TNet 3: DataNode

What are DataNodes?

`DataNode` is a class that can be used to store a variable amount of information. In simplest terms, a `DataNode` is similar to an XML tree -- each node can have children, and those children can have children. TNet's `DataNode` can serialize just about anything -- from your custom classes to entire game object hierarchies and prefabs.

Why were they created?

Typically when a Remote Function Call (RFC) is created to send data and it's usually written with a fixed number of parameters (for example Position, Rotation, and Health). As the development continues and requirements change, more information may need to be added to RFCs and/or saved with your character, such as the player's Faction for example.

In order to add a new faction parameter a new RFC would need to be created, and a previous version of the RFC would need to be kept for backwards compatibility. You don't want to lose previously placed characters in the world for example. This can quickly lead to quite a few additional functions throughout the development of the game, making code unnecessarily messy.

Another example would be saving player data in a file. Imagine a player file that stores player's inventory made up of 30 items. The game ships with the fixed 30 item slots saved in binary form for efficiency, then later down the line you want to add an additional 30 slots for the player's vault. Simply modifying '30' to '60' will break the ability to load previous saves, which would be bad. A more flexible approach is required.

DataNode solves these problems by containing a variable size structure made up of nodes, with each node remembering the type of the data inserted into it. With the trivial templated retrieval in the form of `node.GetChild<T>("name")`, getting your data back is as easy as it is to expand or remove nodes in order to keep up with the changing requirements.

To make it even more robust, `DataNode` can be serialized in text, binary and compressed formats. Its text form is readily modifiable with any text editor by hand, and compressed format uses LZMA-based compression for the ultimate reduction in file and packet size when required.

How are Data Nodes Used?

On a basic level a `DataNode` is queried using a `Get` or `set` functions. Each `DataNode` consists of two parts. A “Name” and a “Value”. The Name is a case sensitive string and is a single word, such as “Color”. The “Value” can be anything -- an integer, a float, a custom struct, a reference to a class… you decide.

The following functions are a basic subset to get you started. Take a look at DataNode.cs if you would like to see which other functions are available.

GetChild

GetChild can either return a `DataNode` or a single value if you include the templated type in angled brackets. If the value is `null`, an optional parameter can be specified to define a default which will be returned instead.

```
DataNode myColorDataNode = node.GetChild("myColor");
Color myColor0 = node.GetChild<Color>("myColor");
Color myColor1 = node.GetChild<Color>("Path1/Color", Color.blue);
```

AddChild

AddChild adds a new `DataNode` to the list of DataNodes without doing any checks. This means, that it can add two identical nodes.

```
node.AddChild("Color1", Color.red);
node.AddChild("Color1", Color.green); // Will add a new value also named “Color1”
```

SetChild

Adds a child to the `DataNode` list, but will first check to see if one exists. If it exists, it will replace the first occurrence with this new value.

```
node.SetChild("Color1", Color.red);
node.SetChild("Color1", Color.blue); // Will replace the existing “Color1” value
```

SetHierarchy

`SetHierarchy` operates similar to `SetChild`, but it parses the path information and splits each path segment into a DataNode with children.

```
node.SetHierarchy("Path1/Color", Color.blue);
```

Results in this structure:

```
node value:null
      Path1 value:null
            Color value:Color.blue
```

Replacing a RFC parameters with a DataNode

It may be a good idea to replace some RFCs with a DataNode instead of a fixed number of parameters. The advantage of a DataNode-based approach is twofold: first, it allows the data to be completely flexible. You can add and remove elements without causing errors loading previously saved data. Second -- you can omit parameters by specifying default values in the `GetChild` function. This lets you send more or less data, depending on what changed.

Before:

```
[RFC]
void OnSyncMarine (Vector3 pos, Quaternion rot, float health)
{
	transform.position = pos;
	transform.rotation = rot;
	mHealth = health;
}
```

After:

```
[RFC]
void OnSyncMarine (DataNode node)
{
	transform.position = node.GetChild<Vector3>("position", transform.position);
	transform.rotation = node.GetChild<Quaternion>("rotation", transform.rotation);
	mHealth = node.GetChild<float>("health", mHealth);
	mFaction = node.GetChild<int>("faction", mFaction); // New addition
}
```

Although note that you can achieve a similar flexible effect by passing a custom class to your RFC. TNet's serialization will use reflection to get/set values, allowing you to add and remove class elements without worrying about breaking backwards compatibility. You can even control what gets saved and what doesn't by adding the `[IgnoredByTNet]` attribute to fields, or by implementing the `IDataNodeSerializable` interface in your class.

Where are DataNodes used?

There are three main areas of TNet which use DataNodes.

