MEZZANINE

AUTO-SAVE

SIMPLE
REACTIVE
DATA

C#
MZ/SRD

Introduction

Mezzanine Auto-Save is a reactive data-modeling system that allows you to define, load, modify, and save game data, using reactive models that can be monitored for changes and auto-saved when changes are detected.

SIMPLE REACTIVE DATA MODELS

- Define your models as you normally would, with straight-forward, easily maintainable C# classes, and you're done. No cruft.
- All runtime changes to the models flow in one direction, through a single channel, providing a "single source of truth" that can be monitored for debugging.
- Any entity interested in the data is notified of all changes, and may respond to changes at any level of the data hierarchy.

NO COMPLICATED BOILERPLATE

- No need to define separate action dispatchers, or anything outside of your simple class.
- No complicated hoops to jump through to make runtime changes to your models.
- Simply call the provided `Set` method on your model. That's it. Changes are even auto-saved by default!

EASY LOADING AND SAVING

- Including auto-save.
- Zero-configuration with sensible defaults.
- Easily save and load the data associated with any entity in the system, on any platform.
- Entity-level auto-save options.
- Serialization format options include JSON, XML, and binary.

In addition, **Mezzanine Auto-Save**, like all **Mezzanine** products, is open source. It's comprised of simple, well-organized and documented code. So, you can customize it any way you like to meet your needs.

While **Mezzanine Auto-Save** was developed for and tested with Unity, it can be used in any C# project, and holds no external dependencies.

Define a model

> Define your model with a simple class:

```
[Serializable]
public class Character
{
    public string Name { get; private set; }
    public int Health { get; private set; }
    public int Stamina { get; private set; } = 100;

    public Character() {}
    public Character(string name, int health = 100)
    {
        Name = name;
        Health = health;
    }
}
```

C# CLASS AS MODEL DEFINITION

Defining models for your game entities is very simple. Just define a class describing the properties of your entity, as you normally would.

There are three things to keep in mind when defining classes that you intend to use as data models, in order for **Mezzanine Auto-Save** to create immutable clones of your model at runtime:

- Tag your class with the `[Serializable]` attribute.
- You can have multiple constructors for your class, but it needs to have a default constructor (with no parameters).
- And, for each of the properties you wish to expose, provide a private setter.

❗ **IMPORTANT:** Tag your class with the \`[Serializable]\` attribute. Provide a **default constructor** and **private setters**.

ℹ **NOTE:** XML is supported, but not recommended. See the **Using XML** section below for details.

Initialize a model

> To create a new data model from your class, remember to include the `Mz.Models` namespace. Then, instantiate your model, like this.

```
var model = new Model<Character>();
```

> If you like, you can pass in an object instance which will be used to initialize the new model.

```
var data = new Character("Tom", 50);
var model = new Model<Character>(data);
```

> Or, use the provided static factory method.

```
var model = Model.Create<Character>();
var modelTom = Model.Create(new Character("Tom", 50));
```

> An `Initialize` method is provided, in case you ever need to reset all the properties of a model.

```
model.Initialize(data);
```

To create a new model from your definition, simply call `new Model<TClass>()`, where `TClass` is the type of your model definition class.

If you'd like to initialize the model with a set of values, pass an instance of your definition class to the constructor, as shown to the right.

ℹ Calling `Initialize()` won't trigger a `Changed` event on your model.

Load data from a file

> Data can be loaded from Assets/Resources, or from Unity's persistent data storage location.

```
var saveOptions = new ModelSaveOptions("packs");
if (isFirstUse) model = Model.LoadResource<GameData>(
    "index",
    "Packs",
    saveOptions
);
else model = Model.Load<GameData>("packs", saveOptions);
```

A common scenario is to load a JSON file into a model from the `Assets/Resources` folder the first time a player uses your appliction. The initial model, once saved, will be serialized to a file in a persistent data location on the user's computer, as defined by Unity. Then, the next time the saved data is loaded, it will be retrieved from this persistent data location.

The `LoadResource` method provides a simple way to load the initial data from the `Assets/Resources` folder. Simply provide the file name (without an extension) as the first argument. Additional optional parameters allow you to specify a sub-directory path, the data format, and a custom unique identifier. You can also provide a `ModelSaveOptions` instance to customize auto-save options.

Call the `Load` method to retrieve a saved file from persistent data storage, and load it into a new model.

Access a model's data

> To access the data in your model, just use the Data property:

```
var name = model.Data.Name;
```

> Model definition classes can contain aggregate objects, arrays, or collections. And, these can be accessed the same way.

```
var ammo = model.Data.
    Weapons["sniper rifle"].
    Ammo.Current;
```

The `Data` property holds the current state of the model's associated data in read-only form, so it's safe to access this property directly. In fact, `Data` actually returns a mutable clone of the definition class instance held by the model, so there's no danger of munging the source data by mistake. In order to modify the data, you'll be using the `Set' method, as shown below.

Change a model's data

> Use the Set method to modify a property in the data hierarchy. Notice that this method takes an expression as the first argument, allowing you to specify the path to the data you want to modify.

```
model.Set(data => data.Name, "Felix", true);
```

> Aggregate object properties can also be modified in this way.

```
model.Set(data => data.Pets["Sven"].
    Armor.Head.Durability, 20);
```

> You can chain set operations.

```
var newName = player.Model
    .Set(data => data.Name, "Billy")
    .Set(data => data.Age, 32)
    .Data.Name;
```

> And, call multiple set operations asynchronously, then wait for the final model to be re-built.

```
model.Set(data => data.Name, "Billy");
model.Set(data => data.Age, 32);
await model.Build();
```

The model will always maintain the immutability of the underlying data object, ensuring that any and all changes to the data flow along a single path. This allows us to avoid bugs, like race conditions, that arise when data can be modified from multiple places at once. It also allows us to listen for changes to the data and respond accordingly in our game. And, we can easily monitor any and all changes to our data from a single place for debugging.

