# Building an application

Table of Contents

[Building an application 1](#_Toc455924461)

[Understanding the web 2](#_Toc455924462)

[Patterns of things that need to be covered in the Web 3](#_Toc455924463)

[Models, Views, Controllers 3](#_Toc455924464)

[VIews - Routing – getting to the page 5](#_Toc455924465)

[What is routing 5](#_Toc455924466)

[Types of routing 6](#_Toc455924467)

[Examples of literal routing 7](#_Toc455924468)

[Dynamic routes rely on parameters ie params 8](#_Toc455924469)

[Views - Templates – getting content onto the page when the router has sent the client there 8](#_Toc455924470)

[Models- Dealing with data 10](#_Toc455924471)

[Parts of a DB 10](#_Toc455924472)

[CRUD – create, read, update, delete 11](#_Toc455924473)

[Controlling everything 12](#_Toc455924474)

[HTTP – what is it? 12](#_Toc455924475)

[Technical overview[edit] 12](#_Toc455924476)

[ReSTFUL architecture – What is it? 14](#_Toc455924477)

[why is rest good? 15](#_Toc455924478)

[Implementing restful architecture - Wikipediea [edit] 16](#_Toc455924479)

[Idempotence – see 19](#_Toc455924480)

[Connecting HTTP methods to urls [edit] 20](#_Toc455924481)

[PUtting it all together Rest and working with DB – connecting URLs, crud and rest 21](#_Toc455924482)

[Other patterns - SOAP 23](#_Toc455924483)

[Files 23](#_Toc455924484)

[COMMANDS 23](#_Toc455924485)

[Object Oriented Programming (OOP) 24](#_Toc455924486)

[Active Record 24](#_Toc455924487)

[Preface - Object Relational Mapping 24](#_Toc455924488)

[What does Active Record do? 24](#_Toc455924489)

[Convention over configuration – names to sort out DB associations 25](#_Toc455924490)

[How do you work with Active Record 25](#_Toc455924491)

[CRUD with Active Record 25](#_Toc455924492)

[Create 25](#_Toc455924493)

[Read 26](#_Toc455924494)

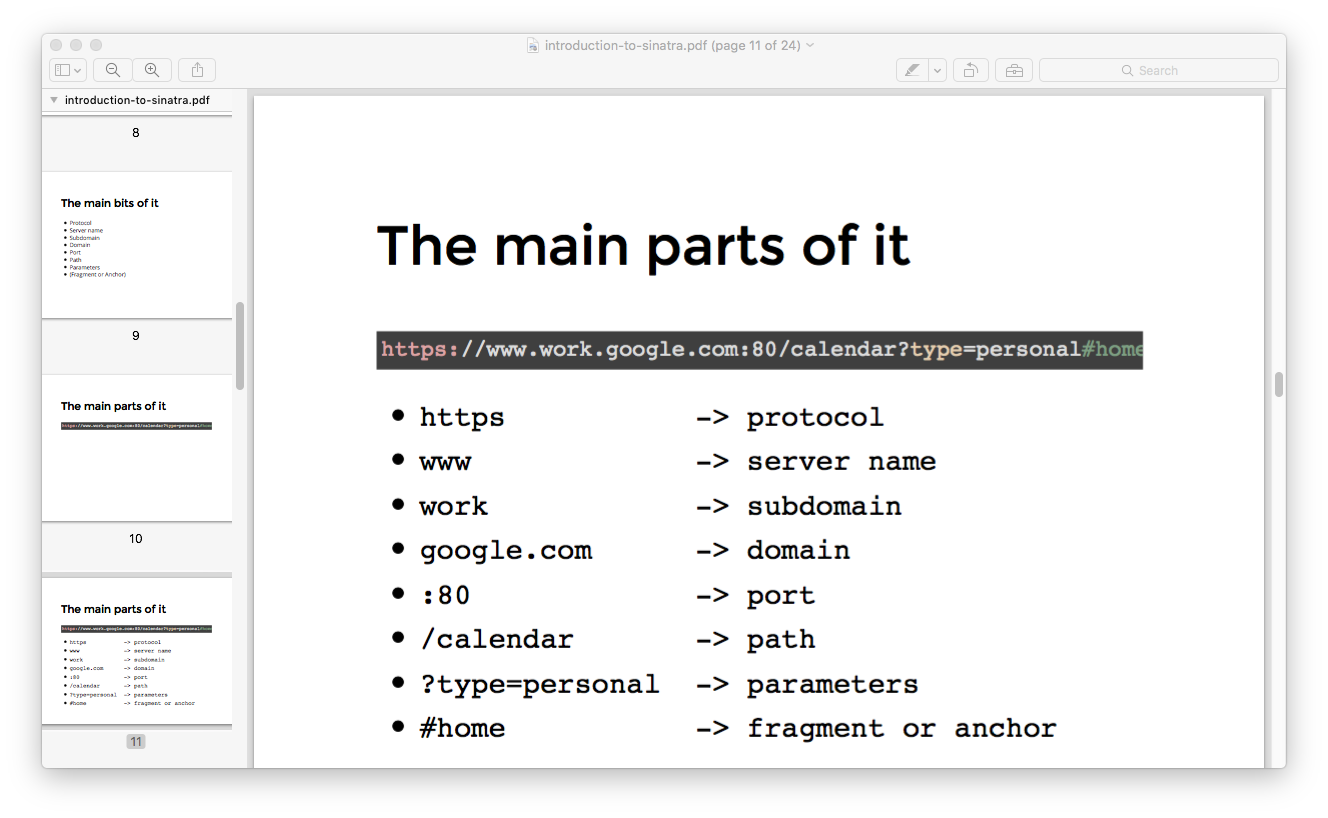
[Update 27](#_Toc455924495)

