
Resource API

A client application consumes or manipulates text, images, documents, or other media files managed by a remote system.

How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimize the need for domain-specific APIs?

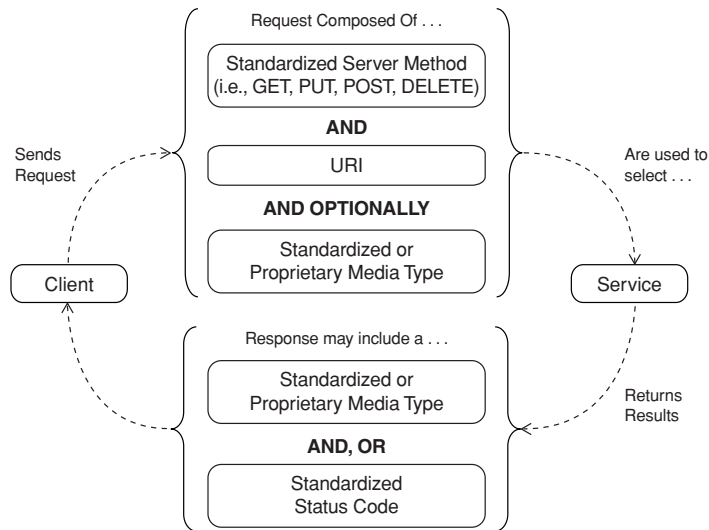
HTTP makes it relatively easy for clients to reuse logic found in remote procedures while insulating them from underlying technologies. One way to invoke these procedures is to have clients send messages that not only identify the procedure to execute, but also include elements that correspond to the procedure's arguments. When these messages are received at the web server, an underlying service framework typically invokes a procedure given the name found in the message. Web services that use this API style are relatively easy to implement and use thanks to modern development tools, but the messages are tightly coupled to the procedures. If the need ever arises to add, change, or remove procedure arguments, then the related message structures must be updated, and the client's *Proxy* (168) will probably have to be regenerated as well. Developers could instead create a service API that doesn't tie messages directly to remote procedures. These messages identify a topic of interest, an event, or a logical command rather than a specific procedure name that is internal to the receiving system. When they are received at the web server, a service framework or custom code uses the content found in the message to determine what procedure should be invoked. While this loosens the dependencies between messages and remote procedures, other factors should be considered.

Many web services use messages to form their own domain-specific API. These messages incorporate common logical commands like Create, Read (i.e., Get), Update, or Delete. This CRUD approach, however, can lead to a proliferation of messages, even in relatively small problem domains. Consider, for example, a set of services that manages company and contact information. In this scenario, the client developer would have to use eight or more distinct messages, one for each combination of a domain entity (i.e., company or contact) and CRUD operation. An API like this might include messages like "CreateCompany", "GetCompany", and so forth. The service owner would also have to create response messages for the various service outcomes (e.g., "CreateCompanyResp", "GetCompanyResp", etc.). Rather than creating a domain-

specific API like this, one could leverage the standards defined in the HTTP specification.

Assign all procedures, instances of domain data, and files a URI. Leverage HTTP as a complete application protocol to define standard service behaviors. Exchange information by taking advantage of standardized media types and status codes when possible.

Resource API



Services that have *Resource APIs* use the requested URI and HTTP server methods issued by the client along with the submitted or requested media type to determine the client’s intent. These services often adhere to the principles of Representational State Transfer (REST), but not every *Resource API* can be considered RESTful. A quick review of the REST architectural style is therefore provided, to help you better understand this pattern.

Resource APIs, as the name implies, provide access to resources. A resource may be a text file, a media file (e.g., images, videos, audio), a specific row in a database table, a collection of related data (e.g., products), a logical transaction, a queue, a downloadable program, a business process (i.e., procedure)—almost anything. Clients manipulate the state of these resources through representations. A database table row may, for example, be represented as XHTML, XML, or JSON. Representations typically capture the current or intended state

of a resource. A client that receives a representation from a service is usually acquiring the most recent state of that resource. When clients send representations to services, their intent is usually to alter the state of a resource. Resource state is transferred when representations are exchanged between clients and services. This is how the term "Representational State Transfer" was derived. A sample representation that uses Atom Publishing Protocol (APP) is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Flight Confirmation</title>
  <id>http://bargainair.net/itineraries/JK3N76</id>
  <summary>BargainAir flight confirmation - August 12, 2010
    - New York to London
  </summary>
  <link rel="depart-trip"
    href="http://bargainair.net/itineraries/JK3N76/123"/>
  <link rel="return-trip"
    href="http://bargainair.net/itineraries/JK3N76/456"/>
  <updated>2010-06-09T16:57:02Z</updated>
</entry>
```

This representation shows how a client application may discover the services that return detailed information about a customer's flights. In this case, information on the customer's departing and returning flights can be discovered by following the URIs found in the href tags. Each URI is a logical address to which clients may send requests in order to invoke a service. These URIs may be permanent or transitory addresses that clients may reference, save, bookmark, and share. A business person could, for example, easily forward the URIs shown above to her accounting department for approval. The following listing demonstrates how URIs can be used to identify other resources.

