encoding binary data within textual formats using Base64 encoding.

ToDo: Update with RAML 1.0 Data types

### **Design API Request and Response Headers**

ToDo: Needs update

* Use Content-Type, Content-Length, Content-Encoding and Content-Language to provide additional metadata that helps to interpret the content of the API request and response messages.
* Use Location header to include a link to the current resource or a collection or to provide a link to a queue (in the case of asynchronous scenario), for example, for example

Location: http://www.example.org/users/1

* Use Last-Modified header to indicates the last time resource has been modified
* Use ETag to represent the a specific “version” of the resource. Clients may choose to save an ETag header’s and use it in future GET requests, as the value of the conditional If-None-Match request header. If the REST API concludes that the entity tag hasn’t changed, then it can save time and bandwidth by not sending the representation again.
* Use Cache-Control, Expires, and Date response headers to encourage caching.
* Use Authorization to implement authentication and authorization

|  |  |
| --- | --- |
| Header | Description |
| Accept | Content-Types that are acceptable for the response, e.g., application/json |
| Accept-Charset | Character sets that are acceptable for the response, e.g., utf-8 |
| Accept-Encoding | List of acceptable encodings, e.g., gzip |
| Accept-Language | List of acceptable human languages for response, en-US |
| Content-Type | The MIME type of the body of the request (used with POST and PUT requests), e.g., application/json |
| Date | The date and time that the message was sent, Mon, 15 Dec 2014 08:15:30 GMT |
| Authorization | Authentication credentials for HTTP authentication, e.g., Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| If-Match | Tells the API to only perform the action if the client-supplied entity matches the same entity on the server. This is mainly used in PUT to only update a resource if it has not been modified since the user last updated it, e.g., If-Match: v1 |
| Pragma | Implementation-specific fields that may have various effects anywhere along the request-response chain (e.g., no-cache) |
| Cache-Control | Used to specify directives that must be obeyed by all caching mechanisms along the request-response chain (e.g., no-cache) |
| Connection | What type of connection the user-agent would prefer (e.g., keep-alive) |
| ETag | An identifier for a specific version of a resource (e.g., ETag: v1) |
| Expires | The date/time after which the response is considered stale, e.g., Mon, 15 Dec 2014 09:20:20 GMT |
| Last-Modified | The last modified date for the resource, e,g., Mon, 15 Dec 2014 10:10:30 GMT |

### **Design Response Codes**

HTTP status codes are grouped into five numeric categories:

|  |  |
| --- | --- |
| **Code** | **Type** |
| 1xx | Informational |
| 2xx | Successful |
| 3xx | Redirection |
| 4xx | client error |
| 5xx | server error |

These response codes should be used as standard, the use of not defined return codes is discouraged and should only be done in exceptional circumstances.

|  |  |  |  |
| --- | --- | --- | --- |
| Code | HTTP Method | Response Body | Description |
| 200 OK | GET, PUT, DELETE | Resource | There are no errors, the request has been successful |
| 201 Created | POST | URI of the resource that has been created | The request has been fulfilled and resulted in a new resource being created |
| 202 Accepted | POST, PUT, DELETE | An URI of a resource which represents the processing status | The request has been accepted for processing, but the processing has not been completed |
| 204 No Content | GET, PUT, DELETE | N/A | There are no errors, the request has been processed and no contact is expected in the body (by design) |
| 304 Not Modified | conditional GET | N/A | The resource has not been modified : there is no new data to return |
| 400 Bad request | GET, POST, PUT, DELETE | Error message | The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications |
| 401 Unauthorized | GET, POST, PUT, DELETE | Error message | The request requires user authentication |
| 403 Forbidden | GET, POST, PUT, DELETE | Error message | The server understood the request, but is refusing to fulfill it. For example, Authentication failure or invalid Application ID. Authorization will not help and the request SHOULD NOT be repeated |
| 404 Not Found | GET, POST, PUT, DELETE | Error message | The server has not found anything matching the request URI |
| 405 Method Not Allowed | GET, POST, PUT, DELETE | Error message | The method specified in the request is not allowed for the resource identified by the URI |
| 406 Not Acceptable | GET, POST, PUT, DELETE | Error message | The request contains parameters that are not acceptable |
| 408 Request Timeout | GET, POST, PUT, DELETE |  |  |
| 409 Conflict | POST, PUT, DELETE | Error message | The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request |
| 410 Gone | GET, POST, PUT, DELETE | Error message | Used to indicate that an API endpoint has been turned off. Could be used to deprecate API, for example, to inform the customer that the API will soon stop functioning and to migrate to new version of the API |
| 412 Precondition Failed | GET, PUT | Error message | The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended |
| 415 Unsupported Media Type | GET, POST, PUT | Error Message | The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method |
| 429 Too Many Requests | GET, POST, PUT, DELETE | Error message | Indicates that the user has sent too many requests in a given amount of time ("rate limiting") |
| 500 Internal Server Error | GET, POST, PUT, DELETE | Error message | The server encountered an unexpected condition which prevented it from fulfilling the request |
| 502 Bad Gateway | GET, POST, PUT, DELETE | Error message | The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request |
| 503 Service Unavailable | GET, POST, PUT, DELETE | Error message | The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay |
| 504 Gateway Timeout | GET, POST, PUT, DELETE | Error message | The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request |
| 509 | GET, POST, PUT, DELETE | Error Message | Bandwidth Limit Exceeded (not included in the W3C standard, implemented as part of the Apache tooling) |
| 510 Not Extended | GET, POST, PUT, DELETE | Error message | The policy for accessing the resource has not been met in the request. The server should send back all the information necessary for the client to issue an extended request |
| 511 | GET, POST, PUT, DELETE | Error Message | Network Authentication Required (not included in the W3C standard) |
| 550 | GET, POST, PUT, DELETE | Error message | Permission denied (not included in the W3C standard |
| 598 | GET, POST, PUT, DELETE | Error Message | Network Read Time-Out Error |
| 599 | GET, POST, PUT, DELETE | Error Message | Network Connect |