[Delete 27](#_Toc455924496)

[What are APIs 28](#_Toc455924497)

[APIs: Windows To The Code 28](#_Toc455924498)

# Understanding the web

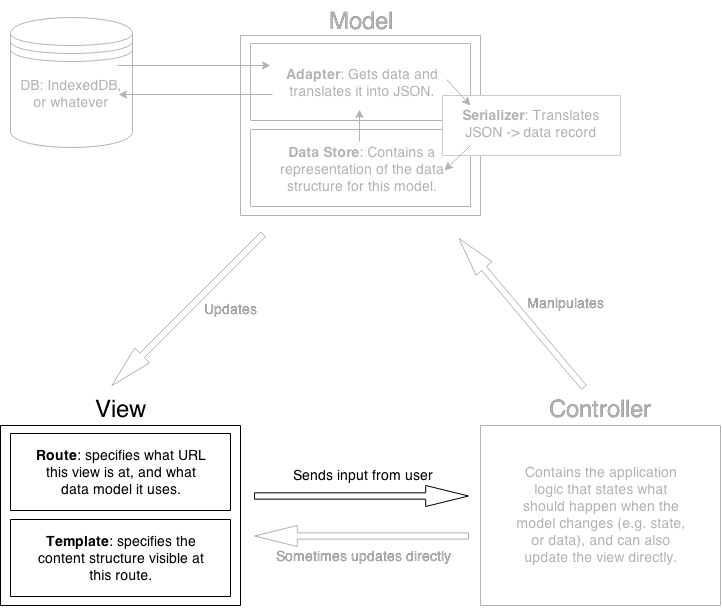


# Patterns of things that need to be covered in the Web

## Models, Views, Controllers

A way of conceptualizing the different parts of a web application.

From MDN



# VIews - Routing – getting to the page

## What is routing

We respond to requests (paths), that's how all web servers work

This is called routing. Routes are always matched in the order they are defined - first route it matches will be used

**https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern\_web\_app\_architecture/Views\_and\_templates**

**Routes** tell an app that a certain part of the content/functionality *will exist at a certain URL*. Along with [Templates](https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/Views_and_templates#Templates) ie VIEWS, which specify the content structure that will be visible at each route, they form the basis of the "V" (View) in MVC in Ember applications. Think of a route as a 'page' of your application.

