Akka.NET Cheatsheet

NUGET PACKAGES

Main: Akka - Akka.Persistence - Akka.Persistence.SqlServer
Cluster: Akka.Cluster - Akka.Cluster.Tools
Logging: Akka.Logger.NLog

ACTOR SYSTEM CREATION

```
var actorSystem = ActorSystem.Create("ActorSystemName");

//Create actorsystem using configuration in akka.conf file
var actorSystem = ActorSystem.Create("ActorSystemName", ConfigurationLoader.Load());

static class ConfigurationLoader
{
  public static global::Akka.Configuration.Config Load() => LoadConfig("akka.conf");

  private static global::Akka.Configuration.Config LoadConfig(string configFile)
  {
      if (File.Exists(configFile))
      {
          string config = File.ReadAllText(configFile);
          return global::Akka.Configuration.ConfigurationFactory.ParseString(config);
      }
      return global::Akka.Configuration.Config.Empty;
  }
}
```

CREATION OF ACTORS

```
// Top level actors
IActorRef myActor1 = system.ActorOf("NameOfMyActor1");

Props props = Props.Create();
IActorRef myActor2 = system.ActorOf(props, "NameOfMyActor2");

Props props2 = Props.Create(() => new MyReceiveActor());
IActorRef myActor3 = system.ActorOf(props2, "NameOfMyActor3");

SupervisorStrategy strat = null; // = new AllForOneStrategy(...);
Props props3 = Props.Create(() => new MyReceiveActor(), strat);
IActorRef myActor4 = system.ActorOf(props3, "NameOfMyActor4");

//Child actors
class MyReceiveActorWithChildren : ReceiveActor
{
  public MyReceiveActorWithChildren()
  {
    Context.ActorOf(Props.Create(), "AChild");
  }
}
```

Akka.NET Cheatsheet

DEFINING ACTORS

```
//Untyped actor
class MyUntypedActor : UntypedActor
{
  protected override void OnReceive(object message)
  {
    if (message is Order) { /*...*/ }
  }
}
//Receive actor
class MyReceiveActor : ReceiveActor
{
  public MyReceiveActor()
  {
    Receive(message => { /*...*/ });
    // Predicate message filters
    Receive(handleIf => handleIf.IsVip ,message => { /*...*/ });
    Receive(message => { /*...*/ }, handleIf => handleIf.IsVip);
  }
}
//Persistent actor
class MyPersistentActor : PersistentActor
{
  public override string PersistenceId => /*Sensible id value*/;

  protected override bool ReceiveCommand(object message)
  {
    //Handle incoming command (message)
  }

  protected override bool ReceiveRecover(object message)
  {
    //Handle state re-build using stored events
  }
}
```

MESSAGE STASHING

```
class MyStashingActor: ReceiveActor, IWithUnboundedStash
{
  public IStash Stash { get; set; }
  public MyStashingActor()
  {
    Receive(message => Stash.Stash());
    Receive(message => Stash.Unstash());
    Receive(message => Stash.UnstashAll());
  }
}
```

Akka.NET Cheatsheet

LOGGING

```
class MyLoggingActor : ReceiveActor
{
  private readonly ILoggingAdapter _log = Context.GetLogger();
  public MyLoggingActor()
  {
    _log.Debug("...");
    _log.Info("...");
    _log.Warning("...");
    _log.Error("...");
  }
}
```

LIFECYCLE EVENTS

```
class MyLifecycleActor : ReceiveActor
{
  protected override void PreStart()
  {
    base.PreStart();
  }

  protected override void PreRestart(Exception reason, object message)
  {
    base.PreRestart(reason, message);
  }

  protected override void PostRestart(Exception reason)
  {
    base.PostRestart(reason);
  }

  protected override void PostStop()
  {
    base.PostStop();
  }
}
```

Akka.NET Cheatsheet

PATTERN MATCHING

```
class BoleroDanceCommand : DanceCommand { }

class QuickStepDanceCommand : DanceCommand { }

class PatternMatchingActor : ReceiveActor
{
  public PatternMatchingActor()
  {
      Receive<DanceCommand>(Handle);
  }
  private bool Handle(DanceCommand command)
  {
    return command.Match()
      .With<BoleroDanceCommand>(cmd =>
      {
        DoBoleroDance();
      })
      .With<QuickStepDanceCommand>(() =>
      {
        DoQuickStepDance()
      })
      .Default(o =>
      {
        Context.System.Log.Warning("I suck at dancing.");
      })
      .WasHandled;
  }
}
```

FINITE STATE MACHINES (SWITCHABLE BEHAVIOURS)

```
class MyBehaviorActor : ReceiveActor
{
  public MyBehaviorActor()
  {
    // Initial behavior
    Happy();
  }
  private void Happy()
  {
    Receive(message => Become(Sad));
  }
  private void Sad()
  {
    Receive(message => Become(Happy));
  }
}
// Stacked behaviour API
Receive(message => BecomeStacked(Sad));
Receive(message => UnbecomeStacked());
```