## **API Patterns**

### **Endpoint Redirection**

Problem

A service provider may change the endpoint of its service over time for business or technical reasons. It may not be possible to replace all references to the old endpoints simultaneously.

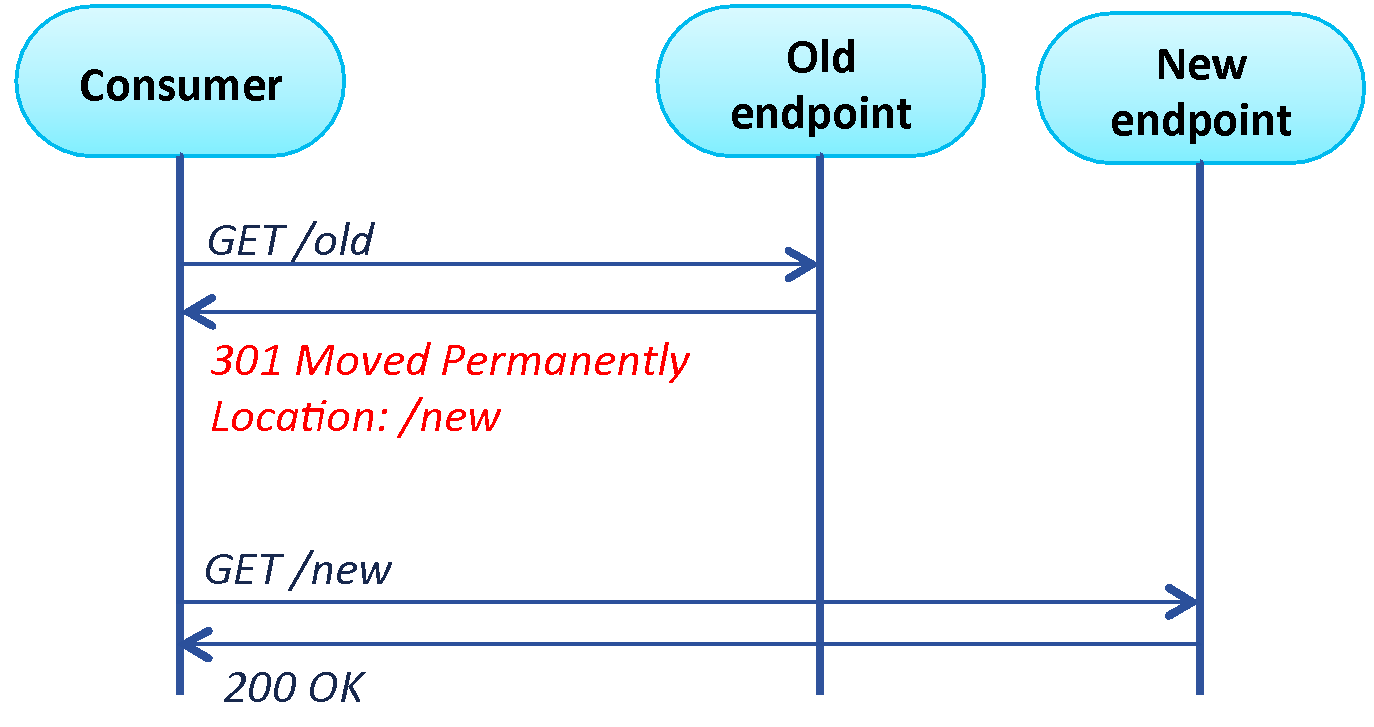
Solution

Automatically refer service consumers that access the old endpoint to the new one.

