**API Pattern**

**1 Description**

The API pattern is used to identify the common ways in which a client application will operate with an API Endpoint. We will default to exposing our API’s as RESTful endpoints.

REST makes it easy to create a great developer experience. REST imposes a couple of architectural constraints on your API design.

In general, as we approach the concepts, we are thinking of API’s as “API as Product”. This means we need to make the usage of the API as smooth and easily accessible as possible.

The scope of this document is to describe API’s as RESTful, from a synchronous request & response perspective.

**1.1 Pattern Information**

|  |  |
| --- | --- |
| **Tag** | **Description** |
| Pattern Family | API |
| Pattern ID | API |
| Pattern Status | Draft |
| Version | 0.1 |
| Author(s) | Jamie Gunn |
| Sources | Biehl, Matthias. RESTful API Design: Best Practices in API Design with REST (API-University Series Book 3) (p. 25). API-University. Kindle Edition.  <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>  <https://github.com/Microsoft/api-guidelines>  <https://bbvaopen4u.com/en/actualidad/rest-api-what-it-and-what-are-its-advantages-project-development>  <https://www.mulesoft.com/resources/api/restful-api> |

**1.2 Context**

Applications expose their functionality to clients by accepting and processing requests. The ways in which these requests and responses are structured is the reason for this pattern.

**1.3 Problem**

In a typical enterprise, API’s need to handle the integration between all different types of client applications, such as web apps, services, internal and external clients, etc. However, there is a plethora of decisions that need to be made in terms of how to make a proper request. This document will explain the pattern in terms of how JM Family will create a common understanding of what an API Request is and some of the boundaries.

Commonly the developer experience when working with API’s is weak. Specifically, when people generate API’s, typically it is not done with a consistent enterprise pattern governing what is in an API as a principle and what is optional.

This pattern is developed to solve that problem.

**1.4 Forces**

Forces For:

* **API as Product**. This is a pattern that is commonly used throughout the business world to aid interoperability between organizations. In some organizations, the API is literally part of the product offering, such as Salesforce, NetSuite, AWS, etc.
* **Consistency**. Developers in an organization will be doing work in a predictable manner.
* **Scalability.** When developed to guidelines regarding statelessness, following appropriate resource based principles, API’s can scale horizontally as wide as the data sources will support and the appropriate hardware is in place.
* **Maintainability**. Consistency in development approach aids debugging and augmentation.
* **Discoverable**. Integration with well designed RESTful API’s are relatively simple due to the open nature of the protocol.

Forces Against:

* **Complexity.**When designing to a resource based pattern, more care and time is needed to design appropriately.
* **Chatty.** Resource based design can become chatty. The tradeoffs are not always known at design time and sometimes have to be resolved during implementation.

**2.0 Solution**

**2.1 Solution Description**

RESTful APIs. REST or RESTful APIs were designed to take advantage of existing protocols. While REST - or Representational State Transfer - can be used over nearly any protocol, when used for web APIs it typically takes advantage of HTTP. This means that developers have no need to install additional software or libraries when creating a REST API.

One of the key advantages of REST APIs is that they provide a great deal of flexibility. Data is not tied to resources or methods, so REST can handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

We will designing towards REST level 2. <https://martinfowler.com/articles/richardsonMaturityModel.html#level2>

**2.2 Applying the Pattern**

When applying the API pattern, the following should be considered:

* **Client-Server:** This constraint operates on the concept that the client and the server should be separate from each other and allowed to evolve individually.
* **Stateless:** REST APIs are stateless, meaning that calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully.
* **Cache:** Because a stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data.
* **Uniform Interface:** The key to the decoupling client from server is having a uniform interface that allows independent evolution of the application without having the application’s services, or models and actions, tightly coupled to the API layer itself.
* **Layered System:** REST APIs have different layers of their architecture working together to build a hierarchy that helps create a more scalable and modular application.

**2.3 High Level Principles**

* Use **Nouns** to name the resource (no actions)
* Use **Plural** naming convention (Users vs. User)
* HTTP **Verbs** (normally) determine intent
* HTTP **Return Codes** describe results
* Use **headers** to carry state, metadata, authorization

**2.4 API Request**

**2.4.0** Resource Based

In addition to friendly URLs, resources that can be moved or be renamed SHOULD expose a URL that contains a unique stable identifier. It MAY be necessary to interact with the service to obtain a stable URL from the friendly name for the resource.



Where:

* {serviceRoot} – the combination of host (site URL) + the root path to the service
* {collection} – the name of the collection, unabbreviated, pluralized
* {id} – the value of the unique id property. When using the "/" pattern this MUST be the raw string/number/guid value with no quoting but properly escaped to fit in a URL segment

**2.4.1** Usage of HTTP Verbs in the request

**GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.

A successful GET method typically returns HTTP status code 200 (OK). If the resource cannot be found, the method should return 404 (Not Found).  
  
