

HTTP API

jim stafford

Fall 2024 v2020-04-25: Built: 2024-11-19 21:31 EST

Table of Contents

1. Introduction	1
1.1. Goals	1
1.2. Objectives	1
2. World Wide Web (WWW)	2
2.1. Example WWW Information System	2
3. REST	3
3.1. HATEOAS	3
3.2. Clients Dynamically Discover State	3
3.3. Static Interface Contracts	4
3.4. Internet Scale	4
3.5. How RESTful?	4
3.6. Buzzword Association	5
3.7. REST-like or HTTP-based	5
3.8. Richardson Maturity Model (RMM)	5
3.9. "REST-like"/"HTTP-based" APIs	6
3.10. Uncommon REST Features Adopted	6
4. RMM Level 2 APIs	7
5. HTTP Protocol Embraced	8
6. Resource	9
6.1. Nested Resources	9
7. Uniform Resource Identifiers (URIs)	10
7.1. Related URI Terms	10
7.2. URI Generic Syntax	11
7.3. URI Component Examples	11
7.4. URI Characters and Delimiters	12
7.5. URI Percent Encoding	12
7.6. URI Case Sensitivity	13
7.7. URI Reference	13
7.8. URI Reference Terms	13
7.9. URI Naming Conventions	14
7.10. URI Variables	15
8. Methods	16
8.1. Additional HTTP Methods	16
9. Method Safety	17
9.1. Safe and Unsafe Methods	17
9.2. Violating Method Safety	17
10. Idempotent	19
10.1. Idempotent and non-Idempotent Methods	19

11. Response Status Codes	20
11.1. Common Response Status Codes.....	20
12. Representations	21
12.1. Content Type Headers.....	21
13. Links	23
14. Summary	24

Chapter 1. Introduction

1.1. Goals

The student will learn:

- how the WWW defined an information system capable of implementing system APIs
- identify key differences between a truly RESTful API and REST-like or HTTP-based APIs
- how systems and some actions are broken down into resources
- how web interactions are targeted at resources
- standard HTTP methods and the importance to use them as intended against resources
- individual method safety requirements
- value in creating idempotent methods
- standard HTTP response codes and response code families to respond in specific circumstances

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify API maturity according to the Richardson Maturity Model (RMM)
2. identify resources
3. define a URI for a resource
4. define the proper method for a call against a resource
5. identify safe and unsafe method behavior
6. identify appropriate response code family and value to use in certain circumstances

Chapter 2. World Wide Web (WWW)

The [World Wide Web \(WWW\)](#) is an information system of web resources identified by [Uniform Resource Locators \(URLs\)](#) that can be interlinked via [hypertext](#) and transferred using [Hypertext Transfer Protocol \(HTTP\)](#).^[1] [Web resources](#) started out being documents to be created, downloaded, replaced, and removed but has progressed to being any identifiable thing — whether it be the entity (e.g., person), something related to that entity (e.g., photo), or an action (e.g., change of address).^[2]

2.1. Example WWW Information System

The example information system below is of a standard set of content types, accessed through a standard set of methods, and related through location-independent links using URLs.