HTTP natively supports this pattern by using a combination of 3xx status codes and standard headers:

* Code 301 Moved Permanently
* Code 307 Temporary Redirect
* Header Location: /new

Redirection responses can be chained, however, be careful not to create redirection loops



**Figure 12. Endpoint redirection**

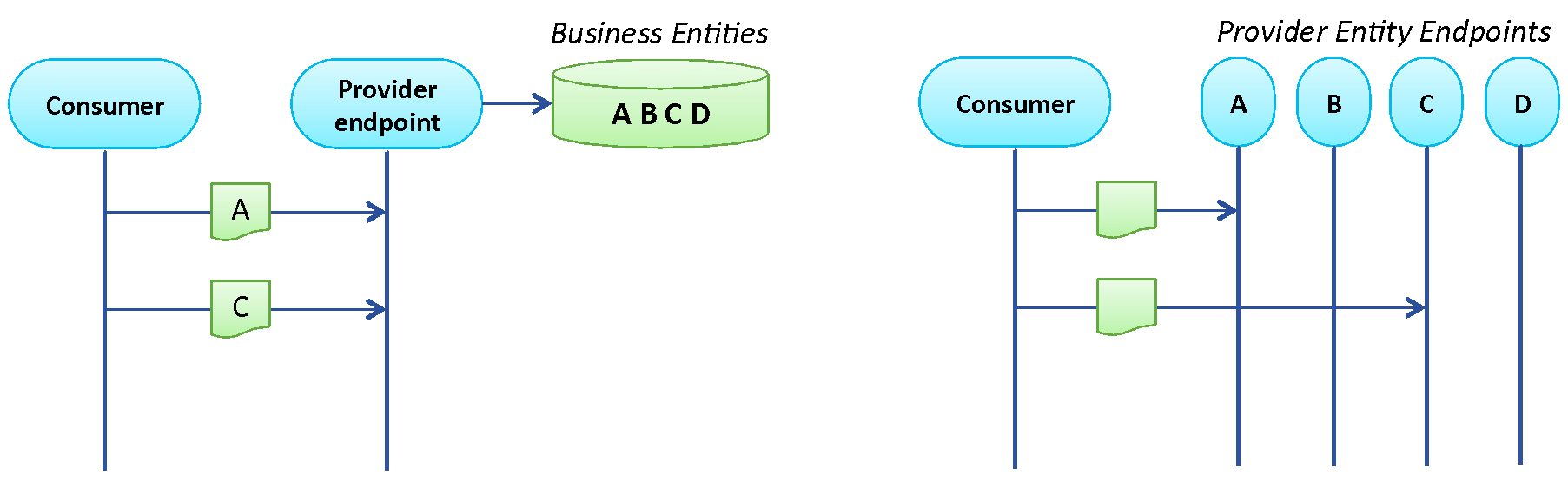
### **Endpoint Per Entity**

Problem

A service with a single endpoint is too coarse-grained when it provides capabilities to manage multiple data entities. A consumer needs to work with two identifiers: a global one for the service and a local one for each business entity. Entity identifiers cannot be reused and shared among multiple services.

Solution

Expose each entity as individual lightweight endpoints of the service they reside in. The benefit of this approach is global addressability of business entities.



**Figure 13. Fine-grained resources**

### **Content Negotiation**

Problem

Service consumers may change their requirements in a non-backwards compatible way. A service may need to support both old and new consumers without having to introduce a specific capability for each kind of consumer.

Solution

A service capability could be negotiated at runtime and based on the outcome, a specific content and data representation formats are returned by a service. The service contract refers to multiple standardized “media types”. The benefits of this solution are loose coupling and increased interoperability

GET /resource

Accept: text/html, application/xml, application/json

The client lists the set of understood formats (MIME types)

200 OK

Content-Type: application/json

The server chooses the most appropriate one for the reply (status 406 if none can be found)

Quality factors allow the client to indicate the relative degree of preference for each representation (or media-range) using

Media/Type; q=X

If a media type has a quality value q=0, then content with this parameter is not acceptable for the client.