```
http://www.acmeCorp.org/products
```

```
http://www.acmeCorp.org/products/Model205
```

```
http://www.acmeCorp.org/orders/b0d891d1-3ddd-4d1a-a90f-b3138388ae1f
```

The first URI shows how a client might access a collection of products. The second references a specific product, and the last provides access to a customer's order. URI schemes like these make it easy to add new services for different resources as the need arises. For example, acmeCorp.org can easily add a new "store finder service" by receiving requests at a new URI.

A one-to-many relationship may exist between resources and URIs. That is, a resource may have many addresses; much like a person can have a proper name and nicknames. A single URI, however, should only be used to refer to a single logical resource. This provides clients the means to uniquely identify and

access specific resources or resource collections. Since each URI refers to a single resource or collection of resources, services can often be added, changed, or removed with minimal impact to other services.

Service Contracts for *Resource APIs* are composed of an application protocol (i.e., HTTP), the media types consumed or produced by the service, the service’s status codes, and the URIs and URI schemes or patterns used to identify resources.

Resource APIs use HTTP as an application protocol that prescribes several standard service behaviors. It is expected that all web servers implement the standard HTTP methods, and that all *Resource APIs* will respond to these methods as follows.

- PUT is used to create or update resources.
- GET is used to retrieve a resource representation.
- DELETE removes a resource.
- The behavior of POST varies. It can be used to create a subordinate of the target resource identified in the client’s request, or for “nonstandard behaviors” where other methods aren’t a good fit. For example, a mutual fund service might accept requests to execute functions like “Exchange Funds”, “Sell X Shares”, and “Sell Dollar Amount” through POST. This method can also be used as a workaround when PUT and DELETE are disabled on the web server or blocked at the firewall. In this situation, the behaviors that would normally be carried out by other methods are tunneled through POST. Many REST advocates argue against tunneling through POST since it tends to obfuscate the purpose of the request. POSTed requests also can’t be cached by intermediaries.
- The OPTIONS verb may be used to discover what HTTP methods are supported at the target URI.
- HEAD is used to acquire metadata about the media types exchanged at a URI. It is similar to GET except that a representation is not returned.
- While the core methods are static, extensions to HTTP have been added (e.g., WebDAV).

Resource APIs should respond to the methods listed above in the prescribed manner. So, if a client issues a GET to `acmeCorp.org/products`, it should expect the service to execute a `Get Products` function. Since the semantics are predetermined by HTTP, clients don’t have to learn a specialized API. However, they must still

know what methods may be used with each URI and when to use each method. Additionally, it's still up to the service developer to implement the function according to the standards. Since the standard behaviors of PUT, GET, and DELETE map roughly to the CRUD paradigm, some believe that *Resource APIs* should only be used for CRUD use cases. This, however, is an incorrect assessment because POST can be used to execute behaviors that don't map well to CRUD.

The HTTP specification also identifies which methods should be “safe” and which should be “idempotent.” Safe operations are supposed to have no side effects. That is, they should not trigger write operations (i.e., creates, updates, or deletes). GET, HEAD, and OPTIONS are supposed to be implemented as safe operations. Idempotence means that no matter how many times a procedure is invoked with the same data, the same results should occur. GET, HEAD, PUT, DELETE, and OPTIONS are idempotent. POST, on the other hand, is not. Therefore, if a client repeatedly POSTs contact information to the same URI one should expect that information will be written each time. There are times when POST should exhibit idempotent behavior. For example, if a client sends the same order over and over again, the client shouldn't have to worry about duplicate orders. This means that the service must differentiate one POST from another. The easiest approach is to have the client insert a unique key (i.e., identifier) into the request that is examined by the service before executing its main logic. If the service finds that it has already processed a request with the identifier, it can reject the new request. The problem is ensuring that these identifiers will indeed be unique. Another approach is to have the client query a service to retrieve a unique URI that may be used exclusively for the subsequent POST. This pattern is known as *Post-Once-Exactly* [Nottingham, Marc].