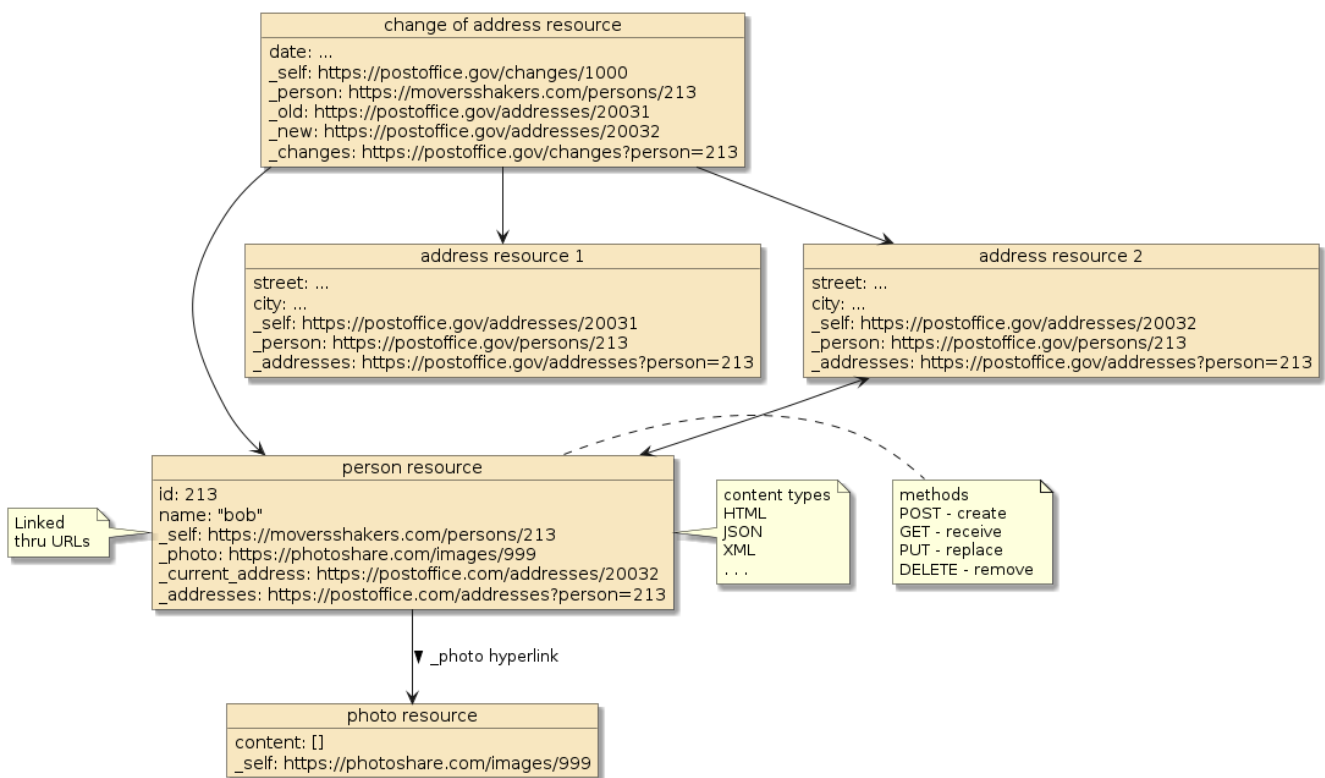


Figure 1. WWW Links Resources thru URLs

[1] ["World Wide Web Wikipedia Page"](#)

[2] ["Web Resource Wikipedia Page"](#)

Chapter 3. REST

Representational State Transfer (REST) is an architectural style for creating web services and web services that conform to this style are considered "Restful" web services ^[1]. REST was defined in 2000 by Roy Fielding in his [doctoral dissertation](#) that was also used to design HTTP 1.1. ^[2] REST relies heavily on the concepts and implementations used in the World Wide Web—which centers around web resources addressable using URIs.

3.1. HATEOAS

At the heart of REST is the notion of hyperlinks to represent state. For example, the presence of a `address_change` link may mean the address of a person can be changed and the client accessing that person representation is authorized to initiate the change. The presence of `current_address` and `addresses` links identifies how the client can obtain the current and past addresses for the person. This is shallow description of what is defined as ["Hypermedia As The Engine Of Application State" \(HATEOAS\)](#).

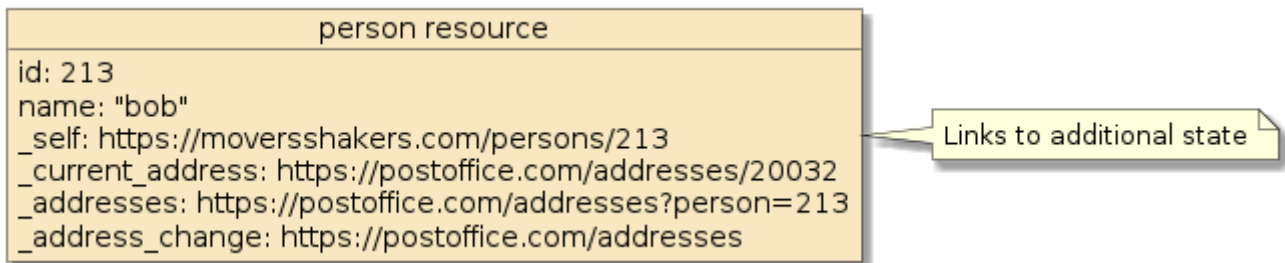


Figure 2. Example of State Represented through Hyperlinks

The interface contract allows clients to dynamically determine current capabilities of a resource and the resource to add capabilities over time.