Accept: text/html; q=0.1 application/xml; q=0.9

The client prefers to receive xml, and html as fall back.

|  |  |  |
| --- | --- | --- |
| **Request Header** | **Example Values** | **Response Header** |
| Accept: | application/xml, application/json | Content-Type: |
| Accept-Language: | en, fr, de, es | Content-Language: |
| Accept-Charset: | iso-8859-5, unicode-1-1 | Charset parameter for the Content-Type header |
| Accept-Encoding: | compress, gzip | Content-Encoding: |

### **Idempotent Capability**

Problem

Problem: Service oriented architectures are distributed systems. Failures (such as the loss of messages) may occur during service capability invocation. A lost request should be retried, but a lost response may cause unintended side-effects if retried automatically.

Solution

Use idempotent service capabilities, whereby services provide a guarantee that capability invocations are safe to repeat in the case of failures that could lead to a response message being lost.

Idempotent

Idempotent requests can be processed multiple times without side-effects

GET /book

PUT /order/x

DELETE /order/y

If something goes wrong (server down, server internal error), the request can be simply repeated until the server is back up again.

Safe

Safe requests are idempotent requests that do not modify the state of the server (can be cached). In contrast, unsafe requests modify the state of the server and cannot be repeated without additional (unwanted) effects. Unsafe requests require special handling in case of exceptional situations (e.g., state reconciliation).

POST /order/x/payment

In some cases the API can be redesigned to use idempotent operations:

GET /account/1/balance //safe

New Balance = Balance + 200$ //calculate on the customer side

PUT /account/1/balance (New Balance) //idempotent

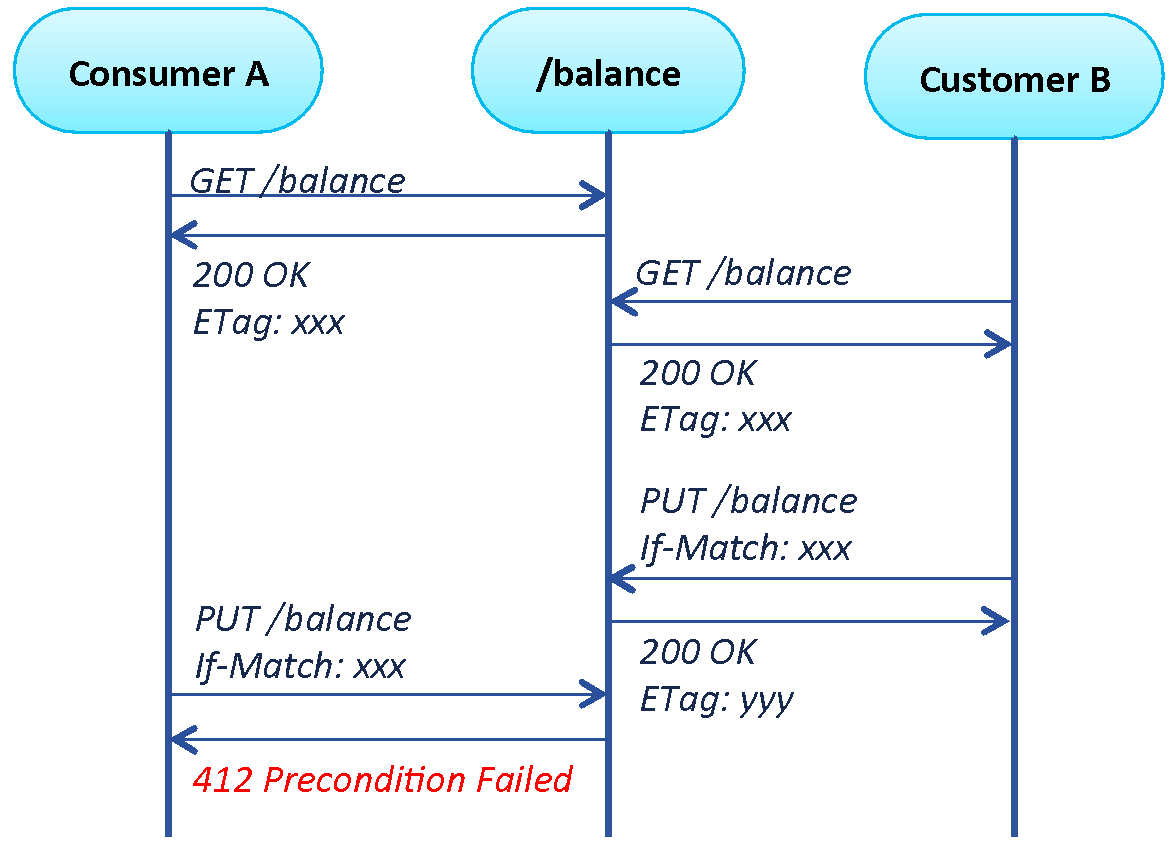
### **Concurrency**

Problem

Breaking down the API into a set of idempotent requests helps to deal with temporary failures. Problem arises when another client concurrently modifies the state of the resource that is to be updated.