## Types of routing

2 types of routing in code

## Examples of literal routing

get '/' do

"This is a get request to the root path - visited at localhost:4567/"

end

get '/anything' do

"This is a get request - visited at localhost:4567/anything" # This is returned as a response

end

## Dynamic routes rely on parameters ie params

Don’t always want to ‘hard code’ the routes. The way we solve this is by using *named parameters*. Whatever is matched by the thing prefixed with the colon is stored in the params hash (which is automatically generated for us) and stored

get '/hello/:name' do

# matches "/hello/foo" or "/hello/bar" or anything else that starts with "/hello/""

# params['name'] might be 'foo' or 'bar'

"Hello #{ params['name'] }!"

end

Params refers to the *parameters* being passed to the controller via a GET or POST request ie the data that gets sent from the browser to the application. The application looks at it and the controller goes “I know where you want to go/what information should be delivered to this address”. I’ll go here to do it.

params is an alias for the paramaters method.

Params comes from ActionController::Base, which is accessed by your application via ApplicationController. Specifically, params refers to the parameters being passed to the controller via a GET or POST request.

In a GET request, params get passed to the controller from the URL in the user’s browser.

# Views - Templates – getting content onto the page when the router has sent the client there

1. Views folder
   1. layout.erb file. This will be used by default everytime Sinatra sees the erb command. This is where all of the stuff that you want to be on every page should go.

Need to include the following on the page at the end before the body

<%= yield %> this pulls in the content as routed (directed by the main.erb folder This grabs the contents of the file requested and puts it straight over itself.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<%= yield %>

</body>

1. Main.erb page

get '/post' do

erb :post

end

</html>

For more information about Sinatra, see [here.](http://www.sinatrarb.com/intro.html)

1. Public folder
   1. style.css

# Models- Dealing with data

## Parts of a DB

Involved in every database, there are a couple of things. We have:

## CRUD – **c**reate, **r**ead, **u**pdate, **d**elete

When we talk about tables and databases, there is really only 4 tasks that we need to do.

* **C**reate
* **R**ead
* **U**pdate
* **D**elete

This is called CRUD. Can use this model in relation to:

* The Database itself
* Individual tables
* Individual records on tables.

# Controlling everything



ie making sure we link the routes get to either the DB or the views that is wanted

## HTTP – what is it?

<https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>

The **Hypertext Transfer Protocol** (**HTTP**) is an [application protocol](https://en.wikipedia.org/wiki/Application_protocol) for *distributed, collaborative,*[*hypermedia*](https://en.wikipedia.org/wiki/Hypermedia)*information systems*.[[1]](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#cite_note-ietf2616-1) HTTP is the foundation of data communication for the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web). WOW PUT THIS IN THE POST!!!

[Hypertext](https://en.wikipedia.org/wiki/Hypertext) is structured text that uses logical links ([hyperlinks](https://en.wikipedia.org/wiki/Hyperlinks)) between [nodes](https://en.wikipedia.org/wiki/Node_(computer_science)) containing text. HTTP is the protocol to exchange or transfer hypertext.

## **Technical overview[**[**edit**](https://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&action=edit&section=1)**]**

[](https://en.wikipedia.org/wiki/File:Internet1.jpg)

[URL](https://en.wikipedia.org/wiki/URL) beginning with the HTTP scheme and the [WWW](https://en.wikipedia.org/wiki/World_Wide_Web) domain name label.

HTTP functions as a [request–response](https://en.wikipedia.org/wiki/Request%E2%80%93response) protocol in the [client–server](https://en.wikipedia.org/wiki/Client%E2%80%93server) computing model.

It manages this relationship:

A [web browser](https://en.wikipedia.org/wiki/Web_browser), for example, may be the *client* and an application running on a computer [hosting](https://en.wikipedia.org/wiki/Host_(network)) a [web site](https://en.wikipedia.org/wiki/Web_site) may be the *server*. The client submits an HTTP *request* message to the server. The server, which provides *resources* such as [HTML](https://en.wikipedia.org/wiki/HTML) files and other content, or performs other functions on behalf of the client, returns a *response* message to the client. The response contains completion status information about the request and may also contain requested content in its message body.

[HTTP resources](https://en.wikipedia.org/wiki/Web_resource) are identified and located on the network by [uniform resource locators (URLs)](https://en.wikipedia.org/wiki/Uniform_resource_locator), using the [uniform resource identifier (URI)](https://en.wikipedia.org/wiki/Uniform_resource_identifier) schemes *http* and *https*. URIs and[hyperlinks](https://en.wikipedia.org/wiki/Hyperlink) in [Hypertext Markup Language](https://en.wikipedia.org/wiki/Hypertext_Markup_Language) (HTML) documents form inter-linked [hypertext](https://en.wikipedia.org/wiki/Hypertext) documents.