3.2. Clients Dynamically Discover State

HATEOAS permits the capabilities of client and server to advance independently through the dynamic discovery of links. ^[3]

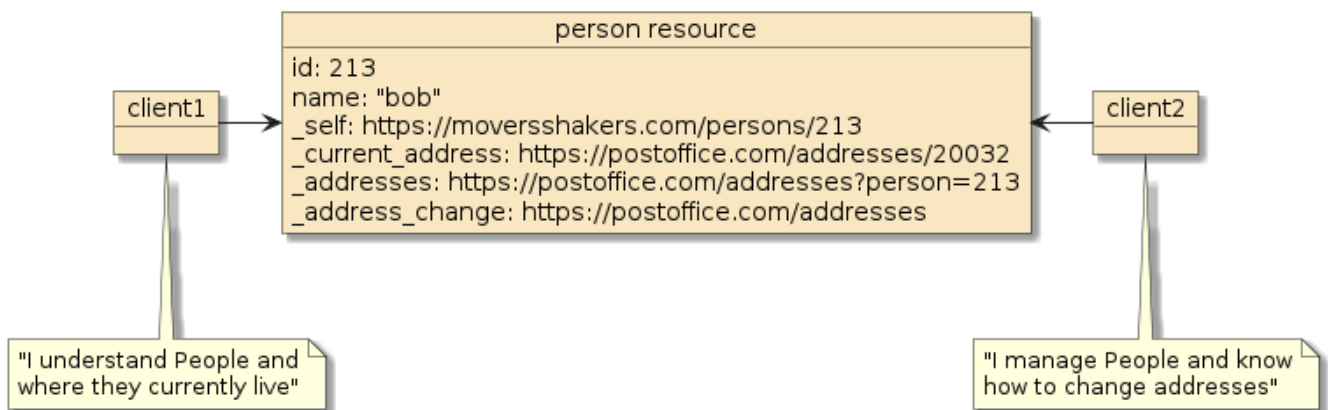


Figure 3. Example of Clients Dynamically Discovering State Represented through Hyperlinks

3.3. Static Interface Contracts

Dynamic discovery differs significantly from remote procedure call (RPC) techniques where static interface contracts are documented in detail to represent a certain level of capability offered by the server and understood by the client. A capability change rollout under the RPC approach may require coordination between all clients involved.

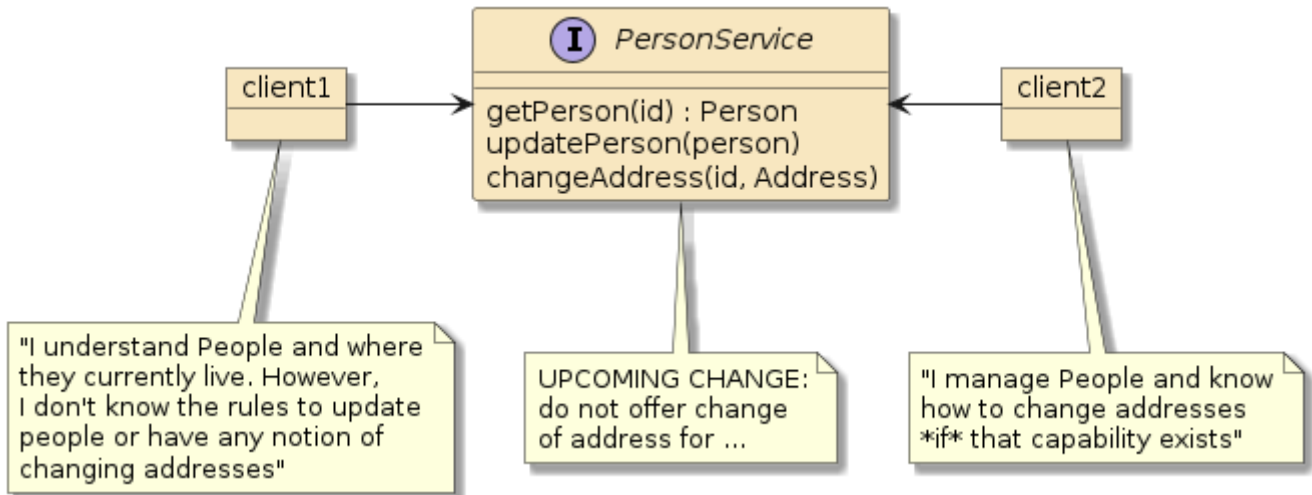


Figure 4. RPC Tight Coupling Example

3.4. Internet Scale

As clients morph from a few, well known sources to millions of lightweight apps running on end-user devices—the need to decouple service capability deployments through dynamic discovery becomes more important. Many features of REST provide this trait.



