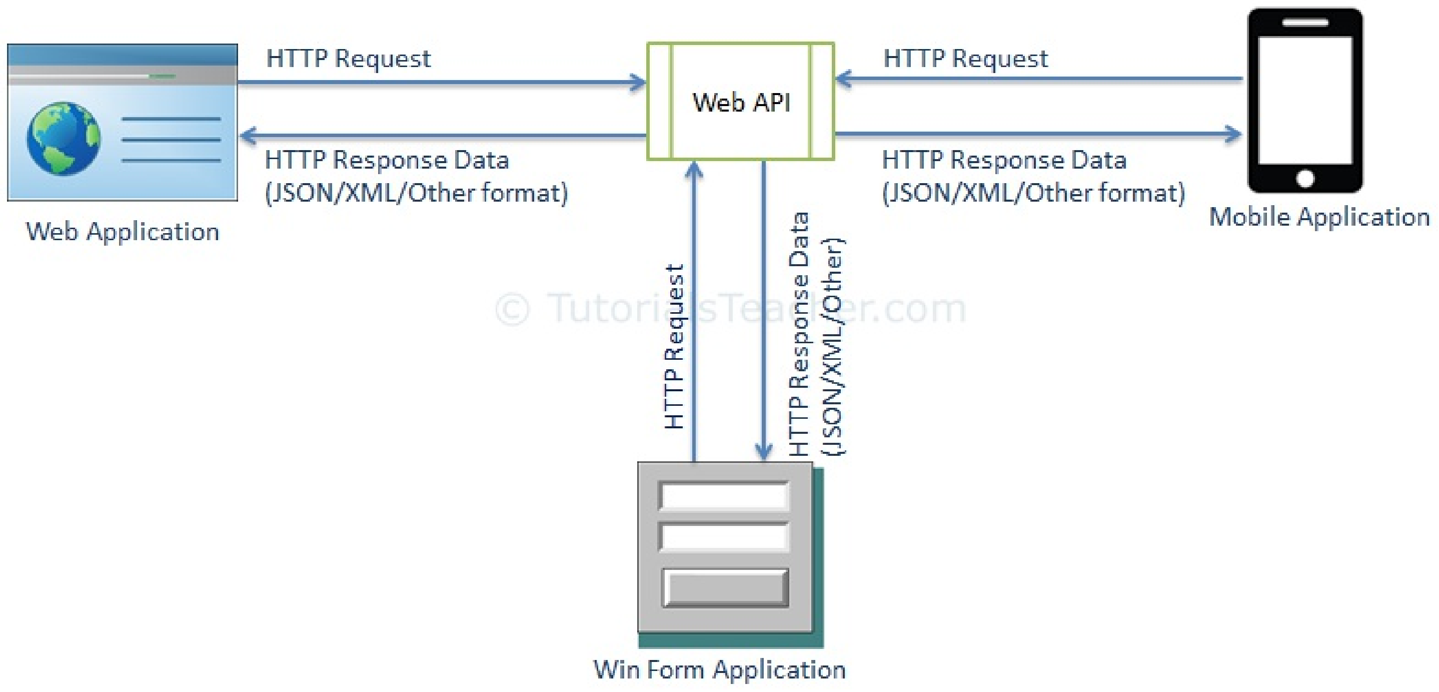
Web API

What is an API?

API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other. Each time you use an app like Facebook, send an instant message, or check the weather on your phone, you're using an API.

So what is a Web API

Web API as the name suggests, is an API over the web which can be accessed using HTTP protocol. It is a concept and not a technology. We can build Web API using different technologies such as Java, .NET etc. For example, Twitter's REST APIs provide programmatic access to read and write data using which we can integrate twitter's capabilities into our own application.



So What is REST?

REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation.

Like any other architectural style, REST also does have it's **own 6 guiding constraints** which must be satisfied if an interface needs to be referred as RESTful.
These principles are listed below

Guiding Principles of REST

- ### Client–server:

  By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

- ### Stateless:

  Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

- ### Cacheable:

  Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

- ### Uniform interface:

  By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by **four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia** as the engine of application state.

- ### Layered system:

  The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.

- ### Code on demand (optional):

  REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resource

The key abstraction of information in REST is a **resource**. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on. REST uses a **resource identifier** to identify the particular resource involved in an interaction between components.

The state of resource at any particular timestamp is known as **resource representation**. A representation consists of data, metadata describing the data and **hypermedia** links which can help the clients in transition to next desired state.
The data format of a representation is known as a **media type.**

Http Verbs (Standard)

The Http verbs are a standart definition from HTTP, basically how the web is builded, normally application used this verbs for specific operation (based on the guidelines of REST, however you can violated the rules, even is highly discouraged)

HTTP GET

Use GET requests to **retrieve resource representation/information only – and not to modify it in any way**. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another methods (POST or PUT) has changed the state of the resource on the server.

