Original Coder – Layers

Architectural Overview

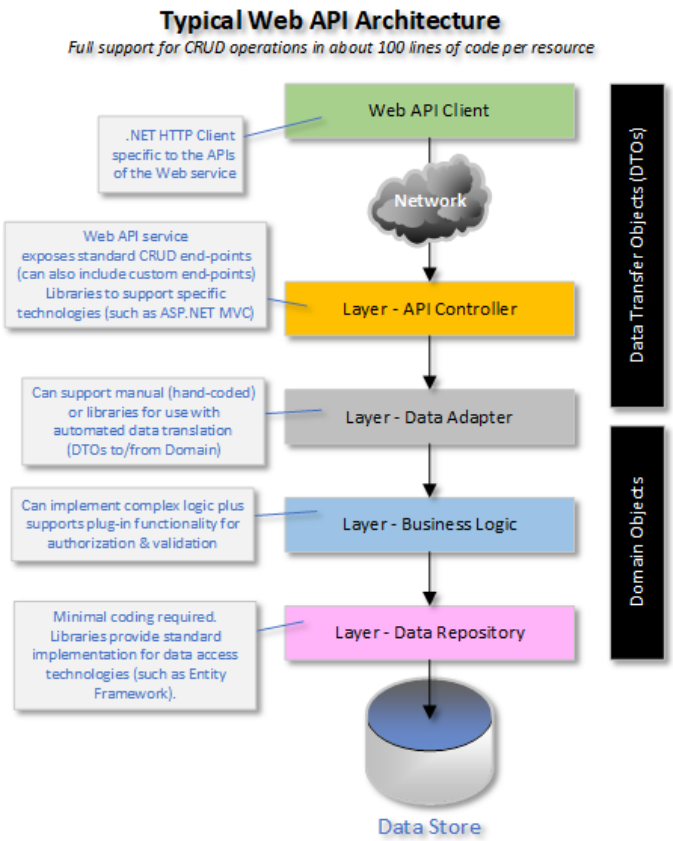
August 5th, 2019 (Alpha)

Visit the Original Coder blog at
http://OriginalCoder.dev

The Original Coder libraries are hosted at
https://github.com/TheOriginalCoder/Original-Coder-Libraries

The Original Coder Layers library is designed to provide a highly flexible and capable architecture for implementing layers of functionality in a system. It also defines a set of standard CRUD (Create, Read, Update & Delete) operations which it provides default implementation for to speed development.

A primary goal of the architecture is to be efficient for application developers to use. Using the architecture, a typical resource (for example a table in a database) can be fully implemented to provide a complete set of CRUD operations in about 100 lines of code. This is because all of the code typically written has been abstracted, modularized and implemented in standard base classes included in the library. The only code that must be written to implement is a resource is that which is required to define the resource and a handful of methods which provide required functionality that is specific to the resource (such as getting the key from an entity or filtering a database fetch by primary key).



Standard CRUD operations that require minimal code:

Create

- TKey Create(TEntity entity);
- TEntity CreateAndReturn(TEntity entity);

Update

- bool Update(TEntity entity);
- TEntity UpdateAndReturn(TEntity entity);

Delete

- bool Delete(TKey key);
- int Delete(IEnumerable<TKey> keys);

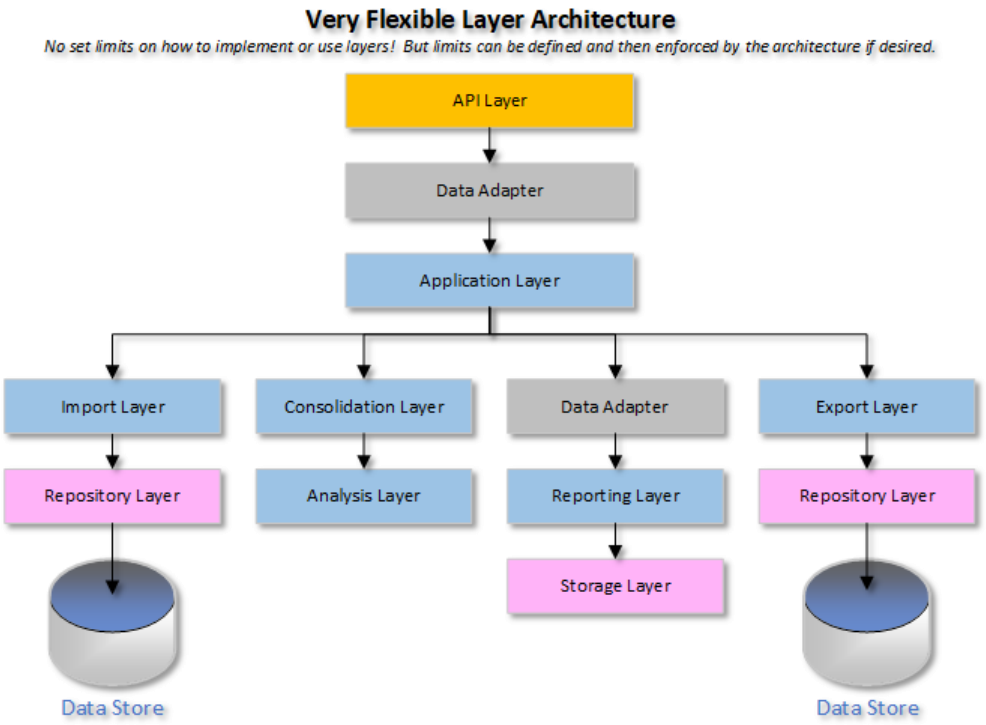
Read

- TEntity Get(TKey key);
- IReadOnlyList<TEntity> Get(IEnumerable<TKey> keys);
- IReadOnlyList<TEntity> Get();
- IReadOnlyList<TEntity> Get(int qtySkip, int qtyReturn, IReadOnlyList<(string column, bool ascending)> sortOrder);

Standardized search functionality can also be added with a bit more code:

- IReadOnlyList<TEntity> Get(TSearch searchParams);
- IReadOnlyList<TEntity> Get(TSearch searchParams, int qtySkip, int qtyReturn,
  IReadOnlyList<(string column, bool ascend)> sortOrder);

In addition to being efficient to implement, the architecture is designed to be highly configurable and flexible. It does not dictate nor limit the types, quantities or connection between layers that are possible.