Do you have control of when clients update?

Design interfaces, clients, and servers with forward and backward compatibility in mind to allow for flexible rollout with minimal downtime.

3.5. How RESTful?

Many of the open and interfacing concepts of the WWW are attractive to today's service interface designers. However, implementing dynamic discovery is difficult—potentially making systems more complex and costly to develop. REST officially contains more than most interface designs use or possibly need to use. This causes developments to take only what they need—and triggers some common questions:

What is your definition of REST?

How RESTful are you?

3.6. Buzzword Association

For many developers and product advertisements eager to get their names associated with a modern and successful buzzword—REST to them is (incorrectly) anything using HTTP that is not SOAP. For others, their version of REST is (still incorrectly) anything that embraces much of the WWW but still lacks the rigor of making the interfaces dynamic through hyperlinks.

This places us in a state where most of the world refers to something as REST and RESTful when what they have is far from the official definition.

3.7. REST-like or HTTP-based

Giving a nod to this situation, we might use a few other terms:

- REST-like
- HTTP-based

Better yet and for more precise clarity of meaning, I like the definitions put forward in the Richardson Maturity Model (RMM).

3.8. Richardson Maturity Model (RMM)

The [Richardson Maturity Model \(RMM\)](#) was developed by Leonard Richardson and breaks down levels of RESTful maturity. ^[4] Some of the old [CORBA](#) and [XML RPC](#) qualify for Level 0 only for the fact they adopt HTTP. However, they tunnel thru many WWW features in spite of using HTTP. Many modern APIs achieve some level of compliance with Levels 1 and 2, but rarely will achieve Level 3. However, that is okay because as you will see in the following sections—there are many worthwhile features in Level 2 without adding the complexity of HATEOAS.

Table 1. Richardson Maturity Model for REST

Level 3	<ul style="list-style-type: none">• using Hypermedia Controls i.e., the basis for Roy Fielding’s definition of REST• dynamic discovery of state and capabilities thru hyperlinks
Level 2	<ul style="list-style-type: none">• using HTTP Methods i.e., handle similar situations in the same way• standardized methods and status codes• publicly expose method performed and status responses to better enable communication infrastructure support
Level 1	<ul style="list-style-type: none">• using Resources i.e., divide and conquer• e.g., rather than a single aggregate for endpoint calls, make explicit reference to lower-level targets
Level 0	<ul style="list-style-type: none">• using HTTP soley as a transport• e.g., CORBA tunneled all calls through HTTP POST

3.9. "REST-like"/"HTTP-based" APIs

Common "REST-like" or "HTTP-based" APIs are normally on a trajectory to strive for RMM Level 2 and are based on a few main principals included within the definition of REST.

- HTTP Protocol
- Resources
- URIs
- Standard HTTP Method and Status Code Vocabulary
- Standard Content Types for Representations

3.10. Uncommon REST Features Adopted

Links are used somewhat. However, they are rarely used in an opaque manner, rarely used within payloads, and rarely used with dynamic discovery. Clients commonly know the resources they are communicating with ahead of time and build URIs to those resources based on exposed details of the API and IDs returned in earlier responses. That is technically not a RESTful way to do things.

[1] ["Representational state transfer" — Wikipedia Page](#)

[2] ["Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation"](#), Roy Thomas Fielding, University of California, Irvine, 2000

[3] ["HATEOUS Wikipedia Page"](#)

[4] ["Richardson Maturity Model"](#), Martin Fowler, 2010

Chapter 4. RMM Level 2 APIs

Although I will commonly hear projects state that they implement a "REST" interface (and sometimes use it as "HTTP without SOAP"), I have rarely found a project that strives for dynamic discovery of resource capabilities as depicted by Roy Fielding and categorized by RMM Level 3.

These APIs try to make the most of HTTP and the WWW, thus at least making the term "HTTP-based" appropriate and RMM-level 2 a more accurate description. Acknowledging that there is technically one definition of REST and very few attempting to (or needing to) achieve it—I will be targeting RMM Level 2 for the web service interfaces developed in this course and will generically refer to them as "APIs".