HTTP/1.1 is a revision of the original HTTP (HTTP/1.0). In HTTP/1.0 a separate [connection](https://en.wikipedia.org/wiki/Connection-oriented_communication) to the same server is made for every resource request. HTTP/1.1 can reuse a connection multiple times to download images, [scripts](https://en.wikipedia.org/wiki/Client-side_scripting), [stylesheets](https://en.wikipedia.org/wiki/Cascading_Style_Sheets), *etc* after the page has been delivered. HTTP/1.1 communications therefore experience less [latency](https://en.wikipedia.org/wiki/Latency_(engineering)) as the establishment of TCP connections presents considerable overhead.

## ReSTFUL architecture – What is it?

<http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming>

<http://web.archive.org/web/20130116005443/http://tomayko.com/writings/rest-to-my-wife>

<https://en.wikipedia.org/wiki/Representational_state_transfer>

REST ReSTis the underlying architectural principle of the web.

Stands for “representational state transfer”

Is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use

STATE refers to the ‘state’ of the communication between the client and the server. Something is “stateless” in the following circumstances:

“no client context being stored on the server between requests. Each request from any client contains *all the information* necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition. “[[12]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-12)

It is a way of managing – in a consistent way – the requests and responses between the client (browser) and the server. The REQUESTs are like REPRESENTATIONS.

An API that adheres to the principles of REST does not require the client to know anything about the structure of the API. Rather, the *server* needs to provide whatever information the client needs to interact with the service. An HTML form is an example of this: The server specifies the *location of the resource*, and the required fields. **The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.** (This principle is called HATEOAS.)

**So, how does this apply to HTTP, and how can it be implemented in practice?** *HTTP is oriented around verbs and resources.* The two verbs in mainstream usage are GET and POST, which I think everyone will recognize. However, the HTTP standard defines several others such as PUT and DELETE. These verbs are then applied to resources, according to the instructions provided by the server.

A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.

The client (browser) can make different

## **why is rest good?**

**Architectural properties[**[**edit**](https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&action=edit&section=2)**] – from wikipedia**

The architectural properties affected by the constraints of the REST architectural style are[[1]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-1)[[10]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-SOA_with_REST-10)

* **Performance** - component interactions can be the dominant factor in user-perceived performance and network efficiency[[11]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch2-11)
* [Scalability](https://en.wikipedia.org/wiki/Scalability) to support large numbers of components and interactions among components. [Roy Fielding](https://en.wikipedia.org/wiki/Roy_Fielding), one of the principal authors of the HTTP specification, describes REST's effect on scalability as follows:

REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—[proxies](https://en.wikipedia.org/wiki/Proxy_server),[gateways](https://en.wikipedia.org/wiki/Gateway_(telecommunications)), and [firewalls](https://en.wikipedia.org/wiki/Firewall_(computing))—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate [cacheability](https://en.wikipedia.org/wiki/Web_cache).[[1]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-1)

* Simplicity of [interfaces](https://en.wikipedia.org/wiki/Interface_(computer_science))
* Modifiability of components to meet changing needs (even while the application is running)
* Visibility of communication between components by service agents
* Portability of components by moving program code with the data
* [Reliability](https://en.wiktionary.org/wiki/reliability) is the resistance to failure at the system level in the presence of failures within components, connectors, or data[[11]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch2-11)

## Implementing restful architecture - Wikipediea **[**[**edit**](https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&action=edit&section=10)**]**

Web service APIs that adhere to the [REST architectural constraints](https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints) are called [RESTful APIs](http://restfulapi.net/). HTTP-based RESTful APIs are defined with the following aspects:[[4]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Richardson_2013-4)