- Server Data
- Channel Data
- Player Data

Reading and Writing Server Data

Data can be written to the server by a player that has authenticated themselves as an administrator and is typically used to configure the server for your game. Any client connected to the server, even if they aren't in a channel, can read this data.

There are two ways to write the server data. A full string or two variables (path and value). If a string is passed in, TNet will attempt to parse it by splitting the string at an equal sign. If you're not sure of the formatting, the string it accepts matches what `DataNode` outputs when .ToString() is called.

```
TNManager.SetServerData("myCustomVar = 42");
TNManager.SetServerData("myCustomFloat", 12.5f);
```

It can be read back all at once, by looking at the `.serverData` parameter. This returns a TNet list which can be looped through to check each value.

```
TNManager.serverData
```

Or, you can read one variable in particular by passing the path to `GetServerData`. Optionally, you can also specify the `Type` of the variable if you know what it is.

```
DataNode node = TNManager.GetServerData(path));
Color myColor = TNManager.GetServerData<Color>("MyColor");
int myInt = TNManager.GetServerData<int>("Path/To/Node");
```

By default the person starting the server is an admin, but new players can be added by calling `.SetAdmin`.

```
TNManager.SetAdmin("passKey");
```

And they can be removed with a very similar command.

```
TNManager.RemoveAdmin("passKey");
```

If the passKey matches a string in `ServerConfig\Admin.txt` on the server, then this person will become a server admin.

If you would like more information on how to use the server data functions, the Chat example included with TNet demonstrates how it is used.

Reading and Writing Channel Data

Once the player joins a channel they are able to write to the `DataNode` associated with it using `TNManager.SetChannelData`:

```
TNManager.SetChannelData("name", "My Channel");
TNManager.SetChannelData("Path/To/Node", <SomeValue>);
```

This information can be read using the `GetChannelData` command. If a player belongs to more than one channel it will require the channel number to be specified as the first parameter. And an optional default value is also accepted.

```
DataNode node = TNManager.GetChannelData("Color");
Color c = TNManager.GetChannelData<Color>(42, "Color", Color.red);
```

If the channel is marked as persistent, the channel data will automatically be written to the hard drive when the server gets shut down. When the server is restarted it will be loaded back up. If the channel isn't persistent, the channel's data will be discarded when the last player leaves that channel.

If you would like more information on reading and writing channel data, please take a look at the Custom Channel Data tutorial.

Reading and Writing Player Data

Every player also has its own `DataNode` associated with it which can be used to record health, equipment, state, etc. `TNManager.SetPlayerData` is what is used to write the data to the server, and each line will trigger an `TNManager.onSetPlayerData` callback event.

Calling `TNManager.SetPlayerSave` will set the server-side filename that should be associated with the player. The server will load this file's contents, and will automatically save the player's data into this file every time it changes.

```
TNManager.SetPlayerSave(SystemInfo.deviceUniqueIdentifier + "/" +
TNManager.playerName);
```

The deviceUniqueIdentifier is a string unique to the computer it's used on. If you log back in using the same computer, you will be able to retrieve the data back. However, you probably want to base it off of something like a login/password combination or a steam id. That way a player can switch computers and not lose any of their data. The device identifier can change if the hardware changes, or even if a major OS update happens.

Call SetPlayerSave the first thing after connecting to the server. You can also set your own player's data at any time using:

```
TNManager.SetPlayerData("Color", Color.yellow);
TNManager.SetPlayerData("Path/To/Node", 123);
```

Reading a `DataNode` is also done in a similar way to the previous commands listed. For yourself:

```
Color c = TNManager.GetPlayerData<Color>("Color");
int val = TNManager.GetPlayerData<int>("Path/To/Node", 123);
```

For other players:

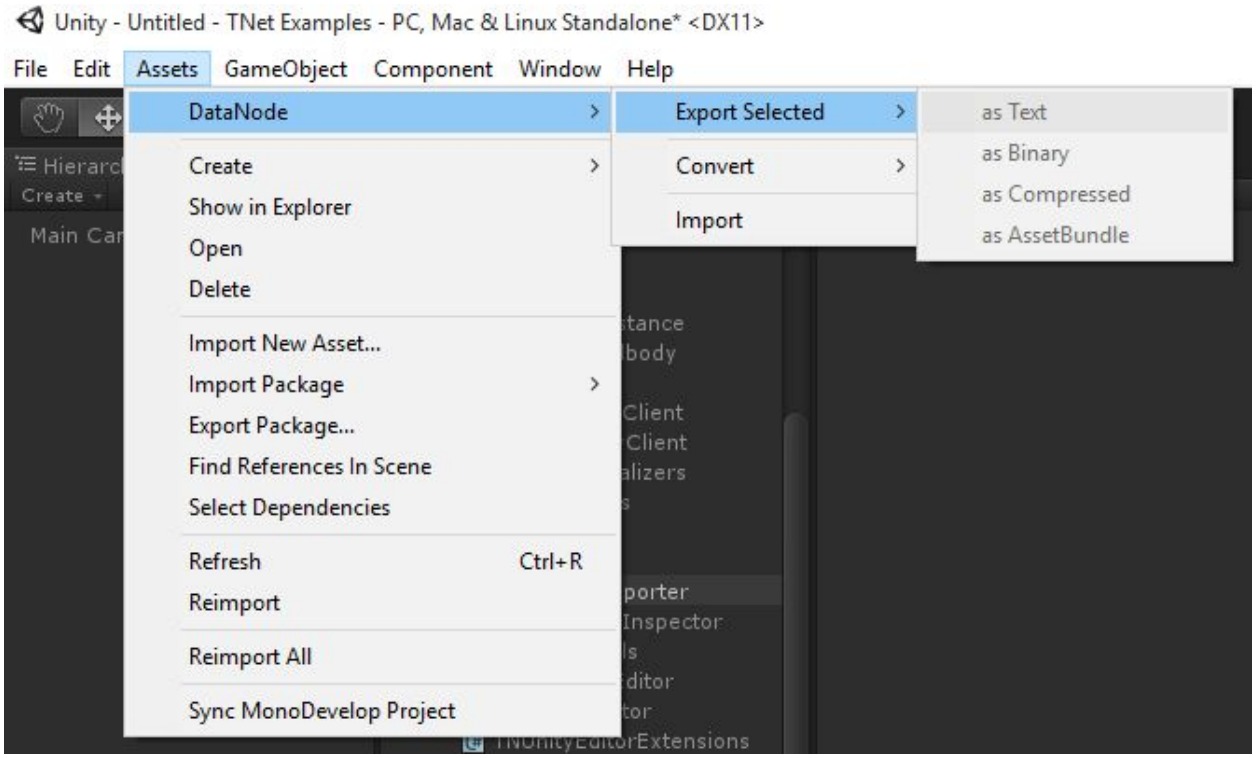
```
Color c = player.Get<Color>("Color");
int val = player.Get<int>("Path/To/Node", 123);
```

If you would like more information on the PlayerData functionality please take a look at the Synchronizing and Saving Player Data tutorial.

Saving Entire Object Hierarchies

DataNodes are also able to save a lot more than just basic data. They can load and save objects much like prefabs. This is perfect for making modded content. When TNet is included in a project it adds a DataNode sub-menu to the Assets dropdown. If you select a prefab, you can export it as a text file.

Asset bundle creation process in Unity 5 is different from Unity 4. To enable Unity 4-style Asset Bundle export option in the export menu, add the ASSET_BUNDLE_EXPORT to the Scripting Define Symbols in Edit -> Project Settings -> Player -> Other Settings.



It supports exporting/importing almost all information on prefabs. The only functionality it doesn't include is dictated by limitations in what Unity exposes, such as example particle system parameters and unique shader source files. DataNode export will keep references to everything in the Resources folder rather than exporting that content -- so any persistent textures you have, shaders, additional prefab references etc should be placed into the Resources folder in order to greatly reduce the size of the exported file. You can also do it via code:

```
DataNode node = gameObject.Serialize();
node.Write("filename.bytes", false);

DataNode myLoadedObject = DataNode.Read("filename.bytes");
GameObject go = myLoadedObject.Instantiate();
```

Also note that TNet's native object instantiation via `TNManger.Instantiate` will treat all DataNode-exported prefabs as if they were native Unity prefabs -- so whether you keep your prefabs in their native .prefab form or DataNode exported .bytes, TNManager.Instantiate will understand them both.

When adding **modding support** to your game, overwrite the `UnityTools.LoadBinary`, `LoadResource` and possibly `LoadResourceEx` delegates with something that will look for the requested data in your game's mod folder, and TNManager.Instantiate will be able to load DataNode-exported objects in your mod folders as well.

You can even use the TNManager.SaveFile function to store files on the server:

```
TNManager.SaveFile("filename.bytes", node.ToArray());
```

You can then load server-side files using:

```
TNManager.LoadFile("filename.bytes", delegate (string filename, byte[] data)
{
	DataNode myLoadedObject = DataNode.Read(data);
});
```

A good example demonstrating how this part of the `DataNode` can be used in your game, is in this Windward example video on how to add a helicopter mod to the game. At the 31:35 mark it demonstrates how to export a ship to a file and around the 50 minute mark how to place the exported helicopter on the server side for TNet to load. In Windward the files uploaded to the server leave an appropriate entry via SetServerData, so when clients connect to the server they get a list of all the uploaded files and proceed to download them before allowing the player to continue… but more on that in the future modding tutorial.