At this point lets cover some of the key points of a RMM Level 2 API that I will be covering as a part of the course.

Chapter 5. HTTP Protocol Embraced

Various communications protocols have been transport agnostic. If you are old enough to remember [SOAP](#), you will have seen references to it being mapped to protocols other than HTTP (e.g., SOAP over JMS) and its use of HTTP lacked any leverage of WWW HTTP capabilities.

For SOAP and many other RPC protocols operating over HTTP—communication was tunnelled through HTTP POST messages, bypassing investments made in the existing and robust WWW infrastructure. For example, many requests for the same status of the same resource tunnelled through POST messages would need to be answered again-and-again by the service. To fully leverage HTTP client-side and server-side caches, an alternative approach of exposing the status as a GET of a resource would save the responding service a lot of unnecessary work and speed up client.

REST communication technically does not exist outside of the HTTP transport protocol. Everything is expressed within the context of HTTP, leveraging the investment into the world's largest information system.

Chapter 6. Resource

By the time APIs reach RMM Level 1 compliance, service domains have been broken down into key areas, known as resources. These are largely noun-based (e.g., Documents, People, Companies), lower-level properties, or relationships. However, they go on to include actions or a long-running activity to be able to initiate them, monitor their status, and possibly perform some type of control.

Nearly anything can be made into a resource. HTTP has a limited number of methods but can have an unlimited number of resources. Some examples could be:

- products
- categories
- customers
- todos

6.1. Nested Resources

Resources can be nested under parent or related resources.

- categories/{id}
- categories/{id}/products
- todos/{name}
- todos/{name}/items

Chapter 7. Uniform Resource Identifiers (URIs)

Resources are identified using [Uniform Resource Identifier \(URIs\)](#).

A URI is a compact sequence of characters that identifies an abstract or physical resource. ^[1]

— URI: Generic Syntax RFC Abstract 2005

URIs have a generic syntax composed of several components and are specialized by individual schemes (e.g., http, mailto, urn). The precise generic URI and scheme-specific rules guarantee uniformity of addresses.

Example URIs

```
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6 ①  
mailto:joe@example.com?cc=bob@example.com&body=hello ②  
urn:isbn:0-395-36341-1 ③
```

① example URL URI

② example email URI ^[2]

③ example URN URI; "isbn" is a URN namespace ^[1]

7.1. Related URI Terms

There are a few terms commonly associated with URI.

Uniform Resource Locator (URL)

URLs are a subset of URIs that provide a means to locate a specific resource by specifying primary address mechanism (e.g., network location). ^[1]

Uniform Resource Name (URN)

URNs are used to identify resources without location information. They are a particular URI scheme. One common use of a URN is to define an XML namespace. e.g., `<core xmlns="urn:activemq:core">`.

URI reference

legal way to specify a full or relative URI

Base URI

leading components of the URI that form a base for additional layers of the tree to be appended

7.2. URI Generic Syntax

URI components are listed in hierarchical significance — from left to right — allowing for scheme-independent references to be made between resources in the hierarchy. The generic URI syntax and components are as follows: ^[3]

Generic URI components ^[3]

```
URI = scheme:[//authority]path[?query][#fragment]
```

The authority component breaks down into subcomponents as follows:

Authority Subcomponents ^[3]

```
authority = [userinfo@]host[:port]
```

Table 2. Generic URI Components

Scheme	sequence of characters, beginning with a letter followed by letters, digits, plus (+), period, or hyphen(-)
Authority	naming authority responsible for the remainder of the URI
User	how to gain access to the resource (e.g., username) - rare, authentication use deprecated
Host	case-insensitive DNS name or IP address
Port	port number to access authority/host
Path	identifies a resource within the scope of a naming authority. Terminated by the first question mark ("?"), pound sign("#"), or end of URI. When the authority is present, the path must begin with a slash ("/") character
Query	indicated with first question mark ("?") and ends with pound sign("#") or end of URI
Fragment	indicated with a pound("#") character and ends with end of URI

7.3. URI Component Examples

The following shows the earlier URI examples broken down into components.

Example URL URI Components

```
          -- authority                                fragment --
          /                                          \
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
\
-- scheme      \ -- path
```

Path cannot begin with the two slash ("/") character string when the authority is not present.

