Akka Modules Documentation

Release 1.3.1

Typesafe Inc

March 08, 2012

CONTENTS

CHAPTER

ONE

MODULES

1.1 Microkernel

1.1.1 Download Akka Modules

Download the full Akka Modules distribution from http://akka.io/downloads

1.1.2 Build latest version from source

To build the latest version see *Building Akka Modules*.

1.1.3 Run the microkernel

To start the kernel use the scripts in the `bin` directory.

All services are configured in the `config/akka.conf` configuration file. See the Akka documentation on Configuration for more details. Services you want to be started up automatically should be listed in the list of `boot` classes in the configuration.

Put your application in the `deploy` directory.

Akka Home

Note that the microkernel needs to know where the Akka home is (the base directory of the microkernel). The above scripts do this for you. Otherwise, you can set Akka home by:

- Specifying the `AKKA_HOME` environment variable
- Specifying the `-Dakka.home` java option

1.1.4 Hello Microkernel

There is a very simple Akka Mist sample project included in the microkernel `deploy` directory. Start the microkernel with the start script and then go to http://localhost:9998 to say Hello to the microkernel.

1.2 Camel

For an introduction to akka-camel, see also the Appendix E - Akka and Camel (pdf) of the book Camel in Action.

Other, more advanced external articles are:

- Akka Consumer Actors: New Features and Best Practices

- Akka Producer Actors: New Features and Best Practices

1.2.1 Introduction

The akka-camel module allows actors, untyped actors, and typed actors to receive and send messages over a great variety of protocols and APIs. This section gives a brief overview of the general ideas behind the akka-camel module, the remaining sections go into the details. In addition to the native Scala and Java actor API, actors can now exchange messages with other systems over large number of protocols and APIs such as HTTP, SOAP, TCP, FTP, SMTP or JMS, to mention a few. At the moment, approximately 80 protocols and APIs are supported.

The akka-camel module is based on Apache Camel, a powerful and leight-weight integration framework for the JVM. For an introduction to Apache Camel you may want to read this Apache Camel article. Camel comes with a large number of components that provide bindings to different protocols and APIs. The camel-extra project provides further components.

Usage of Camel's integration components in Akka is essentially a one-liner. Here's an example.

```
import akka.actor.Actor
import akka.actor.Actor._
import akka.camel.{Message, Consumer}

class MyActor extends Actor with Consumer {
  def endpointUri = "mina:tcp://localhost:6200?textline=true"

  def receive = {
    case msg: Message => { /* ... */}
    case _            => { /* ... */}
  }
}

// start and expose actor via tcp
val myActor = actorOf[MyActor].start
```

The above example exposes an actor over a tcp endpoint on port 6200 via Apache Camel's Mina component. The actor implements the endpointUri method to define an endpoint from which it can receive messages. After starting the actor, tcp clients can immediately send messages to and receive responses from that actor. If the message exchange should go over HTTP (via Camel's Jetty component), only the actor's endpointUri method must be changed.

```
class MyActor extends Actor with Consumer {
  def endpointUri = "jetty:http://localhost:8877/example"

  def receive = {
    case msg: Message => { /* ... */}
    case _            => { /* ... */}
  }
}
```

Actors can also trigger message exchanges with external systems i.e. produce to Camel endpoints.

```
import akka.actor.Actor
import akka.camel.{Producer, Oneway}

class MyActor extends Actor with Producer with Oneway {
  def endpointUri = "jms:queue:example"
}
```

In the above example, any message sent to this actor will be added (produced) to the example JMS queue. Producer actors may choose from the same set of Camel components as Consumer actors do.

The number of Camel components is constantly increasing. The akka-camel module can support these in a plug-and-play manner. Just add them to your application's classpath, define a component-specific endpoint URI and

use it to exchange messages over the component-specific protocols or APIs. This is possible because Camel components bind protocol-specific message formats to a Camel-specific normalized message format. The normalized message format hides protocol-specific details from Akka and makes it therefore very easy to support a large number of protocols through a uniform Camel component interface. The akka-camel module further converts mutable Camel messages into immutable representations which are used by Consumer and Producer actors for pattern matching, transformation, serialization or storage, for example.

1.2.2 Dependencies

Akka's Camel Integration consists of two modules

- akka-camel - this module depends on akka-actor and camel-core (+ transitive dependencies) and implements the Camel integration for (untyped) actors
- akka-camel-typed - this module depends on akka-typed-actor and akka-camel (+ transitive dependencies) and implements the Camel integration for typed actors

The akka-camel-typed module is optional. To have both untyped and typed actors working with Camel, add the following dependencies to your SBT project definition.

```
import sbt._

class Project(info: ProjectInfo) extends DefaultProject(info) with AkkaProject {
  // ...
  val akkaCamel = akkaModule("camel")
  val akkaCamelTyped = akkaModule("camel-typed") // optional typed actor support
  // ...
}
```

1.2.3 Consume messages

Actors (untyped)

For actors (Scala) to receive messages, they must mixin the Consumer trait. For example, the following actor class (Consumer1) implements the endpointUri method, which is declared in the Consumer trait, in order to receive messages from the `file:data/input/actor` Camel endpoint. Untyped actors (Java) need to extend the abstract UntypedConsumerActor class and implement the getEndpointUri() and onReceive(Object) methods.

Scala

```
import akka.actor.Actor
import akka.camel.{Message, Consumer}

class Consumer1 extends Actor with Consumer {
  def endpointUri = "file:data/input/actor"

  def receive = {
    case msg: Message => println("received %s" format msg.bodyAs[String])
  }
}
```

Java

```
import akka.camel.Message;
import akka.camel.UntypedConsumerActor;

public class Consumer1 extends UntypedConsumerActor {
  public String getEndpointUri() {
    return "file:data/input/actor";
  }
```

```
  public void onReceive(Object message) {
    Message msg = (Message)message;
    String body = msg.getBodyAs(String.class);
    System.out.println(String.format("received %s", body))
  }
}
```

Whenever a file is put into the data/input/actor directory, its content is picked up by the Camel file component and sent as message to the actor. Messages consumed by actors from Camel endpoints are of type Message. These are immutable representations of Camel messages.

For Message usage examples refer to the unit tests:

- Message unit tests - Scala API
- Message unit tests - Java API

Here's another example that sets the endpointUri to `jetty:http://localhost:8877/camel/default`. It causes Camel's Jetty component to start an embedded Jetty server, accepting HTTP connections from localhost on port 8877.

Scala

```
import akka.actor.Actor
import akka.camel.{Message, Consumer}

class Consumer2 extends Actor with Consumer {
  def endpointUri = "jetty:http://localhost:8877/camel/default"

  def receive = {
    case msg: Message => self.reply("Hello %s" format msg.bodyAs[String])
  }
}
```

Java

```
import akka.camel.Message;
import akka.camel.UntypedConsumerActor;

public class Consumer2 extends UntypedConsumerActor {
  public String getEndpointUri() {
    return "jetty:http://localhost:8877/camel/default";
  }

  public void onReceive(Object message) {
    Message msg = (Message)message;
    String body = msg.getBodyAs(String.class);
    getContext().replySafe(String.format("Hello %s", body));
  }
}
```

After starting the actor, clients can send messages to that actor by POSTing to `http://localhost:8877/camel/default`. The actor sends a response by using the self.reply method (Scala). For returning a message body and headers to the HTTP client the response type should be Message. For any other response type, a new Message object is created by akka-camel with the actor response as message body.

Typed actors

Typed actors can also receive messages from Camel endpoints. In contrast to (untyped) actors, which only implement a single receive or onReceive method, a typed actor may define several (message processing) methods, each of which can receive messages from a different Camel endpoint. For a typed actor method to be exposed as Camel

endpoint it must be annotated with the @consume annotation. For example, the following typed consumer actor defines two methods, foo and bar.

Scala

```
import org.apache.camel.{Body, Header}
import akka.actor.TypedActor
import akka.camel.consume

trait TypedConsumer1 {
  @consume("file:data/input/foo")
  def foo(body: String): Unit

  @consume("jetty:http://localhost:8877/camel/bar")
  def bar(@Body body: String, @Header("X-Whatever") header: String): String
}

class TypedConsumer1Impl extends TypedActor with TypedConsumer1 {
  def foo(body: String) = println("Received message: %s" format body)
  def bar(body: String, header: String) = "body=%s header=%s" format (body, header)
}
```

Java

```
import org.apache.camel.Body;
import org.apache.camel.Header;
import akka.actor.TypedActor;
import akka.camel.consume;

public interface TypedConsumer1 {
  @consume("file:data/input/foo")
  public void foo(String body);

  @consume("jetty:http://localhost:8877/camel/bar")
  public String bar(@Body String body, @Header("X-Whatever") String header);
}

public class TypedConsumer1Impl extends TypedActor implements TypedConsumer1 {
  public void foo(String body) {
    System.out.println(String.format("Received message: ", body));
  }

  public String bar(String body, String header) {
    return String.format("body=%s header=%s", body, header);
  }
}
```

The foo method can be invoked by placing a file in the data/input/foo directory. Camel picks up the file from this directory and akka-camel invokes foo with the file content as argument (converted to a String). Camel automatically tries to convert messages to appropriate types as defined by the method parameter(s). The conversion rules are described in detail on the following pages:

- Bean integration
- Bean binding
- Parameter binding

The bar method can be invoked by POSTing a message to http://localhost:8877/camel/bar. Here, parameter binding annotations are used to tell Camel how to extract data from the HTTP message. The @Body annotation binds the HTTP request body to the first parameter, the @Header annotation binds the X-Whatever header to the second parameter. The return value is sent as HTTP response message body to the client.

Parameter binding annotations must be placed on the interface, the @consume annotation can also be placed on the methods in the implementation class.

Consumer publishing

Actors (untyped)

Publishing a consumer actor at its Camel endpoint occurs when the actor is started. Publication is done asynchronously; setting up an endpoint (more precisely, the route from that endpoint to the actor) may still be in progress after the ActorRef.start method returned.

