Orbit

Orbit is a set of tooling to assist with client-side and server-side synchronization of objects.

Both sets of tooling can be used independently, but provide a simple-to-implement sync process when used in conjunction.

Orbit.Client

`Orbit.Client` provides a way to synchronize local values for objects with latest copies of objects as provided by a data provider.

Client Setup

The orbit client itself is intended to be registered as a singleton instance and to use that same instance through out the app. This can be done a number of ways, but it is likely that you will want to register it with a Inversion of Control (IoC) framework.

```
new OrbitClient()
```

The client itself is setup to be registered for use and it provides a series of fluent functions to get the client setup.

The first function that needs to be called is `Initialize`. Initialize needs to be passed in a directory where the sync cache will be stored. In the below example, we are using Xamarin.Essentials to get the data storage directory for our app. It is important to call this function before any of the other setup functions as it ensures that all of our core data components are available. Calling `Initialize` multiple times does not come at any penalty.

After the client has been initialized, we need to register objects that we are interested in. In the below example, we add type registrations for two objects `Post` and `User`. A type registration for the orbit client requires two things: the object you want to persist and a pointer to a property which will be treated as the unique identifier for the object.

```
var orbitClient =
    new OrbitClient()
        .Initialize(Xamarin.Essentials.FileSystem.AppDataDirectory)
        .AddTypeRegistration<Post, string>(x => x.Id)
        .AddTypeRegistration<User, string>(x => x.Username);
```

The call to `AddTypeRegistration` provides all of the information that the orbit client needs to persist your objects. In this example, `AddTypeRegistration<User, string>(x => x.Username)` will register the `User` object. The parameter supplied, in this example `x => x.Username`, is an expression that points to the unique identifying property for that object. Orbit needs that information to assist with indexing and resolution of objects later on. It is worth pointing out that the orbit client does not provide any build-in mechanisms for populating that unique identifier, so please do ensure that your process is populating that property. Additionally, the property that is used as the unique identifier for a project will have the `ToString` function called on it, so ensure that the value provided by that call is what you would expect.

Once the client has been initialized and had all of the objects of interest registered with it, you are now ready to use the client to get objects. As stated earlier, it would be recommended to register your client instance with an IoC framework. Below is an example of how this could be done with the `Splat` service locator.

```
Locator.CurrentMutable.RegisterConstant<OrbitClient>(orbitClient);
```

Typed Object Setup and Population

The orbit client is meant to be used in conjunction with data server, such as an API or remote database client. As such, we will need to depend on those outside systems to retrieve our baseline or core data that we will use to prime the cache.

In the example below, we are going to use an `HttpClient` to call out to an API and retrieve our user objects. Once we have our objects retrieved from the server, we can use them to populate our cache using the `PopulateCache` function call.

```
var client = Locator.Current.GetService<OrbitClient>();

var httpClient = new HttpClient();

var usersJson = await
httpClient.GetStringAsync("https://jsonplaceholder.typicode.com/users").Co
nfigureAwait(false);
var users = JsonConvert.DeserializeObject<IEnumerable<User>>(usersJson);

await client.PopulateCache(users);
```

Populating the cache is fairly important as it provides our client with the baseline set of objects that it knows about. Going forward, any time we create or modify an object and queue that change, it will be basing those changes off of the baseline object that was provided during the `PopulateCache` call.

It is possible that you may have multiple uses for the same object type. For example, perhaps you have `Users` that are in management and some that are peons. `PopulateCache` can work with the different types of users by providing the `category` with your call. It is important that the category name that you use can be referenced later on as you will need it going forward to get the cached objects going forward.

```
await client.PopulateCache(users, category: "peons");
```

By default, calling `PopulateCache` will not clear out any changes that have been built up during application operation. If you would like to fully clear out any outstanding local changes to objects, set the value `terminateSyncQueueHistory` to `true` when you call `PopulateCache`.

```
await client.PopulateCache(users, terminateSyncQueueHistory: true);
```