Example mailto URI Components

```
      -- path
      /
mailto:joe@example.com?cc=bob@example.com&body=hello
\
-- scheme          \
                   -- query
```

Example URN URI Components

```
-- scheme
/
urn:isbn:0-395-36341-1
\
  -- path
```

7.4. URI Characters and Delimiters

URI characters are encoded using [UTF-8](#). Component delimiters are slash ("/"), question mark ("?"), and pound sign("#"). Many of the other special characters are reserved for use in delimiting the sub-components.

Reserved Generic URI Delimiter Characters

```
: / @ [ ] ? ①
```

① square brackets("[]") are used to surround newer (e.g., IPv6) network addresses

Reserved Sub-delimiter Characters

```
! $ & ' ( ) * + , ; =
```

Unreserved URI Characters

```
alpha(A-Z,a-z), digit (0-9), dash(-), period(.), underscore(_), tilde(~)
```

7.5. URI Percent Encoding

(Case-insensitive) Percent encoding is used to represent characters reserved for delimiters or other purposes (e.g., %x2f and %X2F both represent slash("/") character). Unreserved characters should not be encoded.

Example Percent Encoding

```
https://www.google.com/search?q=this+%2F+that ①
```

① slash("/") character is Percent Encoded as %2F

7.6. URI Case Sensitivity

Generic components like scheme and authority are case-insensitive but normalize to lowercase. Other components of the URI are assumed to be case-sensitive.

Example Case Sensitivity

```
HTTPS://GITHUB.COM/SPRING-PROJECTS/SPRING-BOOT ①  
https://github.com/SPRING-PROJECTS/SPRING-BOOT ②
```

① value pasted into browser

② value normalized by browser

7.7. URI Reference

Many times we need to reference a target URI and do so without specifying the complete URI. A URI reference can be the full target URI or a relative reference. A relative reference allows for a set of resources to reference one another without specifying a scheme or upper parts of the path. This also allows entire resource trees to be relocated without having to change relative references between them.

7.8. URI Reference Terms

target uri

the URI being referenced

Example Target URI

```
https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

network-path reference

relative reference starting with two slashes ("//"). My guess is that this would be useful in expressing a URI to forward to without wishing to express http versus https (i.e., "use the same scheme used to contact me")

Example Network Path Reference

```
//github.com/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

absolute-path reference

relative reference that starts with a slash ("/"). This will be a portion of the URI that our API layer will be well aware of.

Example Absolute Path Reference

```
/spring-projects/spring-boot/blob/master/LICENSE.txt#L6
```

relative-path reference

relative reference that does not start with a slash ("/"). First segment cannot have a ":" — avoid confusion with scheme by prepending a "." to the path. This allows us to express the branch of a tree from a point in the path.

Example Relative Path Reference

```
spring-boot/blob/master/LICENSE.txt#L6  
LICENSE.txt#L6  
../master/LICENSE.txt#L6
```

same-document reference

relative reference that starts with a pound ("#") character, supplying a fragment identifier hosted in the current URI.

Example Same Document Reference

```
#L6
```

base URI

leading components of the URI that form a base for additional layers of the tree to be appended

Example Base URI

```
https://github.com/spring-projects  
/spring-projects
```

7.9. URI Naming Conventions

Although URI specifications do not list path naming conventions and REST promotes opaque URIs — it is a common practice to name resource collections with a URI path that ends in a plural noun. The following are a few example absolute URI path references.

Example Resource Collection URI Absolute Path References

```
/api/products ①  
/api/categories  
/api/customers  
/api/todo_lists
```

① URI paths for resource collections end with a plural noun

Individual resource URIs are identified by an external identifier below the parent resource

collection.

Example Individual Resource Absolute URI Paths

```
/api/products/{productId} ①  
/api/categories/{categoryId}  
/api/customers/{customerId}  
/api/customers/{customerId}/sales
```

① URI paths for individual resources are scoped below parent resource collection URI

Nested resource URIs are commonly expressed as resources below their individual parent.

Example Nested Resource Absolute URI Paths

```
/api/products/{productId}/instructions ①  
/api/categories/{categoryId}/products  
/api/customers/{customerId}/purchases  
/api/todo_lists/{listName}/todo_items
```

① URI paths for resources of parent are commonly nested below parent URI

7.10. URI Variables

The query at the end of the URI path can be used to express optional and mandatory arguments. This is commonly used in queries.