Scala

```
import akka.actor.Actor._

val actor = actorOf[Consumer1] // create Consumer actor
actor.start                    // activate endpoint in background
```

Java

```
import static akka.actor.Actors.*;
import akka.actor.ActorRef;

ActorRef actor = actorOf(Consumer1.class); // create Consumer actor
actor.start();                             // activate endpoint in background
```

Typed actors

Publishing of typed actor methods is done when the typed actor is created with one of the TypedActor.newInstance(..) methods. Publication is done in the background here as well i.e. it may still be in progress when TypedActor.newInstance(..) returns.

Scala

```
import akka.actor.TypedActor

// create TypedConsumer1 object and activate endpoint(s) in background
val consumer = TypedActor.newInstance(classOf[TypedConsumer1], classOf[TypedConumer1Impl])
```

Java

```
import akka.actor.TypedActor;

// create TypedConsumer1 object and activate endpoint(s) in background
TypedConsumer1 consumer = TypedActor.newInstance(TypedConsumer1.class, TypedConumer1Impl.class);
```

Consumers and the CamelService

Publishing of consumer actors or typed actor methods requires a running CamelService. The Akka *Microkernel* can start a CamelService automatically (see *CamelService configuration*). When using Akka in other environments, a CamelService must be started manually. Applications can do that by calling the CamelServiceManager.startCamelService method.

Scala

```
import akka.camel.CamelServiceManager._

startCamelService
```

Java

```
import static akka.camel.CamelServiceManager.*;

startCamelService();
```

If applications need to wait for a certain number of consumer actors or typed actor methods to be published they can do so with the `CamelServiceManager.mandatoryService.awaitEndpointActivation` method, where `CamelServiceManager.mandatoryService` is the current CamelService instance (or throws an IllegalStateException there's no current CamelService).

Scala

```
import akka.camel.CamelServiceManager._

startCamelService

// Wait for three conumer endpoints to be activated
mandatoryService.awaitEndpointActivation(3) {
  // Start three consumer actors (for example)
  // ...
}

// Communicate with consumer actors via their activated endpoints
// ...
```

Java

```
import akka.japi.SideEffect;
import static akka.camel.CamelServiceManager.*;

startCamelService();

// Wait for three conumer endpoints to be activated
getMandatoryService().awaitEndpointActivation(3, new SideEffect() {
  public void apply() {
    // Start three consumer actors (for example)
    // ...
  }
});

// Communicate with consumer actors via their activated endpoints
// ...
```

Alternatively, one can also use `Option[CamelService]` returned by `CamelServiceManager.service`.

Scala

```
import akka.camel.CamelServiceManager._

startCamelService

for(s <- service) s.awaitEndpointActivation(3) {
  // ...
}
```

Java

```
import java.util.concurrent.CountDownLatch;

import akka.camel.CamelService;
import static akka.camel.CamelServiceManager.*;

startCamelService();

for (CamelService s : getService()) s.awaitEndpointActivation(3, new SideEffect() {
  public void apply() {
    // ...
  }
});
```

CamelService configuration additionally describes how a CamelContext, that is managed by a CamelService, can be cutomized before starting the service. When the CamelService is no longer needed, it should be stopped.

Scala

```
import akka.camel.CamelServiceManager._

stopCamelService
```

Java

```
import static akka.camel.CamelServiceManager.*;

stopCamelService();
```

Consumer un-publishing

Actors (untyped)

When an actor is stopped, the route from the endpoint to that actor is stopped as well. For example, stopping an actor that has been previously published at `http://localhost:8877/camel/test` will cause a connection failure when trying to access that endpoint. Stopping the route is done asynchronously; it may be still in progress after the `ActorRef.stop` method returned.

Scala

```
import akka.actor.Actor._

val actor = actorOf[Consumer1] // create Consumer actor
actor.start                    // activate endpoint in background
// ...
actor.stop                     // deactivate endpoint in background
```

Java

```
import static akka.actor.Actors.*;
import akka.actor.ActorRef;

ActorRef actor = actorOf(Consumer1.class); // create Consumer actor
actor.start();                             // activate endpoint in background
// ...
actor.stop();                              // deactivate endpoint in background
```

Typed actors

When a typed actor is stopped, routes to @consume annotated methods of this typed actors are stopped as well. Stopping the routes is done asynchronously; it may be still in progress after the TypedActor.stop method returned.

Scala

```
import akka.actor.TypedActor

// create TypedConsumer1 object and activate endpoint(s) in background
val consumer = TypedActor.newInstance(classOf[TypedConsumer1], classOf[TypedConumer1Impl])

// deactivate endpoints in background
TypedActor.stop(consumer)
```

Java

```
import akka.actor.TypedActor;

// Create typed consumer actor and activate endpoints in background
TypedConsumer1 consumer = TypedActor.newInstance(TypedConsumer1.class, TypedConumer1Impl.class);

// Deactivate endpoints in background
TypedActor.stop(consumer);
```

Acknowledgements

Actors (untyped)

With in-out message exchanges, clients usually know that a message exchange is done when they receive a reply from a consumer actor. The reply message can be a Message (or any object which is then internally converted to a Message) on success, and a Failure message on failure.

With in-only message exchanges, by default, an exchange is done when a message is added to the consumer actor's mailbox. Any failure or exception that occurs during processing of that message by the consumer actor cannot be reported back to the endpoint in this case. To allow consumer actors to positively or negatively acknowledge the receipt of a message from an in-only message exchange, they need to override the `autoack` (Scala) or `isAutoack` (Java) method to return false. In this case, consumer actors must reply either with a special Ack message (positive acknowledgement) or a Failure (negative acknowledgement).

Scala

```
import akka.camel.{Ack, Failure}
// ... other imports omitted

class Consumer3 extends Actor with Consumer {
  override def autoack = false

  def endpointUri = "jms:queue:test"

  def receive = {
    // ...
    self.reply(Ack) // on success
    // ...
    self.reply(Failure(...)) // on failure
  }
}
```

Java

```
import akka.camel.Failure
import static akka.camel.Ack.ack;
// ... other imports omitted

public class Consumer3 extends UntypedConsumerActor {

  public String getEndpointUri() {
    return "jms:queue:test";
  }

  public boolean isAutoack() {
    return false;
  }

  public void onReceive(Object message) {
    // ...
    getContext().replyUnsafe(ack()) // on success
    // ...
```

```
    val e: Exception = ...
    getContext().replyUnsafe(new Failure(e)) // on failure
  }
}
```

Blocking exchanges

By default, message exchanges between a Camel endpoint and a consumer actor are non-blocking because, internally, the ! (bang) operator is used to commicate with the actor. The route to the actor does not block waiting for a reply. The reply is sent asynchronously (see also *Asynchronous routing*). Consumer actors however can be configured to make this interaction blocking.

Scala

```
class ExampleConsumer extends Actor with Consumer {
  override def blocking = true

  def endpointUri = ...
  def receive = {
    // ...
  }
}
```

Java

```
public class ExampleConsumer extends UntypedConsumerActor {

  public boolean isBlocking() {
    return true;
  }

  public String getEndpointUri() {
    // ...
  }

  public void onReceive(Object message) {
    // ...
  }
}
```

In this case, the !! (bangbang) operator is used internally to communicate with the actor which blocks a thread until the consumer sends a response or throws an exception within receive. Although it may decrease scalability, this setting can simplify error handling (see this article) or allows timeout configurations on actor-level (see *Consumer timeout*).

Consumer timeout

Endpoints that support two-way communications need to wait for a response from an (untyped) actor or typed actor before returning it to the initiating client. For some endpoint types, timeout values can be defined in an endpoint-specific way which is described in the documentation of the individual Camel components. Another option is to configure timeouts on the level of consumer actors and typed consumer actors.

Typed actors

For typed actors, timeout values for method calls that return a result can be set when the typed actor is created. In the following example, the timeout is set to 20 seconds (default is 5 seconds).

Scala

```
import akka.actor.TypedActor

val consumer = TypedActor.newInstance(classOf[TypedConsumer1], classOf[TypedConumer1Impl], 20000
```

Java

```
import akka.actor.TypedActor;

TypedConsumer1 consumer = TypedActor.newInstance(TypedConsumer1.class, TypedConumer1Impl.class, 2
```

Actors (untyped)

Two-way communications between a Camel endpoint and an (untyped) actor are initiated by sending the request message to the actor with the ! (bang) operator and the actor replies to the endpoint when the response is ready. In order to support timeouts on actor-level, endpoints need to send the request message with the !! (bangbang) operator for which a timeout value is applicable. This can be achieved by overriding the Consumer.blocking method to return true.

Scala

```
class Consumer2 extends Actor with Consumer {
  self.timeout = 20000 // timeout set to 20 seconds

  override def blocking = true

  def endpointUri = "direct:example"

  def receive = {
    // ...
  }
}
```

Java

```
public class Consumer2 extends UntypedConsumerActor {

  public Consumer2() {
    getContext().setTimeout(20000); // timeout set to 20 seconds
  }

  public String getEndpointUri() {
    return "direct:example";
  }

  public boolean isBlocking() {
    return true;
  }

  public void onReceive(Object message) {
    // ...
  }
}
```

This is a valid approach for all endpoint types that do not "natively" support asynchronous two-way message exchanges. For all other endpoint types (like Jetty endpoints) is it not recommended to switch to blocking mode but rather to configure timeouts in an endpoint-specific way (see also *Asynchronous routing*).

Remote consumers

Actors (untyped)

Publishing of remote consumer actors is always done on the server side, local proxies are never published. Hence the CamelService must be started on the remote node. For example, to publish an (untyped) actor on a remote node at endpoint URI `jetty:http://localhost:6644/remote-actor-1`, define the following consumer actor class.

Scala

```
import akka.actor.Actor
import akka.annotation.consume
import akka.camel.Consumer

class RemoteActor1 extends Actor with Consumer {
  def endpointUri = "jetty:http://localhost:6644/remote-actor-1"

  protected def receive = {
    case msg => self.reply("response from remote actor 1")
  }
}
```

Java

```
import akka.camel.UntypedConsumerActor;

public class RemoteActor1 extends UntypedConsumerActor {
  public String getEndpointUri() {
    return "jetty:http://localhost:6644/remote-actor-1";
  }

  public void onReceive(Object message) {
    getContext().replySafe("response from remote actor 1");
  }
}
```

On the remote node, start a CamelService, start a remote server, create the actor and register it at the remote server.

