Automatonymous Documentation

Release 2.0

Chris Patterson, Travis Smith, and Dru Sellers

January 10, 2017

Contents

Automatonymous is a state machine library for .NET developers. Automatonymous provides a fluent syntax for declaring state machines, including the states, events (both trigger and data events are supported), and state/event activities. While surprising easy to use for simple state machines, Automatonymous has many advanced features that make it usable in a variety of contexts.

Automatonymous is completely open source and licensed under the very permissive Apache 2.0 license, making usable at no cost to anyone for both commercial and non-commercial use.

Automatonymous is hosted on GitHub at: https://github.com/MassTransit/Automatonymous

Contents:

CHAPTER 1

Installing Automatonymous

The easiest way to install Automatonymous is with NuGet. Open the package manager and add Automatonymous to your project and the proper references will be added for you.

1.1 Requirements

Automatonymous requires .NET 4.5 (or later). Due to the extensive use of the TPL and *async/await*, it is not compatible with .NET versions prior to 4.5.

1.1.1 How to install

NuGet

The simplest way to install MassTransit into your solution/project is to use NuGet.:

```
nuget Install-Package MassTransit
```

Raw Binaries

If you are a fan of getting the binaries you can get released builds from

http://github.com/masstransit/MassTransit/downloads

Then you will need to add references to

- MassTransit.dll
- MassTransit.<Transport>.dll (Either MSMQ or RabbitMQ)
- MassTransit.<ContainerSupport>.dll (If you are so inclined)

Compiling From Source

Lastly, if you want to hack on MassTransit or just want to have the actual source code you can clone the source from github.com.

To clone the repository using git try the following:

```
git clone git://github.com/MassTransit/MassTransit.git
```

If you want the development branch (where active development happens):

```
git clone git://github.com/MassTransit/MassTransit.git
git checkout develop
```

Build Dependencies

To compile MassTransit from source you will need the following developer tools installed:

- .Net 4.0 sdk
- ruby v 1.8.7
- gems (rake, albacore)

Compiling

To compile the source code, drop to the command line and type:

```
.\build.bat
```

If you look in the `.\build_output` folder you should see the binaries.

CHAPTER 2

Getting Started with Automatonymous

Once you have added Automatonymous to your project (using NuGet or otherwise), you're ready to create your first state machine.

2.1 Creating Your First State Machine

A state machine is declared with Automatonymous using an internal domain specific language (DSL). To declare a state machine, add a class to your project that inherits from AutomatonymousStateMachine. For our example, we'll use the concept of a vehicle pit stop on the road of life.

```
public class PitStop :
  AutomatonymousStateMachine<VehicleState>
{
  public PitStop()
  {
    State(() => Running);
    Event(() => Start);

    Initially(
      When(Arrived)
        .TransitionTo(Waiting));
  }

  public Event<Vehicle> Arrived {get; private set;}

  public State Waiting {get; private set;}
}
```

So far, only a minimum level of functionality has been declared. When a vehicle arrives, the PitStop is in an initial state. When the Arrived event is raised, the vehicle information is passed to the handler, which transitions to the Waiting state.

2.1.1 Automatonymous Quick Start

So you've got the chops and want to get started quickly using Automatonymous. Maybe you are a bad ass and can't be bothered with reading documentation, or perhaps you are already familiar with the Magnum StateMachine and want to see what things have changed. Either way, here it is, your first state machine configured using Automatonymous.

```
1 class Relationship
2 {
```

```
    public State CurrentState { get; set; }
    public string Name { get; set; }
}

class RelationshipStateMachine :
    AutomatonymousStateMachine<Relationship>
{
    public RelationshipStateMachine()
    {
        Event(() => Hello);
        Event(() => PissOff);
        Event(() => Introduce);

        State(() => Friend);
        State(() => Enemy);

        Initially(
            When(Hello)
                .TransitionTo(Friend),
            When(PissOff)
                .TransitionTo(Enemy),
            When(Introduce)
                .Then(ctx => ctx.Instance.Name = ctx.Data.Name)
                .TransitionTo(Friend)
        );
    }

    public State Friend { get; private set; }
    public State Enemy { get; private set; }

    public Event Hello { get; private set; }
    public Event PissOff { get; private set; }
    public Event<Person> Introduce { get; private set; }
}

class Person
{
    public string Name { get; set; }
}
```

Seriously?

Okay, so two classes are defined above, one that represents the state (`Relationship`) and the other that defines the behavior of the state machine (`RelationshipStateMachine`). For each state machine that is defined, it is expected that there will be at least one instance. In Automatonymous, state is separate from behavior, allowing many instances to be managed using a single state machine.

Note: For some object-oriented purists, this may be causing the hair to raise on the back of your neck. Chill out, it's not the end of the world here. If you have a penchant for encapsulating behavior with data (practices such as domain model, DDD, etc.), recognize that programming language constructs are the only thing in your way here.

Tracking State

State is managed in Automatonymous using a class, shown above as the `Relationship`.

Defining Behavior

Behavior is defined using a class that inherits from `AutomatonymousStateMachine`. The class is generic, and the state type associated with the behavior must be specified. This allows the state machine configuration to use the state for a better configuration experience.

Note: It also makes Intellisense work better.

In a state machine, states must be defined along with the events that can be raised. In the constructor, each state and event must be explicitly defined. As each state or event is defined, the specified property is initialized with the appropriate object type (either a State or an Event), which is why a lambda method is used to specify the property.

Creating Instances

Creating the State Machine

Raising Events

Once a state machine and an instance have been created, it is necessary to raise an event on the state machine instance to invoke some behavior. There are three or four participants involved in raising an event: a state machine, a state machine instance, and an event. If the event includes data, the data for the event is also included.

The most explicit way to raise an event is shown below.

```
var relationship = new Relationship();
var machine = new RelationshipStateMachine();

machine.RaiseEvent(relationship, machine.Hello);
```

If the event has data, it is passed along with the event as shown.

```
var person = new Person { Name = "Joe" };

machine.RaiseEvent(relationship, machine.Introduce, person);
```

Lifters

Lifters allow events to be raised without knowing explicit details about the state machine or the instance type, making it easier to raise events from objects that do not have prior type knowledge about the state machine or the instance. Using an approach known as *currying* (from functional programming), individual arguments of raising an event can be removed.

For example, using an event lift, the state machine is removed.

```
var eventLift = machine.CreateEventLift(machine.Hello);

// elsewhere in the code, the lift can be used
eventLift.Raise(relationship);
```

The instance can also be lifted, making it possible to raise an event without any instance type knowledge.

```
var instanceLift = machine.CreateInstanceLift(relationship);
var helloEvent = machine.Hello;
```

```
// elsewhere in the code, the lift can be used
instanceLift.Raise(helloEvent);
```

Lifts are commonly used by plumbing code to avoid dynamic methods or delegates, making code clean and fast.

CHAPTER 3

Cry For Help

This documentation is stored in the GitHub repository, and as such can be forked, updated, and merged into the main project via pull request. So if you want to help out with the documentation, please do so!

CHAPTER 4

Indices and tables

- genindex
- modindex
- search