Query Parameter Example

```
http://127.0.0.1:8080/jaxrsInventoryWAR/api/categories?name=&offset=0&limit=0  
name => (null) #has value null  
offset => 0  
limit => 0
```

Nested path path parameters may express mandatory arguments.

Path Parameter Example

```
http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/{id}  
http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/1  
id => 1
```

[1] "Uniform Resource Identifier (URI): Generic Syntax RFC", Network Working Group, Berners-Lee, Fielding, Masinter, 2005

[2] "The 'mailto' URI Scheme", Duerst, Masinter, Zawinski, 2010

[3] URI Wikipedia Page

Chapter 8. Methods

HTTP contains a bounded set of methods that represent the "verbs" of what we are communicating relative to the resource. The bounded set provides a uniform interface across all resources.

There are four primary methods that you will see in most tutorials, examples, and application code.

Table 3. Primary HTTP Methods

GET	obtain a representation of resource using a non-destructive read
POST	create a new resource or tunnel a command to an existing resource
PUT	create a new resource with having a well-known identity or replace existing
DELETE	delete target resource

Example: Get Product ID=1


```
GET http://127.0.0.1:8080/jaxrsInventoryWAR/api/products/1
```

8.1. Additional HTTP Methods

There are two additional methods useful for certain edge conditions implemented by application code.

Table 4. Additional HTTP Methods

HEAD	logically equivalent to a GET without response payload - metadata only. Can provide efficient way to determine if resource exists and potentially last updated
PATCH	partial replace. Similar to PUT, but indicates payload provided does not represent the entire resource and may be represented as instructions of modifications to make. Useful hint for intermediate caches



The following [post](#) provides a good starting point to understanding the role of PATCH and the role of the JSON Patch and JSON Merge Patch instructions.

There are three more obscure methods used for debug and communication purposes.

Table 5. Communication Support Methods

OPTIONS	generates a list of methods supported for resource
TRACE	echo received request back to caller to check for changes
CONNECT	used to establish an HTTP tunnel — to proxy communications

Chapter 9. Method Safety

Proper execution of the internet protocols relies on proper outcomes for each method. With the potential of client-side proxies and server-side reverse proxies in the communications chain — one needs to pay attention to what can and should not change the state of a resource. "Method Safety" is a characteristic used to describe whether a method executed against a specific resource modifies that resource or has visible side effects.

9.1. Safe and Unsafe Methods

The following methods are considered "Safe" — thus calling them should not modify a resource and will not invalidate any intermediate cache.

- GET
- HEAD
- OPTIONS
- TRACE

The following methods are considered "Unsafe" — thus calling them is assumed to modify the resource and will invalidate any intermediate cache.

- POST
- PUT
- PATCH
- DELETE
- CONNECT

9.2. Violating Method Safety

Do not violate default method safety expectations

Internet communications is based upon assigned method safety expectations. However, these are just definitions. Your application code has the power to implement resource methods any way you wish and to knowingly or unknowingly violate these expectations. Learn the expected characteristics of each method and abide by them or risk having your API not immediately understood and render built-in Internet capabilities (e.g., caches) useless. The following are examples of what **not** to do:



Example Method Safety Violations

```
GET /jaxrsInventoryWAR/api/products/1?command=DELETE ①  
POST /jaxrsInventoryWAR/api/products/1 ②  
  content: {command:'getProduct'}
```

- ① method violating GET Safety rule
- ② unsafe POST method tunneling safe GET command

Chapter 10. Idempotent

Idempotence describes a characteristic where a repeated event produces the same outcome every time executed. This is a very important concept in distributed systems that commonly have to implement eventual consistency — where failure recovery can cause unacknowledged commands to be executed multiple times.

The idempotent characteristic is independent of method safety. Idempotence only requires that the same result state be achieved each time called.

10.1. Idempotent and non-Idempotent Methods

The application code implementing the following HTTP methods should strive to be idempotent.

- GET
- PUT
- DELETE
- HEAD
- OPTIONS

The following HTTP methods are defined to not be idempotent.

- POST
- PATCH
- CONNECT



Relationship between Idempotent and browser page refresh warnings?

The standard convention of Internet protocol is that most methods except for POST are assumed to be idempotent. That means a page refresh for a page obtained from a GET gets immediately refreshed and a warning dialogue is displayed if it was the result of a POST.