Resource APIs usually take advantage of standard HTTP status codes as a mechanism to provide results to the client. For example, rather than returning an XML message, the service could return an HTTP code of 200 to indicate a request has succeeded. If a resource has moved, the service could return a 301 code, and if the client sends a malformed message, the service might return a 400 error code to inform the client that their request isn't understood and can't be processed. A long list of codes that cover common scenarios may be leveraged (re: www.w3.org/Protocols/rfc2616/rfc2616-sec10.html). These codes enable clients and services to communicate in a standard way, and can also help to optimize network utilization because minimal data is returned to the client.

The media types consumed or produced by the service are the most explicit part of a *Resource API* contract. These define the required data structures (i.e., representations using formats like XML or JSON), character encodings (e.g., Unicode, ASCII), rules for parsing data, and standards for linking to other resources. Media types can be altered or extended as long as these data struc-

tures, character encodings, rules for parsing, and standards for linking don’t incur breaking changes in clients (for more on breaking changes, see the section *What Causes Breaking Changes?* in Chapter 7). This means that services that use this API style shouldn’t suddenly switch to using types that aren’t understood by their clients. Indeed, unilateral changes cannot be allowed in “enterprise business applications” where changes must generally be coordinated. Therefore, clients must have foreknowledge of the service’s media types and must also know how to process them. All parties should also leverage standardized types (e.g., MIMEs) or use common vocabularies (e.g., Atom Publishing Protocol, Microformats) when possible. However, if existing standardized types or vocabularies cannot be used, then the parties may develop their own proprietary types and vocabularies. Of course, this may limit the audience for the service.

Not every client will understand the media types used by a service. Some clients must delegate the handling of specific media types to specialized agents. A service could, for example, wrap a representation in an “execution engine” (e.g., Java applet, JavaScript) that the client is able to host and run. Other media types may require the use of a plug-in that has specific knowledge of the media type’s processing model. In this case, the client must have installed the plug-in and have granted it appropriate execution privileges.

Considerations

Developers who create Resource APIs should consider the following issues.

- **Use with disparate clients:** *Resource APIs* are a great choice when you have a wide mix of clients. Web browsers, feed readers, syndication services, web aggregators, microblogs, mashups, AJAX controls, and mobile applications are all natural clients for this style. This API style can also be used in enterprise integration and workflow scenarios.

Resource APIs are especially effective when large documents and messages or binary files must be exchanged. The advantage is that media types like these need not be wrapped in message envelopes that require clients and services to use additional protocols (e.g., like the Message Transmission Optimization Mechanism or MTOM) to attach or detach the payload to or from the message.

- **Addressability:** Resource APIs make it easy for clients to save and share links to services. However, service owners must first decide whether or not data should be directly addressable. The problem is that URIs often provide malicious users with clues on how to mine an organization for information.

These “hackable” URIs make it easy for anyone to understand the meaning of each URI segment, and to replace the content of specific segments in an attempt to gain access to information that perhaps he shouldn’t see. A URI may, for example, include customer account numbers as URI segments. Such schemes are easy to exploit. The service owner could prevent mischievous users from hacking these URIs by replacing simple account numbers with meaningless UUIDs. This, however, is not enough. Service owners should always implement the appropriate authentication and authorization logic to confirm the identity of the caller and to constrain what each caller can do. Nevertheless, some may consider direct resource addressability to be too much of a security risk, even when the proper authentication and authorization protections have been put into place. The alternative is to funnel all requests through a *Message API* (27) or *RPC API* (18).

- **Code generation of service connectors:** Client developers that use *Resource APIs* often can’t take advantage of code generation tools. This is partially due to the fact that most *Resource API* designers prefer not to offer *Service Descriptors* (175). For those who appreciate code generation of client-side *Service Connectors* (168), services that have *Message APIs* (27) or *RPC APIs* (18) may be a better option.
- **Achieving asynchrony:** Resource-oriented services typically use the *Request/Response* pattern (54), but can also use the *Request/Acknowledge* interaction pattern (59). With this pattern the request is not processed when it is received. Instead, the service forwards the request to an asynchronous background process and returns an acknowledgment (i.e., HTTP code 202). By separating the message receipt from the time it is processed, the system is better able to handle unanticipated spikes in load and control the rate at which requests are processed.