In order to change the data associated with our model, we need to call the `Set` method, as shown to the right.

The first paramater of the method accepts a lambda function, where the parameter of the function will be automatically injected with the Data property of the model. So, you can use this argument to access and modify any property in the model's object hierarchy. The second parameter of the `Set` method accepts the value that you wish to apply to the property.

An additional benefit of using the `Set' method to modify our data, is that we can choose to apply multiple changes asynchronously. We can then wait for the model to be re-built, so that we can respond to any changes that were made.

IMPORTANT: An auto-save event will only be triggered if the third parameter of the 'Set()` method is set to true

NOTE: You can use the `Set` method to modify any property in the model's object hierarchy. However, in order to add or remove elements from collections contained within the hierarchy, you should use the `Evaluate` method, as illustrated in the **Evaluate** section below.

Listen for changes

> Listen for changes to the data.

```
model.Changed += (
    object senderModel,
    ModelChangedEventArgs args
) =>
{
    Console.WriteLine(
        $"Number of changes made: {args.Changes.Count}"
    );

    foreach(var change in args.Changes)
    {
        Console.WriteLine(
            $"Property changed: {change.PropertyPath}"
        );
        Console.WriteLine(
            $"Current value: {change.ValueCurrent}"
        );
        Console.WriteLine(
            $"Previous value: {change.ValuePrevious}"
        );
    }
};
```

> Listen for save events:

```
model.Saved += (
    object senderModel,
    SerializeResult serializeResult
) =>
{
    Console.WriteLine(
        $"Model {senderModel.Key} saved"
    );
    Console.WriteLine(
        $"Serialization was successful: {serializeResult.IsSuccess}"
    );
    Console.WriteLine(
        $"{serializeResult.FilePath}"
    );
};
```

The `Changed` event is invoked on a model whenever the underlying data is modified using the `Set` method. The arguments that are passed to any associated event listeners hold information about each of the properties that was changed, each property's relative path (if the property belongs to an aggregated object), as well as both the current and previous values of the property.

The `Saved` event invoked whenever a save action is completed, whether it's an automatic or manual operation.

Evaluate custom actions

```
model.Evaluate(
    data => data.Weapons.Add(new Weapon(WeaponType.Shotgun))
);
```

If you need to perform an action on a model other than modifying a property, you can use the `Evaluate` method. An example would be adding or removing elements from a collection. As with the `Set` method, you can specify whether or not you'd like change and save actions to be invoked.

Using XML

> Deserializing from XML requires the use of attribute tags and public property setters:

```
[Serializable]
public class GameData
{
    [XmlArray("Packs")]
    [XmlArrayItem("Pack")]
    public PackData[] Packs { get; set; }

    [XmlAttribute("PackCurrent")]
    public int PackCurrent { get; set; }

    [XmlAttribute("LevelCurrent")]
    public int LevelCurrent { get; set; }
}
```

XML is supported but not recommended.

When we're using XML, rather than JSON, our data models are going to be a bit more complicated, since we need to use property attribute tags to describe how we want the data to flow into our models. Also, unlike JSON, XML doesn't support arrays, so we need to structure our models a little differently.

Internally, we use Microsoft's XmlSerializer class, which doesn't allow us to to deserialize our data into private properties. This means we need to declare our property setters as public, and we'll have to be careful not to use the setter directly to modify the data, which is a common source of bugs.

For this reason, I recommend using JSON, rather than XML, if you can. By marking the setters as private, you ensure that the data won't accidentally be modified directly. Your data object becomes effectively immutable, so all changes are constrained to a single channel that flows through your Model interface. This will eliminate a host of potential bugs, and allow you to monitor all your data changes for further debugging purposes. We can still take advantage of these benefits using XML, as long as we remember to only use the Set() method, rather than the property setter to modify the data.

- Tag your class with the `[Serializable]` attribute.
- Use attribute tags to describe the data.
- Declare all property setters as public.

Access the serializer

> Access the internal serializer:

```
Mz.Serializers.Data.SerializeToString(model.Data, out string resultString, true);
```

Mezzanine Auto-Save is built on top of a powerful custom, cross-platform serializer. You can access the serializer directly, if you need to support more specialized data management scenarios. For example, you might want to grab your game data as a JSON string and save it to an online database.

To access the serializer, simply reference `Mz.Serializers.Data`. For details on each of the supported methods, reference the source code at `Assets\Plugins\Mezz\Common\Serializers\`.

Performance

The code has been carefully optimized and tested to minimize performance overhead, but some optimizations are disabled by default in order to support all platforms and older versions of Unity.

If you're using Unity 2019, with a reference to the `System.Reflection.Emit` assembly, and you're not building for either iOS or WebGL, you can take advantage of performance optimizations provided by the .NET `DynamicMethod` class.

To enable these optimizations, go into the `Assets/Mezz/Common/TypeTools/Types.cs` file, and uncomment the `#define REFLECTION_EMIT` line at the top.

Unique ID's

By default, the `Key` property of a model holds an automatically generated unique identifier that can be used to identify specific models in various scenarios, such as when interacting with databases or URI's.

Custom keys can be supplied to models at instantiation.

Internally, **Mezzanine Auto-Save** uses `HashIds` to generate short unique ID's in the form `1PAw2zV5lp5MsP2NaOg`. The algorithm employed to generate the identifiers encodes the number of 100-nanosecond intervals that have elapsed since 12:00:00 midnight, January 1, 0001, then tacks on a random int to be sure we have a unique value, in case we're running on a very fast system.