Solution

ETag and If-Matchheaders are used to implement optimistic locking. Code 412 Precondition Failed can be used to inform the customer that she works with out-of-date resource.



**Figure 14. Concurrency and Optimistic Locking**

### **Asynchronous Processing**

Problem

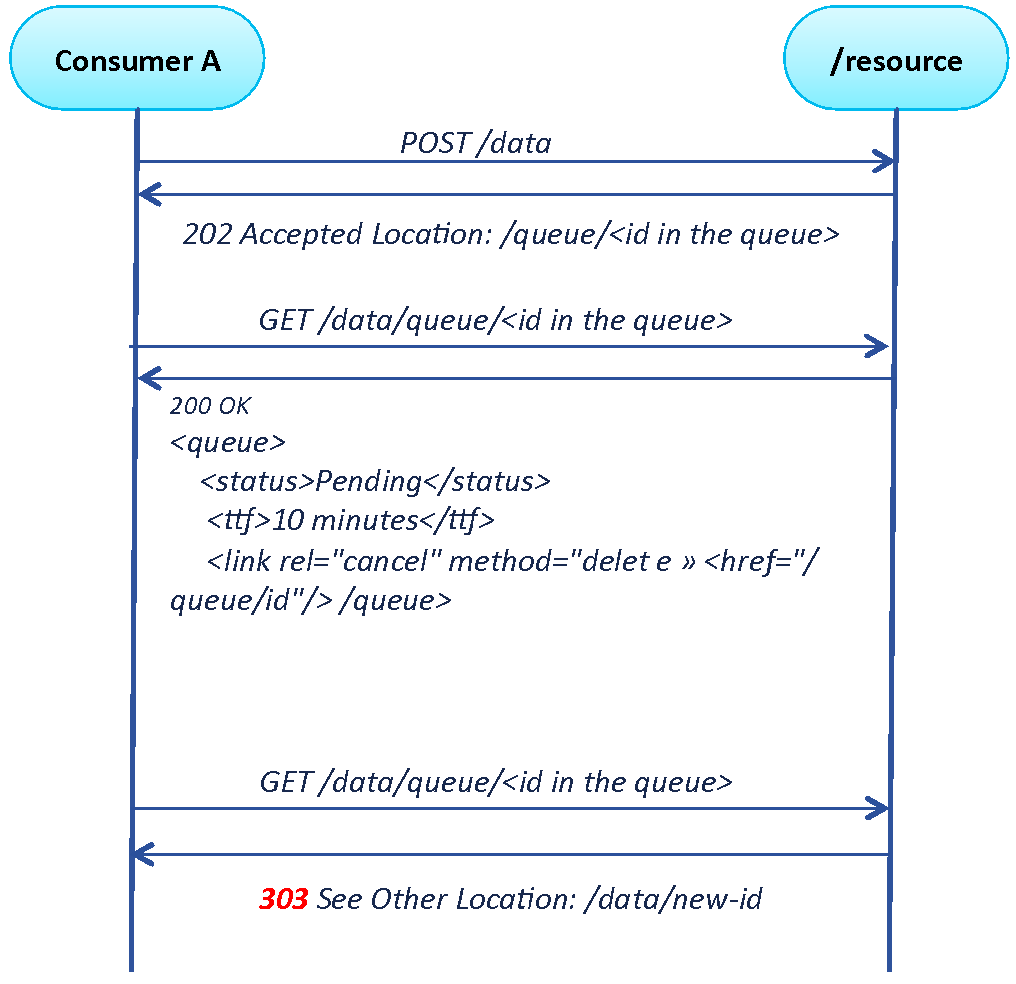
Rest is based on synchronous protocol (HTTP). Through API, we may trigger asynchronous work or process that we want to follow state and track termination to better chain API or service call.

Solution

Asynchronous processing is based on HTTP code 202 Accepted, HATEOS for process management and status and client async checking (for instance, AJAX method in javaScript)

The asynchronous process could be managed by checking regularly the process URL given as a response to the (async) service request.

The process service can give time to finish indication, status and link through HATHEOS on cancel or other subsequent actions available on the process.