- **How to avoid blocking:** Clients that use this API style need not block after sending a message. Rather, they may use an *Asynchronous Response Handler* (184) to enable the client to perform other useful work as soon as a message is sent.
- **Ability to support client preferences:** *Resource APIs* often provide multiple representations of the same logical resource. Rather than using a different URI for each, you can use *Media Type Negotiation* (70) to enable clients to indicate their preferences.

- **Late binding:** Once a service has processed a request, clients often need to call additional services in specific sequences. For example, a client that calls an “Order Creation” service will frequently call “Order Update”, “Order Cancel”, and “Order Status” services thereafter. The *Linked Service* pattern (77) enables clients to discover related services that may be called after receiving a service response.
- **Ability to leverage commodity caching technologies:** This API style leverages commodity caching technologies designed specifically with HTTP in mind. If, for example, a client requests a product that hasn’t changed within the past day, and information on that product can be found in a Reverse Proxy, then the cached representation will be returned and service execution can be bypassed. This reduces the load on the Origin Server, especially in cases where the service would have queried a database or performed a CPU- or memory-intensive computation. *Resource APIs* are therefore well suited for “read scenarios.” Clients can also implement caches that may be checked for matching representations before sending messages to services. Figure 2.2 illustrates the possibilities.

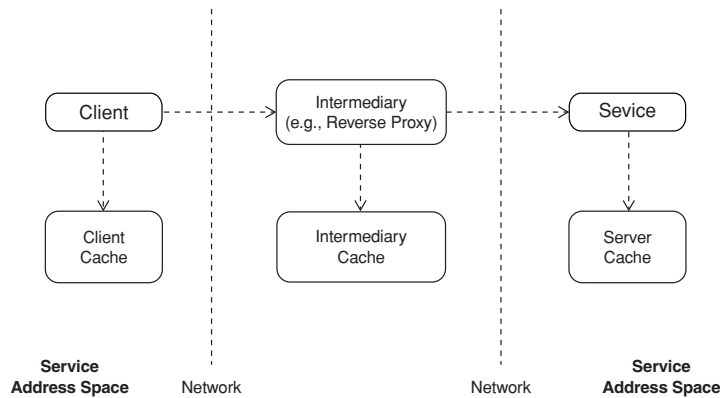


Figure 2.2 *Representations from Resource APIs can be easily served up from client, intermediary, and server caches. Whenever a representation can be provided from a cache close to the client, associated performance costs related to network latency can be avoided, and server loads can be minimized.*

- **Resource APIs and REST:** Earlier in this pattern I said that *Resource APIs* often adhere to the principles of REST, but not every *Resource API* can be considered RESTful. REST is an architectural style defined by several constraints [Fielding]. These include the following.

- **Client/server:** All web services, regardless of the API, meet this constraint.
- **Stateless:** Not every *Resource API* is stateless. Some developers are less concerned about super scalability and opt to create web services that maintain client state across multiple calls. For more information, see the section Design Considerations for Web Service Implementation in Chapter 5.
- **Cacheable responses:** Most *Resource APIs* can leverage commodity caching technologies. Whether you should or shouldn't cache response data is another discussion altogether. It should be noted that responses from *RPC APIs* (18) and *Message APIs* (27) can also be cached by intermediaries, but specialized infrastructures must often be used.
- **Uniform interface:** This is a fairly complex constraint. It suggests that a uniform interface be used between all components (i.e., clients, intermediaries, and servers). For *Resource APIs*, this uniform interface is defined through the HTTP specification. REST practitioners suggest that *Resource APIs* must use the server methods (i.e., GET, PUT, POST, DELETE) exactly as prescribed by the specification. Therefore, an API that relies on POST for all logical operations is not RESTful.

This constraint also includes four architectural constraints. The first is that all resources must be uniquely identified. This occurs through URIs. The second and third subconstraints state that resources should be manipulated through representations, and that messages must be self-descriptive. *Resource APIs* usually meet these requirements. The last constraint says that hypermedia should be the engine of application state. In short, this means that hyperlinks should be used to guide client applications through various workflow state transitions. Many *Resource APIs* do not meet this requirement. For more information on this topic, see the *Linked Services* pattern (77).

- **Layered system:** All web services, regardless of the API style, meet this constraint if we look at them from the perspective of the Open Systems Interconnection (OSI) Model.
- **Code on demand:** This constraint states that client applications can be extended if they are allowed to download and execute scripts or plug-ins that support the media type provided by the service. Adherence to this constraint is therefore determined by the client rather than the API.

Example: *A Resource API Implemented in Java and JAX-RS*

This example shows a partial implementation of a *Resource API* for music queries. The class named `MusicGenreController` is a *Service Controller* (85) that maps client requests to a specific request handler. This handler, named `GetArtistsInGenre`, accepts a music genre and the starting characters of an artist's name as input. The controller uses the *MusicSearch Command* [GoF] to execute a search for artists. The *Artists Data Transfer Object* (94) produces a JSON representation that is returned to the client.