Scala

```
import akka.camel.CamelServiceManager._
import akka.actor.Actor._
import akka.actor.ActorRef

// ...
startCamelService

val consumer = val consumer = actorOf[RemoteActor1]

remote.start("localhost", 7777)
remote.register(consumer) // register and start remote consumer
// ...
```

Java

```
import akka.camel.CamelServiceManager;
import static akka.actor.Actors.*;

// ...
CamelServiceManager.startCamelService();

ActorRef actor = actorOf(RemoteActor1.class);
```

```
remote().start("localhost", 7777);
remote().register(actor); // register and start remote consumer
// ...
```

Explicitly starting a CamelService can be omitted when Akka is running in Kernel mode, for example (see also *CamelService configuration*).

Typed actors

Remote typed consumer actors can be registered with one of the `registerTyped*` methods on the remote server. The following example registers the actor with the custom id "123".

Scala

```
import akka.actor.TypedActor

// ...
val obj = TypedActor.newRemoteInstance(
  classOf[SampleRemoteTypedConsumer],
  classOf[SampleRemoteTypedConsumerImpl])

remote.registerTypedActor("123", obj)
// ...
```

Java

```
import akka.actor.TypedActor;

SampleRemoteTypedConsumer obj = (SampleRemoteTypedConsumer)TypedActor.newInstance(
  SampleRemoteTypedConsumer.class,
  SampleRemoteTypedConsumerImpl.class);

remote.registerTypedActor("123", obj)
// ...
```

1.2.4 Produce messages

A minimum pre-requisite for producing messages to Camel endpoints with producer actors (see below) is an initialized and started CamelContextManager.

Scala

```
import akka.camel.CamelContextManager

CamelContextManager.init  // optionally takes a CamelContext as argument
CamelContextManager.start // starts the managed CamelContext
```

Java

```
import akka.camel.CamelContextManager;

CamelContextManager.init();  // optionally takes a CamelContext as argument
CamelContextManager.start(); // starts the managed CamelContext
```

For using producer actors, application may also start a CamelService. This will not only setup a CamelContextManager behind the scenes but also register listeners at the actor registry (needed to publish consumer actors). If your application uses producer actors only and you don't want to have the (very small) overhead generated by the registry listeners then setting up a CamelContextManager without starting CamelService is recommended. Otherwise, just start a CamelService as described for consumer actors: *Consumers and the CamelService*.

Producer trait

Actors (untyped)

For sending messages to Camel endpoints, actors

- written in Scala need to mixin the Producer trait and implement the endpointUri method.
- written in Java need to extend the abstract UntypedProducerActor class and implement the getEndpointUri() method. By extending the UntypedProducerActor class, untyped actors (Java) inherit the behaviour of the Producer trait.

Scala

```
import akka.actor.Actor
import akka.camel.Producer

class Producer1 extends Actor with Producer {
  def endpointUri = "http://localhost:8080/news"
}
```

Java

```
import akka.camel.UntypedProducerActor;

public class Producer1 extends UntypedProducerActor {
  public String getEndpointUri() {
    return "http://localhost:8080/news";
  }
}
```

Producer1 inherits a default implementation of the receive method from the Producer trait. To customize a producer actor's default behavior it is recommended to override the Producer.receiveBeforeProduce and Producer.receiveAfterProduce methods. This is explained later in more detail. Actors should not override the default Producer.receive method.

Any message sent to a Producer actor (or UntypedProducerActor) will be sent to the associated Camel endpoint, in the above example to `http://localhost:8080/news`. Response messages (if supported by the configured endpoint) will, by default, be returned to the original sender. The following example uses the `!!` operator (Scala) to send a message to a Producer actor and waits for a response. In Java, the sendRequestReply method is used.

Scala

```
import akka.actor.Actor._
import akka.actor.ActorRef

val producer = actorOf[Producer1].start
val response = producer !! "akka rocks"
val body = response.bodyAs[String]
```

Java

```
import akka.actor.ActorRef;
import static akka.actor.Actors.*;
import akka.camel.Message;

ActorRef producer = actorOf(Producer1.class).start();
Message response = (Message)producer.sendRequestReply("akka rocks");
String body = response.getBodyAs(String.class)
```

If the message is sent using the ! operator (or the sendOneWay method in Java) then the response message is sent back asynchronously to the original sender. In the following example, a Sender actor sends a message (a String) to a producer actor using the ! operator and asynchronously receives a response (of type Message).

Scala

```
import akka.actor.{Actor, ActorRef}
import akka.camel.Message

class Sender(producer: ActorRef) extends Actor {
  def receive = {
    case request: String   => producer ! request
    case response: Message => {
      /* process response ... */
    }
    // ...
  }
}
```

Java

```
// TODO
```

Custom Processing

Instead of replying to the initial sender, producer actors can implement custom reponse processing by overriding the receiveAfterProduce method (Scala) or onReceiveAfterProduce method (Java). In the following example, the reponse message is forwarded to a target actor instead of being replied to the original sender.

Scala

```
import akka.actor.{Actor, ActorRef}
import akka.camel.Producer

class Producer1(target: ActorRef) extends Actor with Producer {
  def endpointUri = "http://localhost:8080/news"

  override protected def receiveAfterProduce = {
    // do not reply but forward result to target
    case msg => target forward msg
  }
}
```

Java

```
import akka.actor.ActorRef;
import akka.camel.UntypedProducerActor;

public class Producer1 extends UntypedProducerActor {
    private ActorRef target;

    public Producer1(ActorRef target) {
        this.target = target;
    }

    public String getEndpointUri() {
        return "http://localhost:8080/news";
    }

    @Override
    public void onReceiveAfterProduce(Object message) {
        target.forward((Message)message, getContext());
    }
}
```

To create an untyped actor instance with a constructor argument, a factory is needed (this should be doable without a factory in upcoming Akka versions).

```
import akka.actor.ActorRef;
import akka.actor.UntypedActorFactory;
import akka.actor.UntypedActor;

public class Producer1Factory implements UntypedActorFactory {

    private ActorRef target;

    public Producer1Factory(ActorRef target) {
        this.target = target;
    }

    public UntypedActor create() {
        return new Producer1(target);
    }
}
```

The instanitation is done with the Actors.actorOf method and the factory as argument.

```
import static akka.actor.Actors.*;
import akka.actor.ActorRef;

ActorRef target = ...
ActorRef producer = actorOf(new Producer1Factory(target));
producer.start();
```

Before producing messages to endpoints, producer actors can pre-process them by overriding the receiveBeforeProduce method (Scala) or onReceiveBeforeProduce method (Java).

Scala

```
import akka.actor.{Actor, ActorRef}
import akka.camel.{Message, Producer}

class Producer1(target: ActorRef) extends Actor with Producer {
  def endpointUri = "http://localhost:8080/news"

  override protected def receiveBeforeProduce = {
    case msg: Message => {
      // do some pre-processing (e.g. add endpoint-specific message headers)
      // ...

      // and return the modified message
      msg
    }
  }
}
```

Java

```
import akka.actor.ActorRef;
import akka.camel.Message
import akka.camel.UntypedProducerActor;

public class Producer1 extends UntypedProducerActor {
    private ActorRef target;

    public Producer1(ActorRef target) {
        this.target = target;
    }

    public String getEndpointUri() {
        return "http://localhost:8080/news";
    }
```

```
    @Override
    public Object onReceiveBeforeProduce(Object message) {
        Message msg = (Message)message;
        // do some pre-processing (e.g. add endpoint-specific message headers)
        // ...

        // and return the modified message
        return msg
    }
}
```

Producer configuration options

The interaction of producer actors with Camel endpoints can be configured to be one-way or two-way (by initiating in-only or in-out message exchanges, respectively). By default, the producer initiates an in-out message exchange with the endpoint. For initiating an in-only exchange, producer actors

- written in Scala either have to override the oneway method to return true
- written in Java have to override the isOneway method to return true.

Scala

```
import akka.camel.Producer

class Producer2 extends Actor with Producer {
  def endpointUri = "jms:queue:test"
  override def oneway = true
}
```

Java

```
import akka.camel.UntypedProducerActor;

public class SampleUntypedReplyingProducer extends UntypedProducerActor {
    public String getEndpointUri() {
        return "jms:queue:test";
    }

    @Override
    public boolean isOneway() {
        return true;
    }
}
```

Message correlation

To correlate request with response messages, applications can set the Message.MessageExchangeId message header.

Scala

```
import akka.camel.Message

producer ! Message("bar", Map(Message.MessageExchangeId -> "123"))
```

Java

```
// TODO
```

Responses of type Message or Failure will contain that header as well. When receiving messages from Camel endpoints this message header is already set (see *Consume messages*).

Matching responses

The following code snippet shows how to best match responses when sending messages with the !! operator (Scala) or with the sendRequestReply method (Java).

Scala

```
val response = producer !! message

response match {
  case Some(Message(body, headers)) => ...
  case Some(Failure(exception, headers)) => ...
  case _ => ...
}
```

Java

```
// TODO
```

ProducerTemplate

The Producer trait (and the abstract UntypedProducerActor class) is a very convenient way for actors to produce messages to Camel endpoints. (Untyped) actors and typed actors may also use a Camel ProducerTemplate for producing messages to endpoints. For typed actors it's the only way to produce messages to Camel endpoints.

At the moment, only the Producer trait fully supports asynchronous in-out message exchanges with Camel endpoints without allocating a thread for the full duration of the exchange. For example, when using endpoints that support asynchronous message exchanges (such as Jetty endpoints that internally use Jetty's asynchronous HTTP client) then usage of the Producer trait is highly recommended (see also *Asynchronous routing*).

Actors (untyped)

A managed ProducerTemplate instance can be obtained via CamelContextManager.mandatoryTemplate. In the following example, an actor uses a ProducerTemplate to send a one-way message to a `direct:news` endpoint.

Scala

```
import akka.actor.Actor
import akka.camel.CamelContextManager

class ProducerActor extends Actor {
  protected def receive = {
    // one-way message exchange with direct:news endpoint
    case msg => CamelContextManager.mandatoryTemplate.sendBody("direct:news", msg)
  }
}
```

Java

```
import akka.actor.UntypedActor;
import akka.camel.CamelContextManager;

public class SampleUntypedActor extends UntypedActor {
    public void onReceive(Object msg) {
        CamelContextManager.getMandatoryTemplate().sendBody("direct:news", msg);
    }
}
```

Alternatively, one can also use `Option[ProducerTemplate]` returned by `CamelContextManager.template`.