Ideally, after the first full sync with a server, you will want the local client to keep itself in the most locally accurate state going forward. This is accomplished by using the `Reconcile` function. This function will be passed any new feed of objects that we receive from the data server.

```
await client.Reconcile(updatedServerUsers);
```

The objects passed to this method needed to be wrapped individually with a `ServerSyncInfo<T>` wrapper object. This object provides meta information for what the last known server side modification was. This information will come out-of-the-box with the `Orbit.Web`, but can still be manually implemented when an outside system is being integrated, assuming it can provide all of the meta data needed to perform a reconciliation.

```
public class ServerSyncInfo<T>
{
    public ServerOperationType Operation { get; set; }

    public string Id { get; set; }

    public long ModifiedOn { get; set; }

    public T Value { get; set; }
}
```

For example, the next time the application calls to the API it will provide a result set of objects. The returned result set may contain objects that are in all different states including new, updated, deleted, etc. At the same time, our application will have had its own set of objects that have had their state modified as well. We need functionality to help our application manage which of these objects we trust to be the most accurate.

This process can be quite tricky as the business rules surrounding it can get quite complex. Some typical ways to treat this include looking at last updated times, determining a *source of truth* system and only taking updates from that system, or performing deep data inspections.

The job of the `Reconcile` function is to use a sync reconciler,implemented using the `ISyncReconciler` interface, which provides an implementation of the business logic that is used to reconcile objects for this app. For each server object that is passed into our the sync reconciler, we will determine if there is a local object, and then perform a reconciliation process to determine if we should keep the local information that we have about the object or replace it will the server version of the object.

A simple sync reconciler `ServerWinsSyncReconciler` is provided with the implementation which provides a way to replace any local updates with server updates. Custom ISyncReconcilers can be created as well so that you may provide any custom logic that the app needs.

Working with Cache Objects

Now that the cache has been populated, we can start interacting with objects.

Adding, Updating, and Deleting Objects

If you are adding a new object to the sync cache, you can call the `Create` function and pass in your current object. Ensure that the object type you are inserting has been registered with `AddTypeRegistration`.

```
await client.Create(myObject);
```

If you have updated an object, you can send the updated object to the sync cache using the `Update` call.

```
await client.Update(myObject);
```

With both `Create` and `Update`, you can run into issues if the object already exists or perhaps doesn't. In the case where you would like the framework to handle doing the *right thing*, you can just call into `Upsert` and the framework will handle the rest.

```
await client.Upsert(myObject);
```

If you would like to delete an object, call `Delete`. Now, this does not actually remove the object locally, but it does flag it as being deleted. This can be used later on when syncing back to the origin.

```
await client.Delete(myObject);
```

Retrieving

To get all objects out of the cache, you will want to make the call to `GetAllLatest`. This will get the most recent version of all of a particular type of local object. The returned result here is comprehensive including the most recent version of an object updated in the local cache and any original versions that were inserted during the `PopulateCache` call which are still unmodified. So, this is the *most accurate* verison of the local data that your application has.

```
await client.GetAllLatest<User>();
```

If you want to get the latest local object from your cache, you can either call `GetLatest` using an instance of that object type or by passing in the string version of that object's id.

```
await client.GetLatest<User>(myUserObj);
await client.GetLatest<User>("UserIdValue");
```

If you just want to get any objects that have been created or updated locally, you can retrieve them using the `GetAllLatestSyncQueue` call. If there have been no local modifications to objects, this could return an empty list of objects.

This function is the primary way to get data that you will want to sync back up to your origin/server. This will be the most recent set of local modifications to objects and is likely what we want to present server-side.

```
await client.GetAllLatestSyncQueue<User>();
```

Alternatively, you can call `GetSyncHistory` to get the history of an individual object or list of objects.