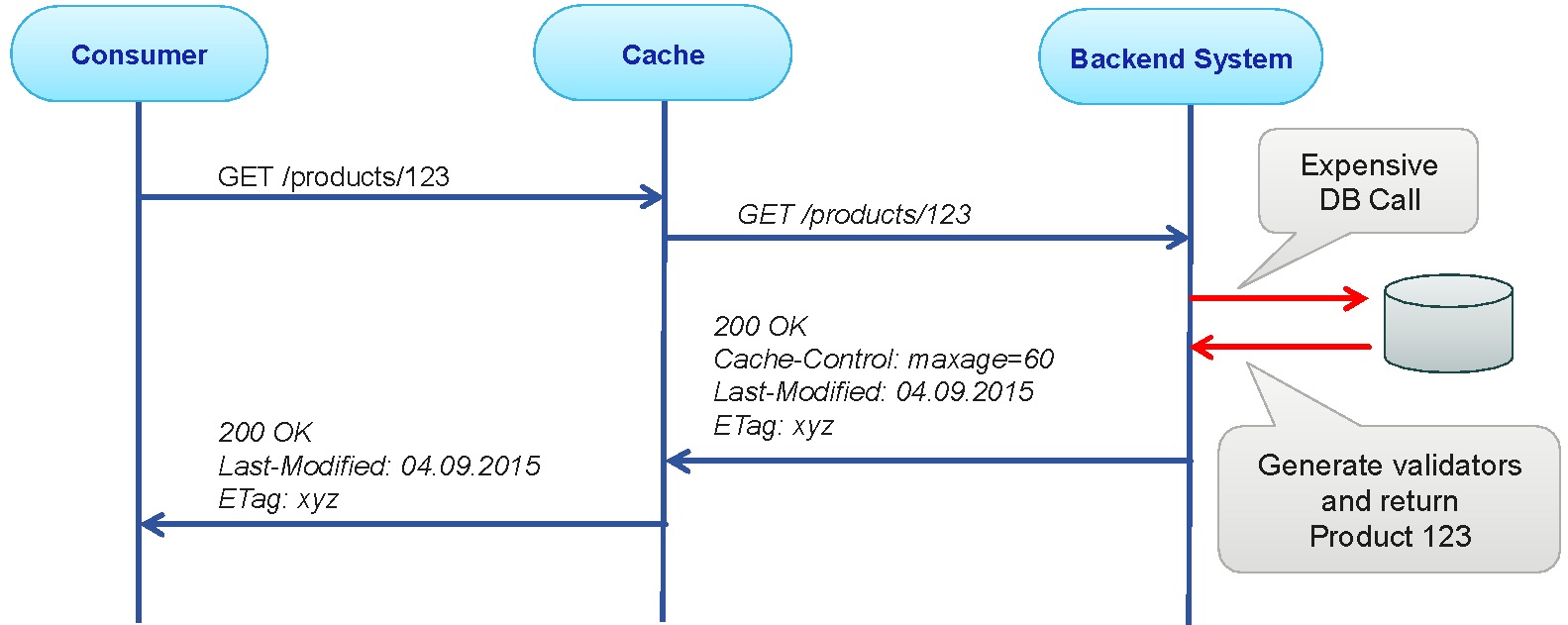


### **Caching**

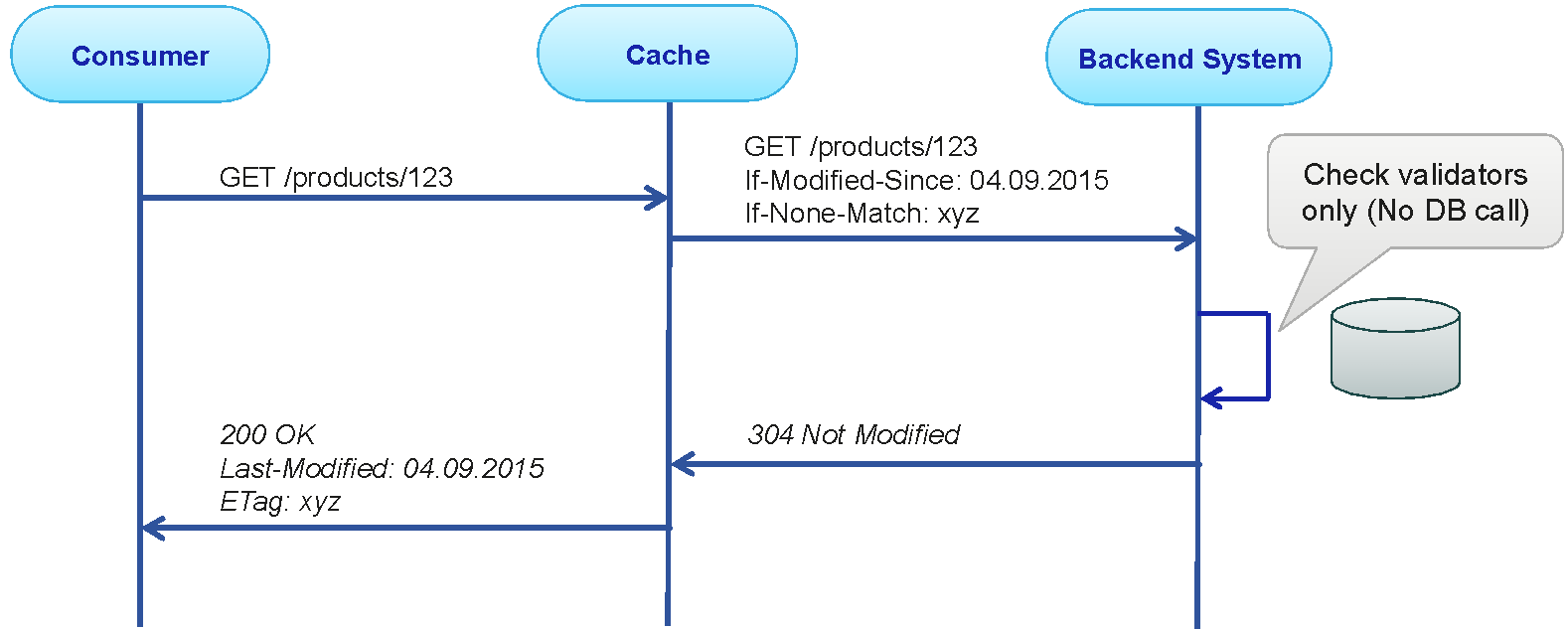
The Caching Pattern has been introduced to improve the performance and load management of the API. In many cases the information requested from the API is fairly static, with limited change over time (typical examples: address information on locations, price information, …). In these cases the API does not necessarily have to retrieve the information from back-end systems, but keep it in memory close to the API, called a Cache. Only if the information is not available in the Cache, the integration retrieves the information