Scala

```
import akka.actor.Actor
import akka.camel.CamelContextManager

class ProducerActor extends Actor {
  protected def receive = {
    // one-way message exchange with direct:news endpoint
    case msg => for(t <- CamelContextManager.template) t.sendBody("direct:news", msg)
  }
}
```

Java

```
import org.apache.camel.ProducerTemplate

import akka.actor.UntypedActor;
import akka.camel.CamelContextManager;

public class SampleUntypedActor extends UntypedActor {
    public void onReceive(Object msg) {
        for (ProducerTemplate t : CamelContextManager.getTemplate()) {
            t.sendBody("direct:news", msg);
        }
    }
}
```

For initiating a a two-way message exchange, one of the `ProducerTemplate.request*` methods must be used.

Scala

```
import akka.actor.Actor
import akka.camel.CamelContextManager

class ProducerActor extends Actor {
  protected def receive = {
    // two-way message exchange with direct:news endpoint
    case msg => self.reply(CamelContextManager.mandatoryTemplate.requestBody("direct:news", msg))
  }
}
```

Java

```
import akka.actor.UntypedActor;
import akka.camel.CamelContextManager;

public class SampleUntypedActor extends UntypedActor {
    public void onReceive(Object msg) {
        getContext().replySafe(CamelContextManager.getMandatoryTemplate().requestBody("direct:news
    }
}
```

Typed actors

Typed Actors get access to a managed ProducerTemplate in the same way, as shown in the next example.

Scala

```
// TODO
```

Java

```
import akka.actor.TypedActor;
import akka.camel.CamelContextManager;
```

```
public class SampleProducerImpl extends TypedActor implements SampleProducer {
    public void foo(String msg) {
        ProducerTemplate template = CamelContextManager.getMandatoryTemplate();
        template.sendBody("direct:news", msg);
    }
}
```

1.2.5 Asynchronous routing

Since Akka 0.10, in-out message exchanges between endpoints and actors are designed to be asynchronous. This is the case for both, consumer and producer actors.

- A consumer endpoint sends request messages to its consumer actor using the `!` (bang) operator and the actor returns responses with self.reply once they are ready. The sender reference used for reply is an adapter to Camel's asynchronous routing engine that implements the ActorRef trait.
- A producer actor sends request messages to its endpoint using Camel's asynchronous routing engine. Asynchronous responses are wrapped and added to the producer actor's mailbox for later processing. By default, response messages are returned to the initial sender but this can be overridden by Producer implementations (see also description of the `receiveAfterProcessing` method in *Custom Processing*).

However, asynchronous two-way message exchanges, without allocating a thread for the full duration of exchange, cannot be generically supported by Camel's asynchronous routing engine alone. This must be supported by the individual Camel components (from which endpoints are created) as well. They must be able to suspend any work started for request processing (thereby freeing threads to do other work) and resume processing when the response is ready. This is currently the case for a subset of components such as the Jetty component. All other Camel components can still be used, of course, but they will cause allocation of a thread for the duration of an in-out message exchange. There's also a *Asynchronous routing and transformation example* that implements both, an asynchronous consumer and an asynchronous producer, with the jetty component.

1.2.6 Fault tolerance

Consumer actors and typed actors can be also managed by supervisors. If a consumer is configured to be restarted upon failure the associated Camel endpoint is not restarted. It's behaviour during restart is as follows.

- A one-way (in-only) message exchange will be queued by the consumer and processed once restart completes.
- A two-way (in-out) message exchange will wait and either succeed after restart completes or time-out when the restart duration exceeds the *Consumer timeout*.

If a consumer is configured to be shut down upon failure, the associated endpoint is shut down as well. For details refer to *Consumer un-publishing*.

For examples, tips and trick how to implement fault-tolerant consumer and producer actors, take a look at these two articles.

- Akka Consumer Actors: New Features and Best Practices
- Akka Producer Actors: New Features and Best Practices

1.2.7 CamelService configuration

For publishing consumer actors and typed actor methods (*Consumer publishing*), applications must start a CamelService. When starting Akka in *Microkernel* mode then a CamelService can be started automatically when camel is added to the enabled-modules list in akka.conf, for example:

```
akka {
  ...
  enabled-modules = ["camel"] # Options: ["remote", "camel", "http"]
  ...
}
```

Applications that do not use the Akka Kernel, such as standalone applications for example, need to start a CamelService manually, as explained in the following subsections.When starting a CamelService manually, settings in akka.conf are ignored.

Standalone applications

Standalone application should create and start a CamelService in the following way.

Scala

```
import akka.camel.CamelServiceManager._

startCamelService
```

Java

```
import static akka.camel.CamelServiceManager.*;

startCamelService();
```

Internally, a CamelService uses the CamelContextManager singleton to manage a CamelContext. A CamelContext manages the routes from endpoints to consumer actors and typed actors. These routes are added and removed at runtime (when (untyped) consumer actors and typed consumer actors are started and stopped). Applications may additionally want to add their own custom routes or modify the CamelContext in some other way. This can be done by initializing the CamelContextManager manually and making modifications to CamelContext **before** the CamelService is started.

Scala

```
import org.apache.camel.builder.RouteBuilder

import akka.camel.CamelContextManager
import akka.camel.CamelServiceManager._

CamelContextManager.init

// add a custom route to the managed CamelContext
CamelContextManager.mandatoryContext.addRoutes(new CustomRouteBuilder)

startCamelService

// an application-specific route builder
class CustomRouteBuilder extends RouteBuilder {
  def configure {
    // ...
  }
}
```

Java

```
import org.apache.camel.builder.RouteBuilder;

import akka.camel.CamelContextManager;
import static akka.camel.CamelServiceManager.*;

CamelContextManager.init();
```

```
// add a custom route to the managed CamelContext
CamelContextManager.getMandatoryContext().addRoutes(new CustomRouteBuilder());

startCamelService();

// an application-specific route builder
private static class CustomRouteBuilder extends RouteBuilder {
    public void configure() {
        // ...
    }
}
```

Applications may even provide their own CamelContext instance as argument to the init method call as shown in the following snippet. Here, a DefaultCamelContext is created using a Spring application context as registry.

Scala

```
import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.spring.spi.ApplicationContextRegistry
import org.springframework.context.support.ClassPathXmlApplicationContext

import akka.camel.CamelContextManager
import akka.camel.CamelServiceManager._

// create a custom Camel registry backed up by a Spring application context
val context = new ClassPathXmlApplicationContext("/context.xml")
val registry = new ApplicationContextRegistry(context)

// initialize CamelContextManager with a DefaultCamelContext using the custom registry
CamelContextManager.init(new DefaultCamelContext(registry))

// ...

startCamelService
```

Java

```
import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.spi.Registry;
import org.apache.camel.spring.spi.ApplicationContextRegistry;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import akka.camel.CamelContextManager;
import static akka.camel.CamelServiceManager.*;

// create a custom Camel registry backed up by a Spring application context
ApplicationContext context = new ClassPathXmlApplicationContext("/context.xml");
Registry registry = new ApplicationContextRegistry(context);

// initialize CamelContextManager with a DefaultCamelContext using the custom registry
CamelContextManager.init(new DefaultCamelContext(registry));

// ...

startCamelService();
```

Standalone Spring applications

A better approach to configure a Spring application context as registry for the CamelContext is to use Camel's Spring support. Furthermore, the *Spring Integration* module additionally supports a <camel-service> element

for creating and starting a CamelService. An optional reference to a custom CamelContext can be defined for <camel-service> as well. Here's an example.

```
<!-- context.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://repo.akka.io/schema/akka"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://repo.akka.io/schema/akka
http://repo.akka.io/akka-1.3.1.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

  <!-- A custom CamelContext (SpringCamelContext) -->
  <camel:camelContext id="camelContext">
    <!-- ... -->
  </camel:camelContext>

  <!-- Create a CamelService using a custom CamelContext -->
  <akka:camel-service>
    <akka:camel-context ref="camelContext" />
  </akka:camel-service>

</beans>
```

Creating a CamelContext this way automatically adds the defining Spring application context as registry to that CamelContext. The CamelService is started when the application context is started and stopped when the application context is closed. A simple usage example is shown in the following snippet.

Scala

```
import org.springframework.context.support.ClassPathXmlApplicationContext
import akka.camel.CamelContextManager

// Create and start application context (start CamelService)
val appctx = new ClassPathXmlApplicationContext("/context.xml")

// Access to CamelContext (SpringCamelContext)
val ctx = CamelContextManager.mandatoryContext
// Access to ProducerTemplate of that CamelContext
val tpl = CamelContextManager.mandatoryTemplate

// use ctx and tpl ...

// Close application context (stop CamelService)
appctx.close
```

Java

```
// TODO
```

If the CamelService doesn't reference a custom CamelContext then a DefaultCamelContext is created (and accessible via the CamelContextManager).

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://repo.akka.io/schema/akka"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
```

```
http://repo.akka.io/schema/akka
http://repo.akka.io/akka-1.3.1.xsd">

  <!-- Create a CamelService using DefaultCamelContext -->
  <akka:camel-service />

</beans>
```

Kernel mode

For classes that are loaded by the Kernel or the Initializer, starting the CamelService can be omitted, as discussed in the previous section. Since these classes are loaded and instantiated before the CamelService is started (by Akka), applications can make modifications to a CamelContext here as well (and even provide their own CamelContext). Assuming there's a boot class sample.camel.Boot configured in akka.conf.