**POST** creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

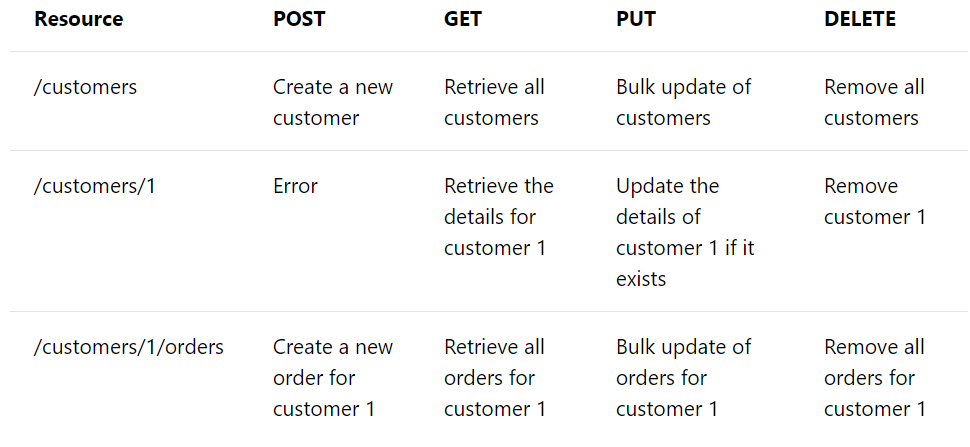
**PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.

If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 200 (OK) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

**DELETE** removes the resource at the specified URI.

If the delete operation is successful, the web server should respond with HTTP status code 204, indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).



**2.4.2** Asynchronous Operations  
  
Sometimes a POST, PUT, PATCH, or DELETE operation might require processing that takes awhile to complete. If you wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:



If the client sends a GET request to this endpoint, the response should contain the current status of the request. Optionally, it could also include an estimated time to completion or a link to cancel the operation.



If the asynchronous operation creates a new resource, the status endpoint should return status code 303 after the operation completes. In the 303 response, include a Location header that gives the URI of the new resource:



**2.4.3** Filtering, Paginating and Sorting

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the /orders URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as /orders?minCost=n. The web API is then responsible for parsing and handling the minCost parameter in the query string and returning the filtered results on the sever side.

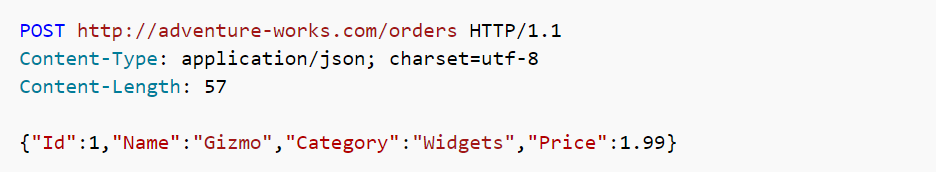
GET requests over collection resources can potentially return a large number of items. You should design an API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:



Also consider imposing an upper limit on the number of items returned, to help prevent Denial of Service attacks. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection.

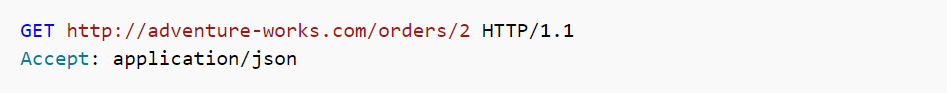
**2.4.4** Media Types

In the HTTP protocol, formats are specified through the use of media types, also called MIME types. For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).

The Content-Type header in a request or response specifies the format of the representation. Here is an example of a POST request that includes JSON data:  
  


If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

A client request can include an Accept header that contains a list of media types the client will accept from the server in the response message. For example:



If the server cannot match any of the media type(s) listed, it should return HTTP status code 406 (Not Acceptable).

**2.4.5** Versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

<http://adventure-works.com/v2/customers/3>

**2.4.6** Security

All API’s must be HTTPS (that is, all inbound calls MUST be HTTPS).

Due to Backwards compatibility reasons, we are supporting the following security processes and protocols:

* 1. SAML
  2. oAuth 1.x and 2
  3. JWT

**(Need more guidance from Security Architect)**

**2.3.5** Standard HTTP Request Headers

| **Header** | **Type** | **Description** |
| --- | --- | --- |
| Authorization | String | Authorization header for the request |
| Date | Date | Timestamp of the request, based on the client's clock, in [RFC 5322](https://tools.ietf.org/html/rfc5322#section-3.3) date and time format. The server SHOULD NOT make any assumptions about the accuracy of the client's clock. This header MAY be included in the request, but MUST be in this format when supplied. Greenwich Mean Time (GMT) MUST be used as the time zone reference for this header when it is provided. For example: Wed, 24 Aug 2016 18:41:30 GMT. Note that GMT is exactly equal to UTC (Coordinated Universal Time) for this purpose. |
| Accept | Content type | The requested content type for the response such as:   * application/xml * text/xml * application/json * text/javascript (for JSONP)   Per the HTTP guidelines, this is just a hint and responses MAY have a different content type, such as a blob fetch where a successful response will just be the blob stream as the payload. For services following OData, the preference order specified in OData SHOULD be followed. |
| Accept-Encoding | Gzip, deflate | REST endpoints SHOULD support GZIP and DEFLATE encoding, when applicable. For very large resources, services MAY ignore and return uncompressed data. |
| Accept-Language | "en", "es", etc. | Specifies the preferred language for the response. Services are not required to support this, but if a service supports localization it MUST do so through the Accept-Language header. |
| Accept-Charset | Charset type like "UTF-8" | Default is UTF-8, but services SHOULD be able to handle ISO-8859-1. |
| Content-Type | Content type | Mime type of request body (PUT/POST/PATCH) |

**2.5 API Response**

**2.5.1 HTTP Headers**

| **Response Header** | **Required** | **Description** |
| --- | --- | --- |
| Content-Type | All responses | The content type |
| Content-Encoding | All responses | GZIP or DEFLATE, as appropriate |

**2.5.2 JSON**

All response should return JSON.

See DevCC for proper JSON format.

**2.5.3 Error Handling**

<https://cloud.google.com/storage/docs/json_api/v1/status-codes>



**2.5.4 HTTP Status Codes**

Use HTTP codes as standard.

<https://cloud.google.com/storage/docs/json_api/v1/status-codes#http-status-and-error-codes>

**2.6 Benefits**

* **Separation between the client and the server**: the REST protocol totally separates the user interface from the server and the data storage. This has some advantages when making developments. For example, it improves the portability of the interface to other types of platforms, it increases the scalability of the projects, and allows the different components of the developments to be evolved independently.
* **Visibility, reliability and scalability**. The separation between client and server has one evident advantage, and that is that each development team can scale the product without too much problem. They can migrate to other servers or make all kinds of changes in the database, provided the data from each request is sent correctly. The separation makes it easier to have the front and the back on different servers, and this makes the apps more flexible to work with.
* **The REST API is always independent of the type of platform or languages**: the REST API always adapts to the type of syntax or platforms being used, which gives considerable freedom when changing or testing new environments within the development. With a REST API you can have PHP, Java, Python or Node.js servers. The only thing is that it is indispensable that the responses to the requests should always take place in the language used for the information exchange, normally XML or JSON.

**2.5 Examples**

* Unknown

**3.0 Architectural Guidance**

**3.1 Rationale**

RESTful API’s are our preferred way of designing & implementing synchronous request / response HTTP Endpoints. API’s fit extremely well for requirements that need synchronous inter-organization integration, web applications, highly scalable transactional systems, versioning, etc.

* **REST is all about simplicity, thanks to HTTP protocols.**
* **REST APIs facilitate client-server communications and architectures.** If it’s RESTful, it’s built on this client-server principle, with round trips between the two passing payloads of information.
* **REST APIs use a single uniform interface.** This simplifies how applications interact with the API by requiring they all interface in the same way, through the same portal. This has advantages and disadvantages; check with your developer to see if this will affect implementation changes down the road.
* **REST is optimized for the web.** Using JSON as its data format makes it compatible with browsers.

Alternatives to RESTful API’s include:

1. Messaging

Messaging is an excellent implementation, but it is geared at asynchronous workloads rather than *synchronous request and response*.

1. Distributed object technologies, such as COM, Corba, EJB’s, Java RMI

These are binary and proprietary based technologies. They lack openness and ease of integration. They tend to be require the client side to understand the protocol and format of the data prior to interoperating.

1. gRPC

Need more research. This appears to be geared at streaming.

**3.2 Related Patterns**

* [Gateway Aggregation](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation)
* [Gateway Offloading](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading)
* [Gateway Routing](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-routing)
* [Backends for Frontends pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends)
* The [Microservice architecture pattern](http://microservices.io/patterns/microservices.html) creates the need for this pattern.
* The API gateway must use either the [Client-side Discovery pattern](http://microservices.io/patterns/client-side-discovery.html) or [Server-side Discovery pattern](http://microservices.io/patterns/server-side-discovery.html) to route requests to available service instances.
* The API Gateway may authenticate the user and pass an [Access Token](http://microservices.io/patterns/security/access-token.html) containing information about the user to the services
* An API Gateway will use a [Circuit Breaker](http://microservices.io/patterns/reliability/circuit-breaker.html) to invoke services
* An API gateway often implements the [API Composition pattern](http://microservices.io/patterns/data/api-composition.html)

**3.3 Known Uses**

* Need more research.