Chapter 11. Response Status Codes

Each HTTP response is accompanied by a standard [HTTP status code](#). This is a value that tells the caller whether the request succeeded or failed and a type of success or failure.

Status codes are separated into five (5) categories

- 1xx - informational responses
- 2xx - successful responses
- 3xx - redirected responses
- 4xx - client errors
- 5xx - server errors

11.1. Common Response Status Codes

The following are common response status codes

Table 6. Common HTTP Response Status Codes

Code	Name	Meaning
200	OK	"We achieved what you wanted - may have previously done this"
201	CREATED	"We did what you asked and a new resource was created"
202	ACCEPTED	"We received your request and will begin processing it later"
204	NO_CONTENT	"Just like a 200 with an empty payload, except the status makes this clear"
400	BAD_REQUEST	"I do not understand what you said and never will"
401	UNAUTHORIZED	"We need to know who you are before we do this"
403	FORBIDDEN	"We know who you are and you cannot say what you just said"
404	NOT_FOUND	"We could not locate the target resource of your request"
422	UNPROCESSABLE_ENTITY	"I understood what you said, but you said something wrong"
500	INTERNAL_ERROR	"Ouch! Nothing wrong with what you asked for or supplied, but we currently have issues completing. Try again later and we may have this fixed."

Chapter 12. Representations

Resources may have multiple independent representations. There is no direct tie between the data format received from clients, returned to clients, or managed internally. Representations are exchanged using standard [MIME or Media types](#). Common media types for information include

- application/json
- application/xml
- text/plain

Common data types for raw images include

- image/jpg
- image/png

12.1. Content Type Headers

Clients and servers specify the type of content requested or supplied in header fields.

Table 7. HTTP Content Negotiation Headers

Accept	defines a list of media types the client understands, in priority order
Content-Type	identifies the format for data supplied in the payload

In the following example, the client supplies a representation in `text/plain` and requests a response in XML or JSON — in that priority order. The client uses the `Accept` header to express which media types it can handle and both use the `Content-Type` to identify the media type of what was provided.

Example `Accept` and `Content-Type` Headers

```
> POST /greeting/hello
> Accept: application/xml,application/json
> Content-Type: text/plain
hi

< 200/OK
< Content-Type: application/xml
<greeting type="hello" value="hi"/>
```

The next exchange is similar to the previous example, with the exception that the client provides no payload and requests JSON or anything else (in that priority order) using the `Accept` header. The server returns a JSON response and identifies the media type using the `Content-Type` header.

Example `JSON Accept` and `Content-Type` Headers

```
> GET /greeting/hello?name=jim
```



```
> Accept: application/json,*/*
```

```
< 200/OK
```

```
< Content-Type: application/json
```

```
{ "msg" : "hi, jim" }
```

Chapter 13. Links

RESTful applications dynamically express their state through the use of hyperlinks. That is an RMM Level 3 characteristic use of links. As mentioned earlier, REST-like APIs do not include that level of complexity. If they do use links, these links will likely be constrained to standard response headers.

The following is an example partial POST response with links expressed in the header.

Example Response Headers with Links

```
POST http://localhost:8080/ejavaTodos/api/todo_lists
{"name":"My First List"}
=> Created/201
Location: http://localhost:8080/ejavaTodos/api/todo_lists/My%20First%20List ①
Content-Location: http://localhost:8080/ejavaTodos/api/todo_lists/My%20First%20List ②
```

- ① Location expresses the URI to the resource just acted upon
- ② Content-Location expresses the URI of the resource represented in the payload

Chapter 14. Summary

In this module, we learned that:

- technically — terms "REST" and "RESTful" have a specific meaning defined by Roy Fielding
- the Richardson Maturity Model (RMM) defines several levels of compliance to RESTful concepts, with level 3 being RESTful
- very few APIs achieve full RMM level 3 RESTful adoption
 - but that is OK!!! — there are many useful and powerful WWW constructs easily made available before reaching the RMM level 3
 - can be referred to as "REST-like", "HTTP-based", or "RMM level 2"
 - marketers of the world attempting to leverage a buzzword, will still call them REST APIs
- most serious REST-like APIs adopt
 - HTTP
 - multiple resources identified through URIs
 - HTTP-compliant use of methods and status codes
 - method implementations that abide by defined safety and idempotent characteristics
 - standard resource representation formats like JSON, XML, etc.