```
akka {
  ...
  boot = ["sample.camel.Boot"]
  ...
}
```

Modifications to the CamelContext can be done like in the following snippet.

Scala

```
package sample.camel

import org.apache.camel.builder.RouteBuilder

import akka.camel.CamelContextManager

class Boot {
  CamelContextManager.init

  // Customize CamelContext with application-specific routes
  CamelContextManager.mandatoryContext.addRoutes(new CustomRouteBuilder)

  // No need to start CamelService here. It will be started
  // when this classes has been loaded and instantiated.
}

class CustomRouteBuilder extends RouteBuilder {
  def configure {
    // ...
  }
}
```

Java

```
// TODO
```

1.2.8 Custom Camel routes

In all the examples so far, routes to consumer actors have been automatically constructed by akka-camel, when the actor was started. Although the default route construction templates, used by akka-camel internally, are sufficient for most use cases, some applications may require more specialized routes to actors. The akka-camel module provides two mechanisms for customizing routes to actors, which will be explained in this section. These are

- Usage of *Akka Camel components* to access (untyped) actor and actors. Any Camel route can use these components to access Akka actors.

- *Intercepting route construction* to (untyped) actor and actors. Default routes to consumer actors are extended using predefined extension points.

Akka Camel components

Akka actors can be access from Camel routes using the actor and typed-actor Camel components, respectively. These components can be used to access any Akka actor (not only consumer actors) from Camel routes, as described in the following sections.

Access to actors

To access (untyped) actors from custom Camel routes, the actor Camel component should be used. It fully supports Camel's asynchronous routing engine.

This component accepts the following enpoint URI formats:

- `actor:<actor-id>[?<options>]`
- `actor:id:[<actor-id>][?<options>]`
- `actor:uuid:[<actor-uuid>][?<options>]`

where `<actor-id>` and `<actor-uuid>` refer to `actorRef.id` and the String-representation of `actorRef.uuid`, respectively. The `<options>` are name-value pairs separated by `&` (i.e. `name1=value1&name2=value2&...`).

URI options

The following URI options are supported:

Name	Type	Default	Description
blocking	Boolean	false	If set to true, in-out message exchanges with the target actor will be made with the `!!` operator, otherwise with the `!` operator. See also *Consumer timeout*.
autoack	Boolean	true	If set to true, in-only message exchanges are auto-acknowledged when the message is added to the actor's mailbox. If set to false, actors must acknowledge the receipt of the message. See also *Acknowledgements*.

Here's an actor endpoint URI example containing an actor uuid:

```
actor:uuid:12345678?blocking=true
```

In actor endpoint URIs that contain id: or uuid:, an actor identifier (id or uuid) is optional. In this case, the in-message of an exchange produced to an actor endpoint must contain a message header with name CamelActorIdentifier (which is defined by the ActorComponent.ActorIdentifier field) and a value that is the target actor's identifier. On the other hand, if the URI contains an actor identifier, it can be seen as a default actor identifier that can be overridden by messages containing a CamelActorIdentifier header.

Message headers

Name	Type	Description
CamelActorIdentifier	String	Contains the identifier (id or uuid) of the actor to route the message to. The identifier is interpreted as actor id if the URI contains id:, the identifier is interpreted as uuid id the URI contains uuid:. A uuid value may also be of type Uuid (not only String). The header name is defined by the ActorComponent.ActorIdentifier field.

Here's another actor endpoint URI example that doesn't define an actor uuid. In this case the target actor uuid must be defined by the CamelActorIdentifier message header:

```
actor:uuid:
```

In the following example, a custom route to an actor is created, using the actor's uuid (i.e. actorRef.uuid). The route starts from a Jetty endpoint and ends at the target actor.

Scala

```
import org.apache.camel.builder.RouteBuilder

import akka.actor._
import akka.actor.Actor
import akka.actor.Actor._
import akka.camel.{Message, CamelContextManager, CamelServiceManager}

object CustomRouteExample extends Application {
  val target = actorOf[CustomRouteTarget].start

  CamelServiceManager.startCamelService
  CamelContextManager.mandatoryContext.addRoutes(new CustomRouteBuilder(target.uuid))
}

class CustomRouteTarget extends Actor {
  def receive = {
    case msg: Message => self.reply("Hello %s" format msg.bodyAs[String])
  }
}

class CustomRouteBuilder(uuid: Uuid) extends RouteBuilder {
  def configure {
    val actorUri = "actor:uuid:%s" format uuid
    from("jetty:http://localhost:8877/camel/custom").to(actorUri)
  }
}
```

Java

```
import com.eaio.uuid.UUID;

import org.apache.camel.builder.RouteBuilder;
import static akka.actor.Actors.*;
import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.camel.CamelServiceManager;
import akka.camel.CamelContextManager;
import akka.camel.Message;

public class CustomRouteExample {
    public static void main(String... args) throws Exception {
        ActorRef target = actorOf(CustomRouteTarget.class).start();
        CamelServiceManager.startCamelService();
        CamelContextManager.getMandatoryContext().addRoutes(new CustomRouteBuilder(target.getUuid
    }
}

public class CustomRouteTarget extends UntypedActor {
    public void onReceive(Object message) {
        Message msg = (Message) message;
        String body = msg.getBodyAs(String.class);
        getContext().replySafe(String.format("Hello %s", body));
    }
}
```

```
public class CustomRouteBuilder extends RouteBuilder {
    private UUID uuid;

    public CustomRouteBuilder(UUID uuid) {
        this.uuid = uuid;
    }

    public void configure() {
        String actorUri = String.format("actor:uuid:%s", uuid);
        from("jetty:http://localhost:8877/camel/custom").to(actorUri);
    }
}
```

When the example is started, messages POSTed to `http://localhost:8877/camel/custom` are routed to the target actor.

Access to typed actors

To access typed actor methods from custom Camel routes, the typed-actor Camel component should be used. It is a specialization of the Camel bean component. Applications should use the interface (endpoint URI syntax and options) as described in the bean component documentation but with the typed-actor schema. Typed Actors must be added to a Camel registry for being accessible by the typed-actor component.

Using Spring

The following example shows how to access typed actors in a Spring application context. For adding typed actors to the application context and for starting *Standalone Spring applications* the *Spring Integration* module is used in the following example. It offers a `<typed-actor>` element to define typed actor factory beans and a `<camel-service>` element to create and start a CamelService.

```
<!--
  context.xml
-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://repo.akka.io/schema/akka"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://repo.akka.io/schema/akka
http://repo.akka.io/akka-1.3.1.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean id="routeBuilder" class="sample.SampleRouteBuilder" />

  <camel:camelContext id="camelContext">
    <camel:routeBuilder ref="routeBuilder" />
  </camel:camelContext>

  <akka:camel-service>
    <akka:camel-context ref="camelContext" />
  </akka:camel-service>

  <akka:typed-actor id="sample"
                    interface="sample.SampleTypedActor"
                    implementation="sample.SampleTypedActorImpl"
                    timeout="1000" />
</beans>
```

SampleTypedActor is the typed actor interface and SampleTypedActorImpl in the typed actor implementation class.

Scala

```
package sample

import akka.actor.TypedActor

trait SampleTypedActor {
  def foo(s: String): String
}

class SampleTypedActorImpl extends TypedActor with SampleTypedActor {
  def foo(s: String) = "hello %s" format s
}
```

Java

```
package sample;

import akka.actor.TypedActor;

public interface SampleTypedActor {
    public String foo(String s);
}

public class SampleTypedActorImpl extends TypedActor implements SampleTypedActor {

    public String foo(String s) {
        return "hello " + s;
    }
}
```

The SampleRouteBuilder defines a custom route from the direct:test endpoint to the sample typed actor using a typed-actor endpoint URI.

Scala

```
package sample

import org.apache.camel.builder.RouteBuilder

class SampleRouteBuilder extends RouteBuilder {
  def configure = {
    // route to typed actor
    from("direct:test").to("typed-actor:sample?method=foo")
  }
}
```

Java

```
package sample;

import org.apache.camel.builder.RouteBuilder;

public class SampleRouteBuilder extends RouteBuilder {
    public void configure() {
        // route to typed actor
        from("direct:test").to("typed-actor:sample?method=foo");
    }
}
```

The typed-actor endpoint URI syntax is::

```
typed-actor:<bean-id>?method=<method-name>
```

where `<bean-id>` is the id of the bean in the Spring application context and `<method-name>` is the name of the typed actor method to invoke.

Usage of the custom route for sending a message to the typed actor is shown in the following snippet.

Scala

```
package sample

import org.springframework.context.support.ClassPathXmlApplicationContext
import akka.camel.CamelContextManager

// load Spring application context (starts CamelService)
val appctx = new ClassPathXmlApplicationContext("/context-standalone.xml")

// access 'sample' typed actor via custom route
assert("hello akka" == CamelContextManager.mandatoryTemplate.requestBody("direct:test", "akka"))

// close Spring application context (stops CamelService)
appctx.close
```

Java

```
package sample;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import akka.camel.CamelContextManager;

// load Spring application context
ClassPathXmlApplicationContext appctx = new ClassPathXmlApplicationContext("/context-standalone.xml

// access 'externally' registered typed actors with typed-actor component
assert("hello akka" == CamelContextManager.getMandatoryTemplate().requestBody("direct:test", "akka

// close Spring application context (stops CamelService)
appctx.close();
```

The application uses a Camel producer template to access the typed actor via the `direct:test` endpoint.

Without Spring

Usage of *Spring Integration* for adding typed actors to the Camel registry and starting a CamelService is optional. Setting up a Spring-less application for accessing typed actors is shown in the next example.

Scala

```
package sample

import org.apache.camel.impl.{DefaultCamelContext, SimpleRegistry}
import akka.actor.TypedActor
import akka.camel.CamelContextManager
import akka.camel.CamelServiceManager._

// register typed actor
val registry = new SimpleRegistry
registry.put("sample", TypedActor.newInstance(classOf[SampleTypedActor], classOf[SampleTypedActor

// customize CamelContext
CamelContextManager.init(new DefaultCamelContext(registry))
CamelContextManager.mandatoryContext.addRoutes(new SampleRouteBuilder)
```

```
startCamelService

// access 'sample' typed actor via custom route
assert("hello akka" == CamelContextManager.mandatoryTemplate.requestBody("direct:test", "akka"))

stopCamelService
```

Java

```
package sample;

// register typed actor
SimpleRegistry registry = new SimpleRegistry();
registry.put("sample", TypedActor.newInstance(SampleTypedActor.class, SampleTypedActorImpl.class)

// customize CamelContext
CamelContextManager.init(new DefaultCamelContext(registry));
CamelContextManager.getMandatoryContext().addRoutes(new SampleRouteBuilder());

startCamelService();