However, if desired, the architecture does have support for defining limits regarding the types of layers supported and which layer types can interact with each other. If defined the architecture will implement and enforce the rules.

Example configuration:

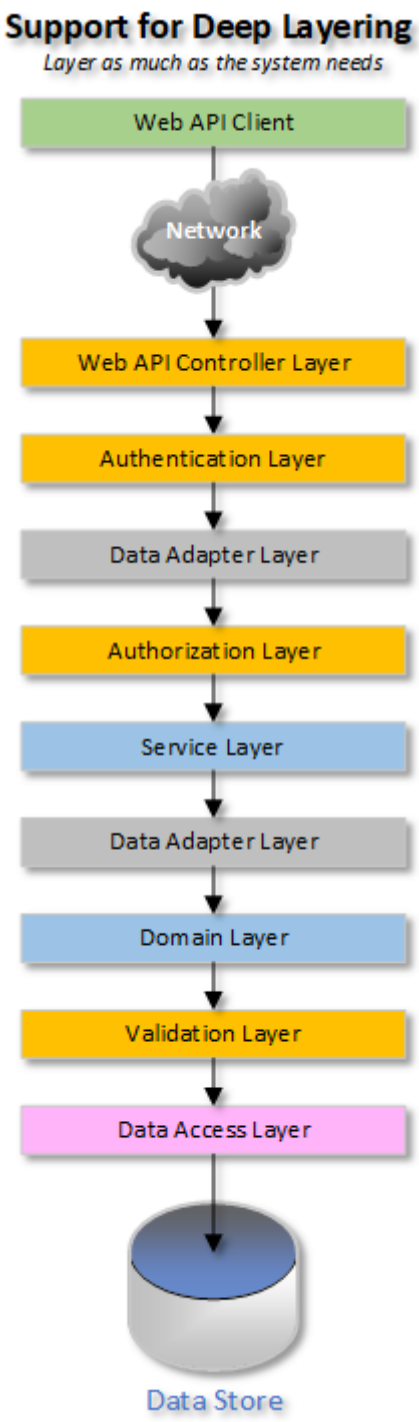
```
var config = new MyLayerConfiguration(new MyAuthorizationService(), new MyMapperService());

// This configuration step is completely optional, skipping it will allow any type of layer to be used.
config.ConfigureAllowedLayerTypes(OcLayerType.Api, OcLayerType.Adapter, OcLayerType.Business, OcLayerType.Repository);

// This configuration step is completely optional, skipping it will allow layer type in the system to access any other layer type.
config.ConfigureLayerRestrictions(OcLayerType.Api, OcLayerType.Adapter);  // Restricts API layers so that they can only communicate with Adapter layers
config.ConfigureLayerRestrictions(OcLayerType.Adapter, OcLayerType.Adapter, OcLayerType.Business);  // Restricts Adapter layers to Adapters and Business layers
config.ConfigureLayerRestrictions(OcLayerType.Business, OcLayerType.Business, OcLayerType.Repository);  // Restricts Business layers to Business and Repositories
config.ConfigureLayerRestrictions(OcLayerType.Repository);  // Restricts Repository layers so that they can not access any other layer (including other Repositories)

// The below registers the various layers available to the system and provides a factory method for constructing them when needed
// Note that it would be possible to use a standard IoC container implementation to handle construction.

config.RegisterLayer(SystemConstants.Resource_Customer, OcLayerType.Api, request => new MyCustomerApiController(config));
config.RegisterLayer(SystemConstants.Resource_Customer, OcLayerType.Adapter, request => new MyCustomerApiAdapter(config, request));
config.RegisterLayer(SystemConstants.Resource_Customer, OcLayerType.Business, request => new MyCustomerBusinessLogic(config, request));
config.RegisterLayer(SystemConstants.Resource_Customer, OcLayerType.Repository, request => new MyCustomerRepository(config, request));
```



There will be additional libraries that work with the Layers architecture and implement support for working with other common technologies.

- Library for working with ASP.NET MVC that implements the details necessary for Web API controllers
- Library that provides a standard Repository implementation that greatly simplifies working with Entity Framework.
- Libraries for common IoT containers (such as Unity) to construct layer and other object instances when needed.
- Support for Authentication and Authorization using common technologies and libraries.

To get a sense of what can be done using the Layers architecture with very minimal code take a look at the example code in the LayerApiMockup project which is currently included in the Original Coder Libraries.

At present all of the Original Coder libraries are in EARLY ALPHA. They are largely based on code in my personal libraries and on lessons learned working in a wide variety of environments. They are available in case anyone is curious or has ideas / feedback. I'm always open to suggestions.

Original Coder Blog
http://OriginalCoder.dev

Original Coder Libraries
https://github.com/TheOriginalCoder/Original-Coder-Libraries