from the back-end. Based on the frequence of the retrieval and the size of the Cache, this information from the back-end could be added. The following charts describe an example of the caching pattern:

The Consumer requests information on the product 123 from the product API. The Cache does not have valid information on the product in the store and sends the request to the back-end system which retrieves the information.

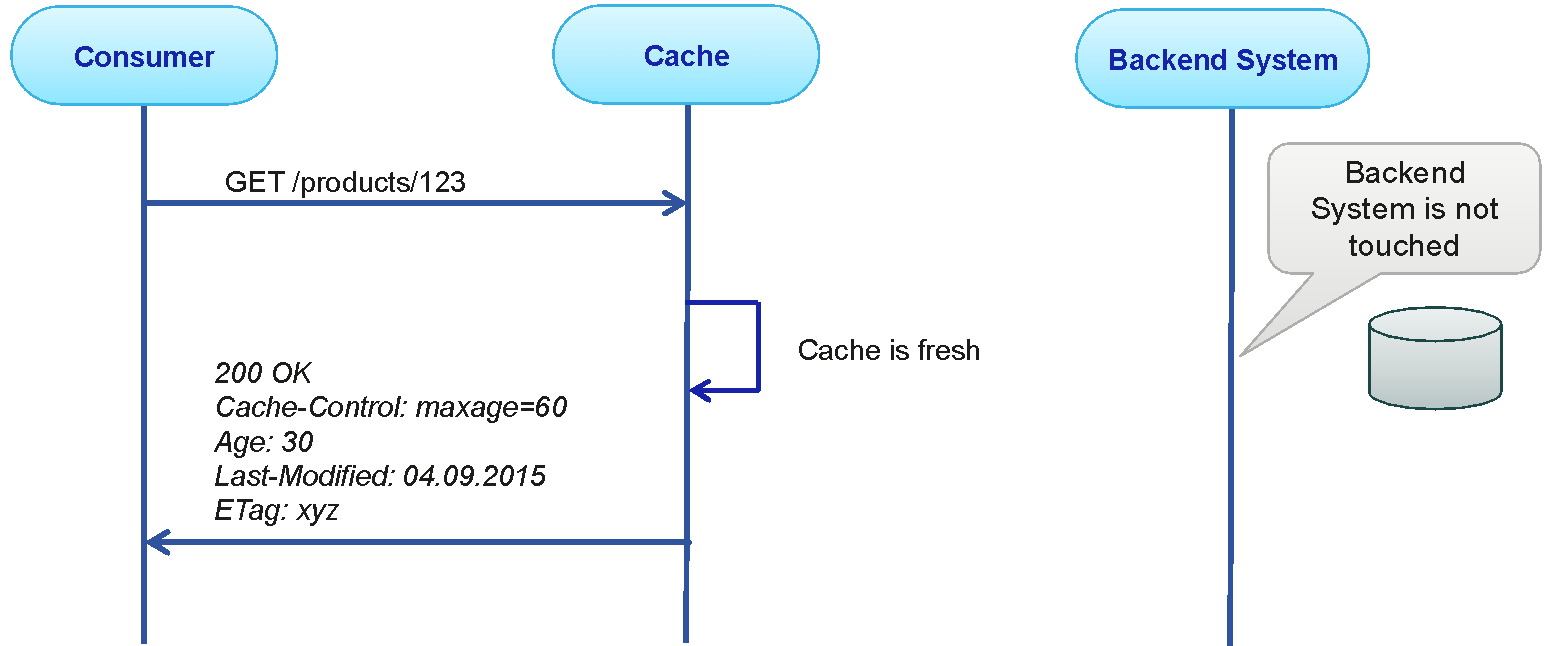
As this is the first request of the information the Cache does not keep the information as frequently used in local memory, but returns it back to the consumer.