* base [URL](https://en.wikipedia.org/wiki/URL), such as http://example.com/resources/
* an [Internet media type](https://en.wikipedia.org/wiki/Internet_media_type) for the data. This is often [JSON](https://en.wikipedia.org/wiki/JSON) but can be any other valid Internet media type (e.g., XML, Atom, microformats, application/vnd.collection+json,[[4]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Richardson_2013-4):91–99 etc.)
* standard [HTTP methods](https://en.wikipedia.org/wiki/HTTP_method) (e.g., OPTIONS, GET, PUT, POST, and DELETE)[[14]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-14)
* state transition resources. The current representation tells the client how to compose all transitions to the next application state. This could be as simple as a URL or as complex as a java applet.[[15]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-15)

ReSTFUL architecture is architected around a series of *verbs*. The key verbs are

See http://www.restapitutorial.com/lessons/httpmethods.html

|  |  |
| --- | --- |
| Browser request | Use it to do the following |
| Post | to \*\*create\*\* new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.  On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.  POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information. |
| Get | The HTTP GET method is used to \*\*read\*\* (or retrieve) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).  According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.  Do not expose unsafe operations via GET—it should never modify any resources on the server. |
| Put | PUT is most-often utilized for \*\*update\*\* capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.  However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.  Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).  On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.  PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.  If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests |
| Patch | PATCH is used for \*\*modify\*\* capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.  This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.  PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request. |
| Delete | DELETE is pretty easy to understand. It is used to \*\*delete\*\* a resource identified by a URI.  On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.  HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.  There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call. |

## Idempotence – see

From http://www.restapitutorial.com/lessons/idempotency.html

Idempotence is a funky word that often hooks people. Idempotence is sometimes a confusing concept, at least from the academic definition.

From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

## Connecting HTTP methods to urls **[**[**edit**](https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&action=edit&section=11)**]**

In order to enable through a user to move through different *states/parts of the application,*  we need to connect the HTTP REQUEST METHODS with the URLS ie our application needs to be able to say *what to do when* users go to a particular URL.

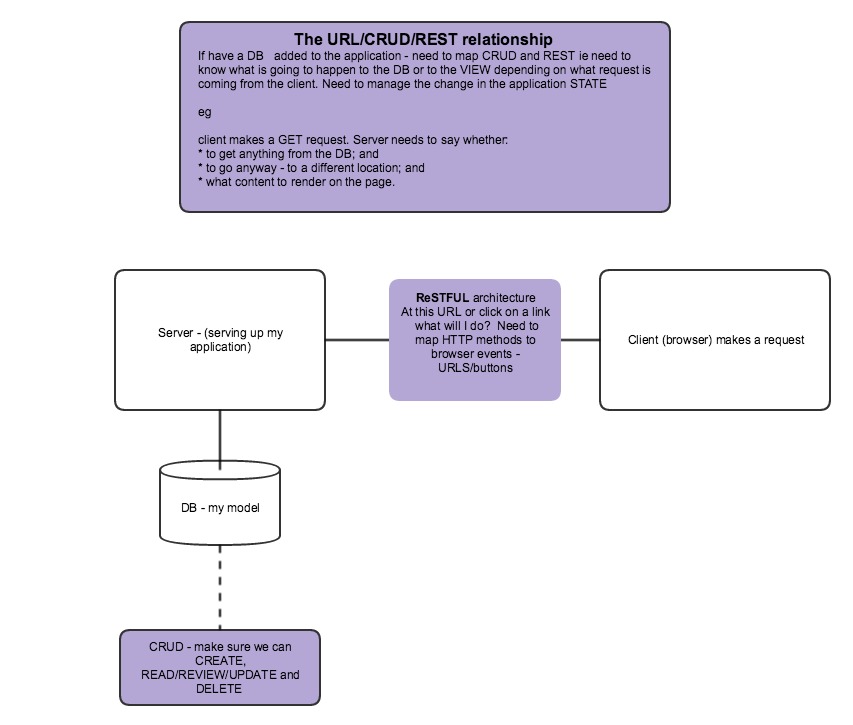
The following table shows how HTTP methods are typically used in a RESTful API:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **HTTP methods** | | | | |
| **Uniform Resource Locator (URL)** | **GET** | **PUT** | **POST** | **DELETE** |
| **Collection, such ashttp://api.example.com/resources/** | **List** the URIs and perhaps other details of the collection's members. | **Replace** the entire collection with another collection. | **Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[[16]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-thereisnorightway-16) | **Delete** the entire collection. |
| **Element, such ashttp://api.example.com/resources/item17** | **Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type. | **Replace** the addressed member of the collection, or if it does not exist, **create**it. | Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.[[16]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-thereisnorightway-16) | **Delete** the addressed member of the collection. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL** | **HTTP** | **DDS** |
| Create | [INSERT](https://en.wikipedia.org/wiki/Insert_(SQL)) | [PUT](https://en.wikipedia.org/wiki/PUT_(HTTP)) / [POST](https://en.wikipedia.org/wiki/POST_(HTTP)) | write |
| Read (Retrieve) | [SELECT](https://en.wikipedia.org/wiki/Select_(SQL)) | [GET](https://en.wikipedia.org/wiki/GET_(HTTP)) | read / take |
| Update (Modify) | [UPDATE](https://en.wikipedia.org/wiki/Update_(SQL)) | [POST](https://en.wikipedia.org/wiki/POST_(HTTP)) / [PUT](https://en.wikipedia.org/wiki/PUT_(HTTP)) / [PATCH](https://en.wikipedia.org/wiki/PATCH_(HTTP)) | write |
| Delete (Destroy) | [DELETE](https://en.wikipedia.org/wiki/Delete_(SQL)) | [DELETE](https://en.wikipedia.org/wiki/DELETE_(HTTP)) | dispose |