If you call this function and pass in an individual object's identifier, it will return the full local history of that object. This can be helpful if you needed to review any changes made to the object over time or if you wanted to rollback changes.

```
await client.GetSyncHistory<User>(myUserObj.UserId)
```

If you want to get all messages that have a change history, just call `GetSyncHistory` without any parameters. This will get you the latest local change for objects. So, for example, if you made five changes to an individual object, it will only notify you of the latest change to that object. If you would like to get all changes returned in chronological order, then pass in the `SyncType.FullHistory` enum to the function. This can be helpful if you wanted to sync all changes back the origin including any incremental changes that were made to all of the objects.

```
await client.GetSyncHistory<User>()
await client.GetSyncHistory<User>(SyncType.FullHistory)
```

Managing Local State

The easiest way to clear out the locally cached change/sync queue is to call `TerminateSyncQueueHistory`. Any objects of that type can be deleted using an Id, an instance of the object itself or without a parameter which will clear out the entire queue.

If the local cache of original objects needs to be modified, use one of the cache management functions.

The `DropCache` function can be used to clear out the entire cache of original objects of a type. Calling this will clear out all local originals, but will not impact the sync queue.

`DeleteCacheItem` and `UpsertCacheItem` can be used to modify individual cached items. This can be helpful if you receive an individual notification from the origin server that an object had changed and the application will change that object manually.

Orbit.Web

`Orbit.Web` provides a simple way to provide synchronization API components made for use with the Cosmos DB engine. With just simple configuration, a full featured API can generated that will provide all the syncing endpoints that your app needs.

Cosmos DB Configuration

In order to appropriately use the `Orbit.Web` component, you will need to have a pre-existing Cosmos DB setup and have it setup for use in an ASP.NET core application.

For instructions on how to Create a new Cosmos DB database are below.

Azure Portal

ASP.NET Core

Please ensure that you have created your Cosmos DB instance and have all of the appropriate information needed to connect to the database including:

- Database URI
- Read-Write Keys
- Database Id

Integrating Orbit.Web

With your database setup, you can now start integrating the components into your application.

We will do this by modifying the `ConfigureServices` function within the `Startup` class. Within this function, we will need make two additional calls to register services needed for the API components.

The first service registration is called using the `AddDefaultOrbitSyncCosmosDataClient` extension method. This method will require all of the cosmos connection information gathered earlier which consists of the Database URI, the authentication key, and the database id. Additionally, we will need to configure storage for all of the objects that we want to provide synchronization for.

```
services.AddDefaultOrbitSyncCosmosDataClient(cosmosUri, cosmosAuthKey,
cosmosDataBaseId, x =>
{
    Task.Run(async () =>
    {
        await x.EnsureCollectionAsync<Models.User>(y=> y.Username, y =>
y.Company.Name);
        await x.EnsureCollectionAsync<Models.Post>(y => y.Id, y =>
y.UserId);
    }).Wait();
});
```

The final parameter is an `Action` which is used to register types for the CosmosDB. In the example above, this function is seen providing object registration for `User` and `Post` objects. This registration takes two parameters. The first being an expression that returns the ID property for the object. This is similar to the `AddTypeRegistration` function that we used in the `Orbit.Client` and will need to register the same property as used on that side. The second parameter is an expression that instructs CosmosDB on how to partition this data. The partition property is usually a shared property value that some, but not all objects will have in common. For example, if you were setting up registration for an `Address` object, you may want to use the `City`, `State`, or `ZipCode` properties to partition your objects out. Each partition can only hold up to a max size of 10GB worth of data.

With the data registrations complete, we will next want to have our application generate our API endpoints for us. This is done adding another service registration using the `AddOrbitSyncControllers` extension method.

```
services.AddOrbitSyncControllers(x =>
{
    x.EnsureSyncController<Models.User>();
    x.EnsureSyncController<Models.Post>(false);
});
```

This method takes a single parameter of an `Action` which is used to configure the API endpoints. In the example above, we are adding API controllers for `User` and `Post` objects with calls to `EnsureSyncController`. The `Post` object does not require authentication, so we can pass false to the `EnsureSyncController` function which will disable authentication on this controller.

Wrapping Up

Once this is done, you are have both the client and the server components setup and ready to integrate with each other in your application. The final steps would be to integrate the client into your application and to create an `HttpClient` that can connect to your API. An implementation of that is not provided as authentication schemes and business needs will vary from project to project.