The second request for the product now triggers the cache to inspect the information retrieved. As the information has not changed it is likely to be static. Also the information has an expiration time added, so the Cache can keep the information local for any following request.



Any following request will now be handled by the Cache unless the information is expired in the Cache or (because of changes in the back-end system) removed from the Cache.



It is important that the Cache is refreshed (i.e. deleted) if the back-end is changed, or the maximal Cache storage period (age of the information) exceeds the allowed age of the information.

## **Versioning**

An API should be designed for long-term, but since change is inevitable and the API will never be completely stable. What is important is to manage the changes well through comprehensive versioning strategy. This includes multi-month deprecation schedule and documentation that is up to date and describes the change in sufficient level of detail. Not all changes require new version. API’s designed for backwards compatibility do normally not require new versions for minor changes and is recommended practice to design for this. Nevertheless, it is not always possible the design APIs in a way. The table below outlines when to introduce a new version and when a new version is not required.

|  |  |  |
| --- | --- | --- |
| **Type of Change** | **Requires a new version** | **New version NOT required** |
| **Service contract** | * Remove API endpoint * Change effect of API operation * Change response type of API operation * Add new required operation parameter | * Add new API operation |
| **Data contract** | * Remove existing element (or attribute) * Add new required element (or required attribute) * Change existing element (or attribute) | * Add optional element (or attribute) * Add derived element type |
| **Representation format** | * Remove existing representation | * Add new representation |
| **Accessibility** | * Restrict permissions | * Relax permissions |

Besides the types of changes listed in the table above, there are some special considerations when to introduce a new version or not:

*Change in quality of service (QoS).* When a change affects qualities of service, such as response time and availability, the API version might need to be change as the change in the QoS might break the applications that use the API.

*Changing assumed domain of values in the output message.* Applications that use the API may have an expectation about the domain of values of certain fields in the response messages. For instance, the API might only return products of a certain category, or specific payment methods. If the API starts returning values outside that assumed domain, it will likely have an adverse effect on the existing consumers. In this case, consider introducing a new API version.

As part of the standards introduced with the versioning of the APIs, there should be well defined rules of impact of the change of version numbers:

* Major versions: introduce a change in the structure of the API that require the user of the API to adopt the interface on the consumer side
* Minor versions: introduce changes to the API that do not require the API user to change, or contain bug fixes.

Using these rules allows to define the following naming conventions:

* Specify the version with a “v“ prefix.
* Move it all the way to the left in the URL so that it has the highest scope.
* Use a simple ordinal number, e.g., “v1“,“v2“, and so on.
* Avoid using dot notation, e.g., “v1.2“

http://www.example.org/v1/customer/1234

http://www.example.org/v2/product/1234

In general, avoid more than **2** versions of the API at the same time. Declare the old version of the service *deprecated,* update the developer’s portal and notify the consumers. Allow time to for service customers to implement and test against the new service interface and functionality before decommissioning the old service.

## **RAML Best Practices**

1. <http://www.iana.org/assignments/media-types/> [↑](#footnote-ref-1)