```
@Path("/genre")
public class MusicGenreController {

    private String NEXT_URI =
        "http://www.acmeCorp.org/MusicService/artists";

    @GET
    @Path("/{genreName}/{artistNameStartsWith}")
    @ProducesMime("application/json")

    public JAXBElement<Artists> GetArtistsInGenre(
        @PathParam("genreName") String genreName,
        @PathParam("artistNameStartsWith") String startsWith){

        MusicSearch search = new MusicSearch(NEXT_URI);

        Artists artists =
            search.getArtists(genreName, startsWith);

        return new JAXBElement<Artists>(
            new QName("Artists"), Artists.class, artists);
    }
}
```

This service enables clients to issue GET requests that look like this:

```
GET /MusicService/artists/genre/rock/roll HTTP/1.1
Host: acmeCorp.org
```

The representation for this resource might look something like this:

```
{"@StartsWith":"roll","@Genre":"rock",
 "Artist":
 [
 {"Name":"Rolling Stones",
  "URI":"http://www.acmeCorp.org/MusicService/artists/Rolling&Stones"},
 {"Name":"Rollins Band",
  "URI":"http://www.acmeCorp.org/MusicService/artists/Rollins&Band"}
 ]}
```

Example: *Procedure Invocation*

Flickr, the popular image and video hosting web site, provides several APIs (re: www.flickr.com/services/api/). The following are examples of *Resource APIs* that enable clients to upload or replace standard binary photos.

```
http://api.flickr.com/services/upload/
```

```
http://api.flickr.com/services/replace/
```

Clients can also invoke procedures by issuing an HTTP GET or POST to URIs that look like this:

```
http://api.flickr.com/services/rest/?method=X&arg1=Y
```

This cannot be considered a RESTful API because it doesn't utilize the uniform interface of HTTP. In other words, a client could invoke an operation that has side effects (e.g., a routine that writes to a database) without using PUT or DELETE. Regardless, this API provides a simple way for client developers to execute procedures. It's interesting to note that a specific invocation of a procedure, inclusive of its arguments (as specified by the query strings), can be saved or bookmarked.

Example: *Conditional Queries and Updates*

Clients can often help to minimize latency by specifying that representations should only be returned if something has changed recently. Clients may, for example, use standard HTTP header fields like *If-Modified-Since* to tell the service to only return a response if the requested entity was modified after the time specified. Here's an example of a client request:

```
GET /products/pricelist HTTP/1.1  
Host: acmeCorp.org  
If-Modified-Since: Fri, 30 Sep 2010 18:00:00 GMT
```

In this case, the service should only return a representation if the price list has changed since the date indicated in the *If-Modified-Since* header. If the resource has not been modified, the service may return an HTTP code of 304 to indicate that nothing has changed. This helps to optimize network bandwidth because a full representation isn't sent to the client. However, server load may not be significantly reduced because the service must still execute the necessary logic to retrieve the data and evaluate whether the resource has changed since the date specified in the client's header.

The service owner has several options to reduce server load. One option is to use the *Service Interceptor* pattern (195). An inbound interceptor may, for example, be configured to check to see if the requested data can be found in a distributed memory cache that is shared across web servers. If the requested information can be found in this cache, then the information may be returned directly from the interceptor, and the request handler will not be called. Server utilization is optimized somewhat because the handler that would have presumably queried the database and formatted a response is not executed. However, the server still receives the request and, as a result, its load is higher than it might otherwise be. Server load can be reduced by configuring Reverse Proxies to cache responses. In this case, the proxy evaluates the client’s criterion against its cache and will return a representation if the criterion is met. Regardless of the caching approach, the service owner must determine how long data may be kept in any cache before it is considered stale.

The Lost-Update Problem can be prevented through a similar conditional statement.

```
PUT /products/pricelist/123 HTTP/1.1
Host: acmeCorp.org
If-Unmodified-Since: Fri, 30 Sep 2011 18:00:00 GMT
```

If requests are sent directly to a service that has no interceptors (including Reverse Proxies), then the request will only be processed if the service determines that the resource has not been modified since the date provided in the client’s request. Otherwise, the service will return a 412 status code to indicate that a “precondition” has failed. In this case, the precondition is the modification date. Again, server load can be reduced by using *Service Interceptors* (195) that leverage distributed caches, or with Reverse Proxies.