// access 'sample' typed actor via custom route
assert("hello akka" == CamelContextManager.getMandatoryTemplate().requestBody("direct:test", "akka

stopCamelService();
```

Here, SimpleRegistry, a java.util.Map based registry, is used to register typed actors. The CamelService is started and stopped programmatically.

Intercepting route construction

The previous section, *Akka Camel components*, explained how to setup a route to an (untyped) actor or typed actor manually. It was the application's responsibility to define the route and add it to the current CamelContext. This section explains a more conventient way to define custom routes: akka-camel is still setting up the routes to consumer actors (and adds these routes to the current CamelContext) but applications can define extensions to these routes. Extensions can be defined with Camel's Java DSL or Scala DSL. For example, an extension could be a custom error handler that redelivers messages from an endpoint to an actor's bounded mailbox when the mailbox was full.

The following examples demonstrate how to extend a route to a consumer actor for handling exceptions thrown by that actor. To simplify the example, we configure *Blocking exchanges* which reports any exception, that is thrown by receive, directly back to the Camel route. One could also report exceptions asynchronously using a Failure reply (see also this article) but we'll do it differently here.

Actors (untyped)

Scala

```
import akka.actor.Actor
import akka.camel.Consumer

import org.apache.camel.builder.Builder
import org.apache.camel.model.RouteDefinition

class ErrorHandlingConsumer extends Actor with Consumer {
  def endpointUri = "direct:error-handler-test"

  // Needed to propagate exception back to caller
  override def blocking = true
```

```
  onRouteDefinition {rd: RouteDefinition =>
    // Catch any exception and handle it by returning the exception message as response
    rd.onException(classOf[Exception]).handled(true).transform(Builder.exceptionMessage).end
  }

  protected def receive = {
    case msg: Message => throw new Exception("error: %s" format msg.body)
  }
}
```

Java

```
import akka.camel.UntypedConsumerActor;

import org.apache.camel.builder.Builder;
import org.apache.camel.model.ProcessorDefinition;
import org.apache.camel.model.RouteDefinition;

public class SampleErrorHandlingConsumer extends UntypedConsumerActor {

    public String getEndpointUri() {
        return "direct:error-handler-test";
    }

    // Needed to propagate exception back to caller
    public boolean isBlocking() {
        return true;
    }

    public void preStart() {
        onRouteDefinition(new RouteDefinitionHandler() {
            public ProcessorDefinition<?> onRouteDefinition(RouteDefinition rd) {
                // Catch any exception and handle it by returning the exception message as respons
                return rd.onException(Exception.class).handled(true).transform(Builder.exceptionMe
            }
        });
    }

    public void onReceive(Object message) throws Exception {
        Message msg = (Message)message;
        String body = msg.getBodyAs(String.class);
        throw new Exception(String.format("error: %s", body));
   }

}
```

For (untyped) actors, consumer route extensions are defined by calling the onRouteDefinition method with a route definition handler. In Scala, this is a function of type `RouteDefinition => ProcessorDefinition[_]`, in Java it is an instance of `RouteDefinitionHandler` which is defined as follows.

```
package akka.camel

import org.apache.camel.model.RouteDefinition
import org.apache.camel.model.ProcessorDefinition

trait RouteDefinitionHandler {
  def onRouteDefinition(rd: RouteDefinition): ProcessorDefinition[_]
}
```

The akka-camel module creates a RouteDefinition instance by calling from(endpointUri) on a Camel RouteBuilder (where endpointUri is the endpoint URI of the consumer actor) and passes that instance as argument to the route definition handler *). The route definition handler then extends the route and returns a ProcessorDefinition (in the

above example, the ProcessorDefinition returned by the end method. See the org.apache.camel.model package for details). After executing the route definition handler, akka-camel finally calls a to(actor:uuid:actorUuid) on the returned ProcessorDefinition to complete the route to the comsumer actor (where actorUuid is the uuid of the consumer actor).

*) Before passing the RouteDefinition instance to the route definition handler, akka-camel may make some further modifications to it.

Typed actors

For typed consumer actors to define a route definition handler, they must provide a RouteDefinitionHandler implementation class with the @consume annotation. The implementation class must have a no-arg constructor. Here's an example (in Java).

```
import org.apache.camel.builder.Builder;
import org.apache.camel.model.ProcessorDefinition;
import org.apache.camel.model.RouteDefinition;

public class SampleRouteDefinitionHandler implements RouteDefinitionHandler {
    public ProcessorDefinition<?> onRouteDefinition(RouteDefinition rd) {
        return rd.onException(Exception.class).handled(true).transform(Builder.exceptionMessage()
    }
}
```

It can be used as follows.

Scala

```
trait TestTypedConsumer {
  @consume(value="direct:error-handler-test", routeDefinitionHandler=classOf[SampleRouteDefinitio
  def foo(s: String): String
}

// implementation class omitted
```

Java

```
public interface SampleErrorHandlingTypedConsumer {

    @consume(value="direct:error-handler-test", routeDefinitionHandler=SampleRouteDefinitionHandle
    String foo(String s);

}

// implementation class omitted
```

1.2.9 Examples

For all features described so far, there's running sample code in akka-sample-camel. The examples in sample.camel.Boot are started during Kernel startup because this class has been added to the boot configuration in akka-reference.conf.

```
akka {
  ...
  boot = ["sample.camel.Boot", ...]
  ...
}
```

If you don't want to have these examples started during Kernel startup, delete it from akka-reference.conf (or from akka.conf if you have a custom boot configuration). Other examples are standalone applications (i.e. classes with a main method) that can be started from sbt.

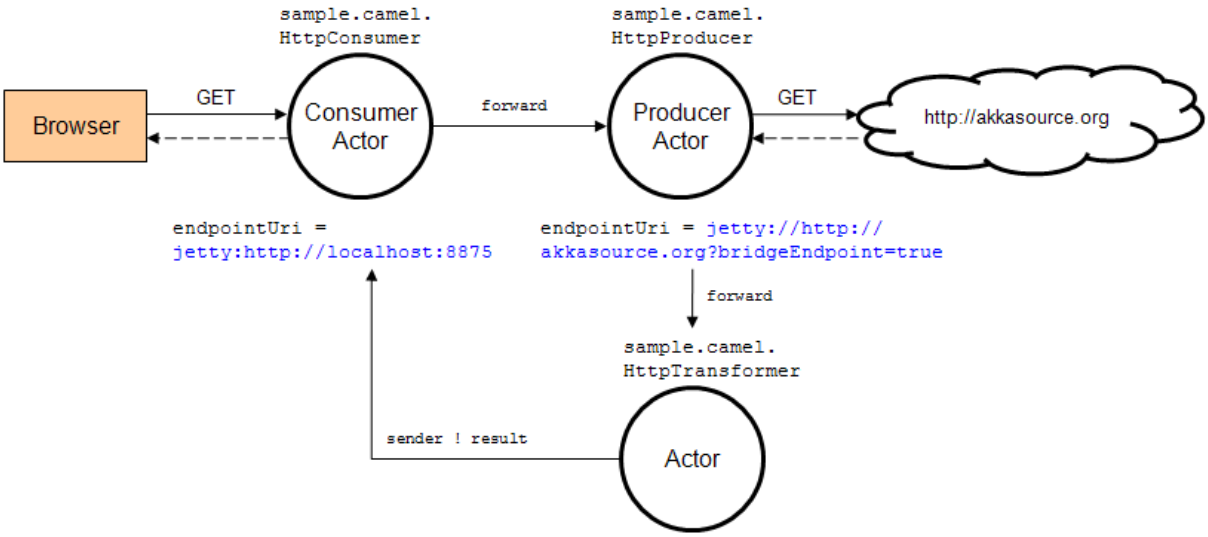
```
$ sbt
[info] Building project akka-modules 1.3.1 against Scala 2.9.0
[info]    using AkkaModulesParentProject with sbt 0.7.6 and Scala 2.7.7
> project akka-sample-camel
Set current project to akka-sample-camel 1.3.1
> run
...
Multiple main classes detected, select one to run:

 [1] sample.camel.ClientApplication
 [2] sample.camel.ServerApplication
 [3] sample.camel.StandaloneSpringApplication
 [4] sample.camel.StandaloneApplication
 [5] sample.camel.StandaloneFileApplication
 [6] sample.camel.StandaloneJmsApplication
```

Some of the examples in akka-sample-camel are described in more detail in the following subsections.

Asynchronous routing and transformation example

This example demonstrates how to implement consumer and producer actors that support *Asynchronous routing* with their Camel endpoints. The sample application transforms the content of the Akka homepage, http://akka.io, by replacing every occurrence of *Akka* with *AKKA*. After starting the *Microkernel*, direct the browser to http://localhost:8875 and the transformed Akka homepage should be displayed. Please note that this example will probably not work if you're behind an HTTP proxy.

The following figure gives an overview how the example actors interact with external systems and with each other. A browser sends a GET request to http://localhost:8875 which is the published endpoint of the `HttpConsumer` actor. The `HttpConsumer` actor forwards the requests to the `HttpProducer` actor which retrieves the Akka homepage from http://akka.io. The retrieved HTML is then forwarded to the `HttpTransformer` actor which replaces all occurences of *Akka* with *AKKA*. The transformation result is sent back the HttpConsumer which finally returns it to the browser.

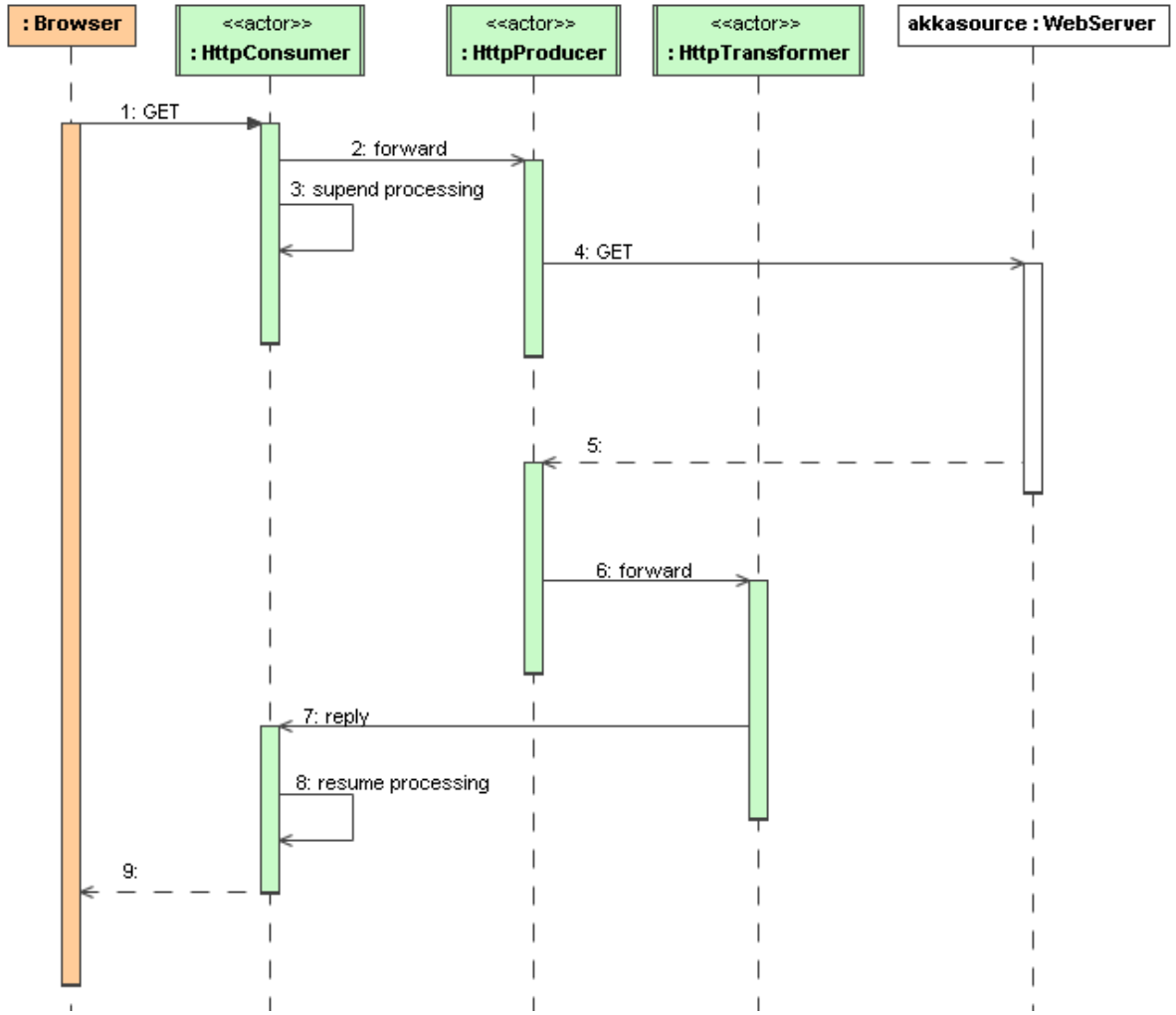


Implementing the example actor classes and wiring them together is rather easy as shown in the following snippet (see also sample.camel.Boot).

```
import org.apache.camel.Exchange
import akka.actor.Actor._
import akka.actor.{Actor, ActorRef}
import akka.camel.{Producer, Message, Consumer}

class HttpConsumer(producer: ActorRef) extends Actor with Consumer {
  def endpointUri = "jetty:http://0.0.0.0:8875/"
```

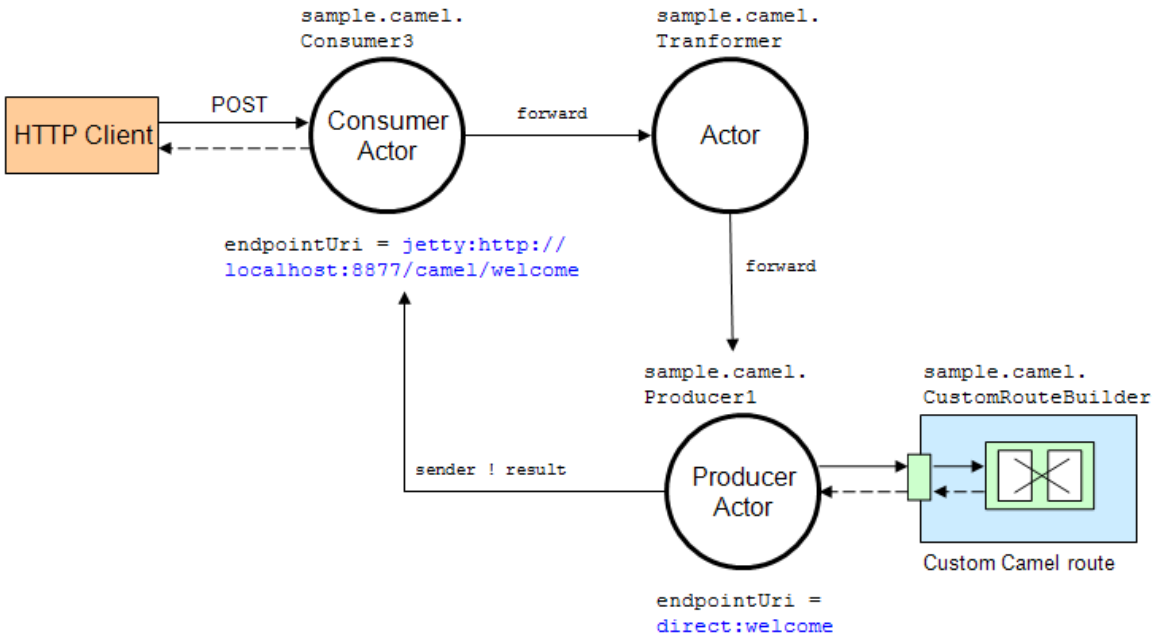
```
  protected def receive = {
    case msg => producer forward msg
  }
}

class HttpProducer(transformer: ActorRef) extends Actor with Producer {
  def endpointUri = "jetty://http://akka.io/?bridgeEndpoint=true"

  override protected def receiveBeforeProduce = {
    // only keep Exchange.HTTP_PATH message header (which needed by bridge endpoint)
    case msg: Message => msg.setHeaders(msg.headers(Set(Exchange.HTTP_PATH)))
  }

  override protected def receiveAfterProduce = {
    // do not reply but forward result to transformer
    case msg => transformer forward msg
  }
}

class HttpTransformer extends Actor {
  protected def receive = {
    case msg: Message => self.reply(msg.transformBody {body: String => body replaceAll ("Akka ",
    case msg: Failure => self.reply(msg)
  }
}

// Wire and start the example actors
val httpTransformer = actorOf(new HttpTransformer).start
val httpProducer = actorOf(new HttpProducer(httpTransformer)).start
val httpConsumer = actorOf(new HttpConsumer(httpProducer)).start
```

The jetty endpoints of HttpConsumer and HttpProducer support asynchronous in-out message exchanges and do not allocate threads for the full duration of the exchange. This is achieved by using Jetty continuations on the consumer-side and by using Jetty's asynchronous HTTP client on the producer side. The following high-level sequence diagram illustrates that.



Custom Camel route example

This section also demonstrates the combined usage of a `Producer` and a `Consumer` actor as well as the inclusion of a custom Camel route. The following figure gives an overview.



- A consumer actor receives a message from an HTTP client
- It forwards the message to another actor that transforms the message (encloses the original message into hyphens)
- The transformer actor forwards the transformed message to a producer actor
- The producer actor sends the message to a custom Camel route beginning at the `direct:welcome` endpoint
- A processor (transformer) in the custom Camel route prepends "Welcome" to the original message and creates a result message
- The producer actor sends the result back to the consumer actor which returns it to the HTTP client

The example is part of sample.camel.Boot. The consumer, transformer and producer actor implementations are as follows.

```
package sample.camel

import akka.actor.{Actor, ActorRef}
import akka.camel.{Message, Consumer}

class Consumer3(transformer: ActorRef) extends Actor with Consumer {
  def endpointUri = "jetty:http://0.0.0.0:8877/camel/welcome"

  def receive = {
    // Forward a string representation of the message body to transformer
    case msg: Message => transformer.forward(msg.setBodyAs[String])
  }
}

class Transformer(producer: ActorRef) extends Actor {
  protected def receive = {
    // example: transform message body "foo" to "- foo -" and forward result to producer
    case msg: Message => producer.forward(msg.transformBody((body: String) => "- %s -" format body
  }
}

class Producer1 extends Actor with Producer {
  def endpointUri = "direct:welcome"
}
```

The producer actor knows where to reply the message to because the consumer and transformer actors have forwarded the original sender reference as well. The application configuration and the route starting from direct:welcome are as follows.

```
package sample.camel

import org.apache.camel.builder.RouteBuilder
import org.apache.camel.{Exchange, Processor}

import akka.actor.Actor._
import akka.camel.CamelContextManager

class Boot {
  CamelContextManager.init()
  CamelContextManager.mandatoryContext.addRoutes(new CustomRouteBuilder)

  val producer = actorOf[Producer1]
  val mediator = actorOf(new Transformer(producer))
  val consumer = actorOf(new Consumer3(mediator))

  producer.start
  mediator.start
  consumer.start
}

class CustomRouteBuilder extends RouteBuilder {
  def configure {
    from("direct:welcome").process(new Processor() {
      def process(exchange: Exchange) {
        // Create a 'welcome' message from the input message
        exchange.getOut.setBody("Welcome %s" format exchange.getIn.getBody)
      }
    })
  }
}
```

To run the example, start the *Microkernel* and POST a message to `http://localhost:8877/camel/welcome`.