## PUtting it all together Rest and working with DB – connecting URLs, crud and rest

If working with a DB, need to also ensure that the ReSTful architecture which controls what is happening between the client and the server, effectively also manages what is happening between the client and the database.



## Other patterns - SOAP

Other patterns - [SOAP](https://en.wikipedia.org/wiki/SOAP)-based web services, there is no "official" standard for RESTful web APIs.[[17]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Elkstein-17) This is because REST is an architectural style, while SOAP is a protocol. Even though REST is not a standard *per se*, most RESTful implementations make use of standards such as [HTTP](https://en.wikipedia.org/wiki/HTTP), [URL](https://en.wikipedia.org/wiki/URL), [JSON](https://en.wikipedia.org/wiki/JSON), and [XML](https://en.wikipedia.org/wiki/XML).[[17]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Elkstein-17)

### Files

## COMMANDS

# Object Oriented Programming (OOP)

Basically, OOP is an approach to development that tries to replicate real life. It is pretty much always done using objects or classes as namespaces and treats them as a way to make your code "modular".

# What are APIs

## APIs: Windows To The Code

<http://readwrite.com/2013/09/19/api-defined/>

In the simplest terms, APIs are sets of requirements that govern how one application can talk to another. APIs aren’t at all new; whenever you use a desktop or laptop, APIs are what make it possible to move information between programs—for instance, by cutting and pasting a snippet of a LibreOffice document into an Excel spreadsheet. System-level APIs makes it possible for applications like LibreOffice to run on top of an OS like Windows in the first place.

On the Web, APIs make it possible for big services like Google Maps or Facebook to let other apps “piggyback” on their offerings. Think about the way Yelp, for instance, displays nearby restaurants on a Google Map in its app, or the way some video games n2ow let players [chat, post high scores and invite friends to play via Facebook](http://readwrite.com/2013/08/28/facebook-unity-sdk-3d-gaming), right there in the middle of a game.

***See also:***[***The New API Gold Rush***](http://readwrite.com/2013/04/24/api-gold-rush)

APIs do all this by “exposing” some of a program’s internal functions to the outside world in a limited fashion. That makes it possible for applications to share data and take actions on one another’s behalf without requiring developers to share all of their software’s code. Code-sharing on that scale wouldn’t just ruffle the feathers of programmers who’d rather keep it secret; it would also be grossly inefficient.

That’s true even for open-source programs. Who has the time to comb through all the code for somebody else’s application—which, trust me, can be awfully messy—just to use one function? (It’s also possible to run into tricky licensing issues if you’re not careful.)

APIs simplify all that by limiting outside program access to a specific set of features—often enough, requests for data of one sort or another. Feel free to think of them as doors, windows or levers if you like. Whatever the metaphor, APIs clearly define exactly how a program will interact with the rest of the software world—saving time, resources and potentially nasty legal entanglements along the way.