For any given HTTP GET, if the resource is found on the server then it must return HTTP response `code 200 (OK)` – along with response body which is usually either XML or JSON content (due to their platform independent nature).

In case resource is NOT found on server then it must return HTTP response `code 404 (NOT FOUND)`. Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code `400 (BAD REQUEST)`.

Example request URIs
HTTP GET http://www.appdomain.com/users

HTTP POST

Use POST to **create new subordinate resources**, e.g. a file is subordinate to a directory containing it or a row is subordinate to a database table. Talking strictly in terms of REST, POST methods are used to **create a new resource into the collection of resources.**

Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response `code 201 (Created)` and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response `code 200 (OK)` or `204 (No Content)` is the appropriate response status.

Example request URIs

HTTP POST http://www.appdomain.com/users

HTTP PUT

Use PUT APIs primarily to **update existing resource** (if the resource does not exist then API may decide to create a new resource or not). If a new resource has been created by the PUT, the origin server MUST inform the user agent via the HTTP response `code 201 (Created)` response and if an existing resource is modified, either the `200 (OK)` or `204 (No Content)` *response codes SHOULD be sent to indicate successful completion of the request*.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. *Responses to this method are not cacheable*.

> *The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections whereas PUT requests are made on an individual resource.*

Example request URIs

HTTP PUT http://www.appdomain.com/users/123

HTTP DELETE

As the name applies, DELETE are **used to delete resources** (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response `code 200 (OK)` if the response includes an entity describing the status, `202 (Accepted)` if the action has been queued, or `204 (No Content)` if the action has been performed but the response does not include an entity.

DELETE operations are *idempotent*. If you DELETE a resource, it’s removed from the collection of resource. Repeatedly calling DELETE on that resource will not change the outcome – however calling DELETE on a resource a second time will return a `404 (NOT FOUND)` since it was already removed.

Example request URIs
HTTP DELETE http://www.appdomain.com/users/123

HTTP PATCH

HTTP PATCH requests are to *make partial update on a resource*. If you see PUT requests also modify a resource entity so to make more clear – PATCH method is the correct choice for partially updating an existing resource and PUT should only be used if you’re replacing a resource in its entirety.

Please note that there are some challenges if you decide to use PATCH in *your application:*

> *Support for PATCH in browsers, servers, and web application frameworks is not universal. IE8, PHP, Tomcat, Django, and lots of other software has missing or broken support for it.*

Request payload of PATCH request is not straightforward as it is for PUT request. Example:
HTTP GET /users/1
produces below response:

```
{id: 1, username: 'admin', email: 'email@example.org'}
```

A sample patch request to update the email will be like this:
HTTP PATCH /users/1

```
[{ “op”: “replace”, “path”: “/email”, “value”: “new.email@example.org” }]
```

There may be following possible operations are per HTTP specification.

```
[
{ "op": "test", "path": "/a/b/c", "value": "foo" },
{ "op": "remove", "path": "/a/b/c" },
{ "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
{ "op": "replace", "path": "/a/b/c", "value": 42 },
{ "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
{ "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

PATCH method is not a replacement for the POST or PUT methods. It applies a delta (diff) rather than replacing the entire resource.

ASP Web API

The ASP .NET Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms such as web, windows, mobile etc. It works more or less the same way as ASP .NET MVC web application except that it sends data as a response instead of html view. It is like a webservice or WCF service but the exception is that it only supports HTTP protocol

ASP .Net Web API Characteristics

- ASP .NET Web API is an ideal platform for building RESTful services.
- ASP .NET Web API is built on top of ASP .NET and supports ASP .NET request/response pipeline
- ASP .NET Web API maps HTTP verbs to method names.
- ASP .NET Web API supports different formats of response data. Built-in support for JSON, XML, BSON format.
- ASP .NET Web API can be hosted in IIS, Self-hosted or other web server that supports .NET 4.0+.
- ASP .NET Web API framework includes new HttpClient to

communicate with Web API server. HttpClient can be used in ASP.MVC server side, Windows Form application, Console application or other apps.

Differences between WCF and Web API

ASP .NET Web API vs WCF

Web API	**WCF**
Syntax	Description
Open source and ships with .NET framework.	Ships with .NET framework
Supports only HTTP protocol.	Supports HTTP, TCP, UDP and custom transport protocol.
Maps http verbs to methods	Uses attributes based programming model.
Uses routing and controller concept similar to ASP .NET MVC.	Uses Service, Operation and Data contracts.
Does not support Reliable Messaging and transaction.	Supports Reliable Messaging and Transactions.
Web API can be configured using HttpConfiguration class but not in web.config.	Uses web.config and attributes to configure a service.
Ideal for building RESTful services.	Supports RESTful services but with limitations.