**Data Access Layer (REST API)**

A **data access layer** (**DAL**) in computer software, is a [layer](https://en.wikipedia.org/wiki/Layer_(object-oriented_design)) of a [computer program](https://en.wikipedia.org/wiki/Computer_program) which provides simplified access to [data](https://en.wikipedia.org/wiki/Data) stored in [persistent storage](https://en.wikipedia.org/wiki/Persistent_storage) of some kind, such as an [entity-relational](https://en.wikipedia.org/wiki/Entity_relationship) [database](https://en.wikipedia.org/wiki/Database). This [acronym](https://en.wikipedia.org/wiki/Acronym) is prevalently used in [Microsoft](https://en.wikipedia.org/wiki/Microsoft) environments.

For example, the DAL might return a reference to an [object](https://en.wikipedia.org/wiki/Object_(computer_science)) (in terms of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming)) complete with its attributes instead of a [row](https://en.wikipedia.org/wiki/Row_(database)) of [fields](https://en.wikipedia.org/wiki/Field_(computer_science)) from a database [table](https://en.wikipedia.org/wiki/Table_(database)). This allows the [client](https://en.wikipedia.org/wiki/Client_(computing)) (or user) modules to be created with a higher level of [abstraction](https://en.wikipedia.org/wiki/Abstraction). This kind of model could be implemented by creating a class of data access methods that directly reference a corresponding set of database stored procedures. Another implementation could potentially retrieve or write records to or from a file system. The DAL hides this complexity of the underlying data store from the external world.

For example, instead of using commands such as insert, delete, and update to access a specific table in a database, a class and a few stored procedures could be created in the database. The procedures would be called from a method inside the class, which would return an object containing the requested values. Or, the insert, delete and update commands could be executed within simple functions like registeruser or loginuser stored within the data access layer.

Also, business logic methods from an application can be mapped to the Data Access Layer. So, for example, instead of making a query into a database to fetch all users from several tables the application can call a single method from a DAL which abstracts those database calls.

Applications using a data access layer can be either database server dependent or independent. If the data access layer supports multiple database types, the application becomes able to use whatever databases the DAL can talk to. In either circumstance, having a data access layer provides a centralized location for all calls into the database, and thus makes it easier to port the application to other database systems (assuming that 100% of the database interaction is done in the DAL for a given application).

When working with data one option is to embed the data-specific logic directly into the presentation layer (in a web application, the ASP.NET pages make up the presentation layer). This may take the form of writing ADO.NET code in the ASP.NET page's code portion or using the SqlDataSource control from the markup portion. In either case, this approach tightly couples the data access logic with the presentation layer. The recommended approach, however, is to separate the data access logic from the presentation layer. This separate layer is referred to as the Data Access Layer, DAL for short, and is typically implemented as a separate Class Library project. The benefits of this layered architecture are well documented (see the "Further Readings" section at the end of this tutorial for information on these advantages) and is the approach we will take in this series.

All code that is specific to the underlying data source – such as creating a connection to the database, issuing SELECT, INSERT, UPDATE, and DELETE commands, and so on – should be located in the DAL. The presentation layer should not contain any references to such data access code, but should instead make calls into the DAL for any and all data requests. Data Access Layers typically contain methods for accessing the underlying database data.

REST stands for ‘Representational State Transfer’ and it is an architectural pattern for creating an API that uses HTTP as its underlying communication method. The term API stands for ‘Application Programming Interface’. In the world of web development the term ‘API’ is synonymous with online web services which client apps can use to retrieve and update data. These online services have had several names/formats over the years such as SOAP, however the current popular choice is to create a REST (or RESTful) API.



.Net’s Web API is an easy way to implement a RESTful web service using all of the goodness that the .net framework provides. Once you understand the basic principles of REST, then a .net Web API will be very easy to implement.

Web API is built on .net’s modular, pluggable pipeline model. This means that when a server hosting a web API receives a request, it passes through .nets request pipeline first. This enables you to easily add your own modules if you find that the default capabilities are not enough for your needs. With the recent announcements on ASP.net vNext this also means you can potentially host your Web API outside of Windows Server which opens up a whole range of usage cases. See <http://www.asp.net/vnext> for detail.

Web API uses the Controller and Action concepts from MVC so if you already understand .net MVC you are in a good place. If you don’t, then Web API is a great way to learn MVC.

Resources are mapped directly to controllers; you would typically have a different controller for each of your main data entities (Product, Person, Order etc). Web API uses the .net routing engine to map URLs to controllers. Typically, APIs are held within a ‘/api/’ route which helps to distinguish API controllers from other non-API in the same website.

Actions are used to map to specific HTTP verbs, for example you would typically have a GET action which returns all of the entities. This action would respond to /api/Products (where ‘products’ is your controller) and would look something like this:

public IEnumerable<string> Get()  
 {  
 return new string[] { "value1", "value2" };  
 }

You may also have a GET action which accepts a specific ID and returns a specific entity. It would respond to /api/Products/81 and would look something like this:

public string Get(int id)  
 {  
 return "value";  
 }