```
curl -H "Content-Type: text/plain" -d "Anke" http://localhost:8877/camel/welcome
```

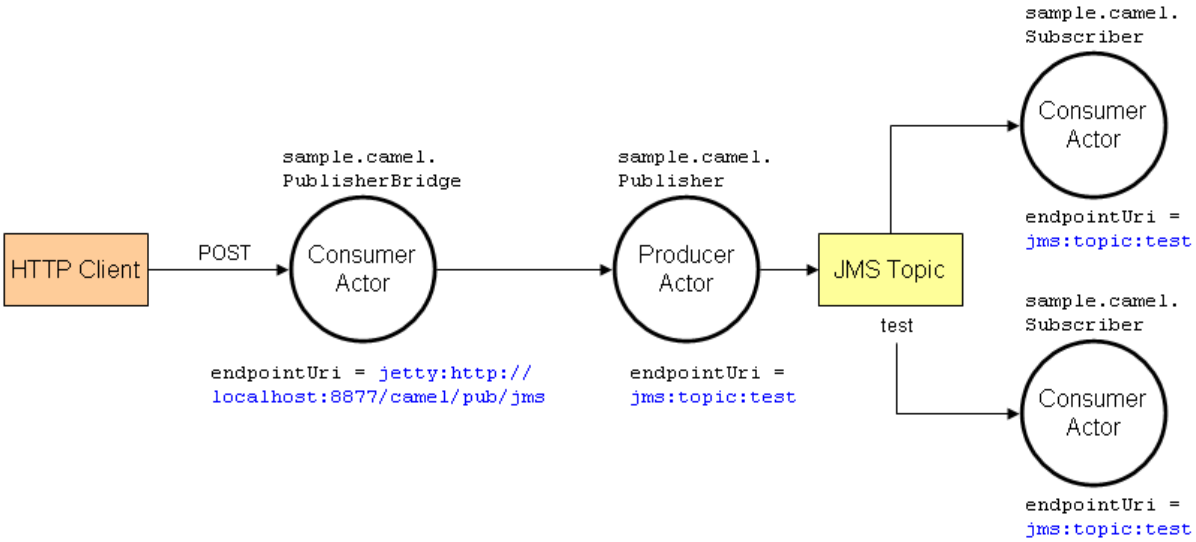
The response should be:

```
Welcome - Anke -
```

Publish-subcribe example

JMS

This section demonstrates how akka-camel can be used to implement publish/subscribe for actors. The following figure sketches an example for JMS-based publish/subscribe.



A consumer actor receives a message from an HTTP client. It sends the message to a JMS producer actor (publisher). The JMS producer actor publishes the message to a JMS topic. Two other actors that subscribed to that topic both receive the message. The actor classes used in this example are shown in the following snippet.

```
package sample.camel

import akka.actor.{Actor, ActorRef}
import akka.camel.{Producer, Message, Consumer}

class Subscriber(name:String, uri: String) extends Actor with Consumer {
  def endpointUri = uri

  protected def receive = {
    case msg: Message => println("%s received: %s" format (name, msg.body))
  }
}

class Publisher(name: String, uri: String) extends Actor with Producer {
  self.id = name

  def endpointUri = uri

  // one-way communication with JMS
  override def oneway = true
}

class PublisherBridge(uri: String, publisher: ActorRef) extends Actor with Consumer {
  def endpointUri = uri

  protected def receive = {
    case msg: Message => {
      publisher ! msg.bodyAs[String]
      self.reply("message published")
    }
  }
}
```

Wiring these actors to implement the above example is as simple as

```
package sample.camel

import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.spring.spi.ApplicationContextRegistry
import org.springframework.context.support.ClassPathXmlApplicationContext
```

```
import akka.actor.Actor._
import akka.camel.CamelContextManager

class Boot {
  // Create CamelContext with Spring-based registry and custom route builder
  val context = new ClassPathXmlApplicationContext("/context-jms.xml", getClass)
  val registry = new ApplicationContextRegistry(context)
  CamelContextManager.init(new DefaultCamelContext(registry))

  // Setup publish/subscribe example
  val jmsUri = "jms:topic:test"
  val jmsSubscriber1 = actorOf(new Subscriber("jms-subscriber-1", jmsUri)).start
  val jmsSubscriber2 = actorOf(new Subscriber("jms-subscriber-2", jmsUri)).start
  val jmsPublisher   = actorOf(new Publisher("jms-publisher", jmsUri)).start

  val jmsPublisherBridge = actorOf(new PublisherBridge("jetty:http://0.0.0.0:8877/camel/pub/jms",
}
```

To publish messages to subscribers one could of course also use the JMS API directly; there's no need to do that over a JMS producer actor as in this example. For the example to work, Camel's jms component needs to be configured with a JMS connection factory which is done in a Spring application context XML file (context-jms.xml).

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <!-- ========================================================================= -->
  <!--  Camel JMS component and ActiveMQ setup                                   -->
  <!-- ========================================================================= -->

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
      <property name="configuration" ref="jmsConfig"/>
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
      <property name="connectionFactory" ref="singleConnectionFactory"/>
  </bean>

  <bean id="singleConnectionFactory" class="org.springframework.jms.connection.SingleConnectionFa
      <property name="targetConnectionFactory" ref="jmsConnectionFactory"/>
  </bean>

  <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://testbroker"/>
  </bean>

</beans>
```

To run the example, start the *Microkernel* and POST a message to `http://localhost:8877/camel/pub/jms`.

```
curl -H "Content-Type: text/plain" -d "Happy hAkking" http://localhost:8877/camel/pub/jms
```

The HTTP response body should be

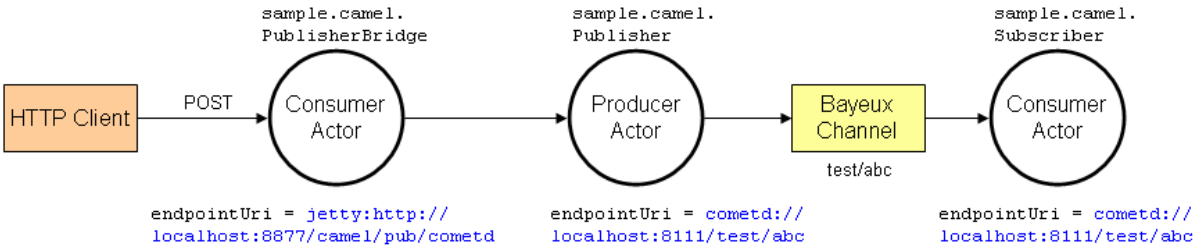
```
message published
```

On the console, where you started the Akka Kernel, you should see something like

```
...
INF [20100622-11:49:57.688] camel: jms-subscriber-2 received: Happy hAkking
INF [20100622-11:49:57.688] camel: jms-subscriber-1 received: Happy hAkking
```

Cometd

Publish/subscribe with CometD is equally easy using Camel's cometd component.



All actor classes from the JMS example can re-used, only the endpoint URIs need to be changed.

```
package sample.camel

import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.spring.spi.ApplicationContextRegistry
import org.springframework.context.support.ClassPathXmlApplicationContext

import akka.actor.Actor._
import akka.camel.CamelContextManager

class Boot {
  // ...

  // Setup publish/subscribe example
  val cometdUri = "cometd://localhost:8111/test/abc?resourceBase=target"
  val cometdSubscriber = actorOf(new Subscriber("cometd-subscriber", cometdUri)).start
  val cometdPublisher  = actorOf(new Publisher("cometd-publisher", cometdUri)).start

  val cometdPublisherBridge = actorOf(new PublisherBridge("jetty:http://0.0.0.0:8877/camel/pub/co
}
```

Quartz Scheduler Example

Here is an example showing how simple is to implement a cron-style scheduler by using the Camel Quartz component in Akka.

The following example creates a "timer" actor which fires a message every 2 seconds:

```
package com.dimingo.akka

import akka.actor.Actor
import akka.actor.Actor.actorOf

import akka.camel.{Consumer, Message}
import akka.camel.CamelServiceManager._

class MyQuartzActor extends Actor with Consumer {

    def endpointUri = "quartz://example?cron=0/2+*+*+*+*+*+?"

    def receive = {

        case msg => println("===============> received %s " format msg)
```

```
    } // end receive

} // end MyQuartzActor

object MyQuartzActor {

    def main(str: Array[String]) {

        // start the Camel service
        startCamelService

        // create a quartz actor
        val myActor = actorOf[MyQuartzActor]

        // start the quartz actor
        myActor.start

    } // end main

} // end MyQuartzActor
```

The full working example is available for download here: http://www.dimingo.com/akka/examples/example-akka-quartz.tar.gz

You can launch it using the maven command:

```
$ mvn scala:run -DmainClass=com.dimingo.akka.MyQuartzActor
```

For more information about the Camel Quartz component, see here: http://camel.apache.org/quartz.html

1.3 AMQP (Scala)

Module stability: **STABLE**

Akka has an AMQP module which abstracts AMQP Connection, Producer and Consumer as Actors. It is fault-tolerant through supervisor hierarchies and does auto-reconnect and recreation of channels and message handlers on failure. It is currently based on the RabbitMQ Java client

Documentation is best described in code, so therefore you can find most of the usage described here:

- **Scala:** ExampleSession.scala
- **Java:** ExampleSessionJava.java. Be sure to check it out since it also contains examples for doing simple RPC with Strings or ProtoBuf messages.

1.3.1 Connection

To make a connection to the broker with default settings all that is needed is:

```
val connection = AMQP.newConnection()
```

This will connect using amqp://guest:guest@localhost:5672/

Specific connections can be made by providing 'ConnectionParameters'. Then you can also specify an array of addresses to connect to for fail-over purposes.

```
val myAddresses = Array(new Address("myhost.com", 5672), new Address("mybackuphost.com", 5672))
val connectionParameters = ConnectionParameters(myAddresses, "notguest", "password", "/vhost")
val connection = AMQP.newConnection(connectionParameters)
```

1.3.2 Connection callback

The 'ConnectionParameters' can also take an actor to receive the connection lifecycle messages. This is done via the 'connectionCallback' property on the 'ConnectionParameters'

```
val myCallback = actorOf(new Actor { def receive = {
  case Connected => log.info("Connection callback: Connected!")
  case Reconnecting => log.info("Connection callback: Reconnecting!")
  case Disconnected => log.info("Connection callback: Disconnected!")
}})
val connectionParameters = new ConnectionParameters(connectionCallback = Some(myCallback)
```

1.3.3 Channel callback

All communication a producer or consumer does happens over a channel. In addition to the pluggable return listener, there is also a possibility to plug in an actor which receives the channel lifecycle messages. This is done via the 'channelCallback' property on the 'ChannelParameters'.

```
val myCallback = actorOf( new Actor { def receive = {
  case Started => log.info("Channel callback: Started")
  case Restarting => log.info("Channel callback: Restarting")
  case Stopped => log.info("Channel callback: Stopped")
}}).start
val channelParameters = ChannelParameters(channelCallback = Some(myCallBack))
```

1.3.4 Exchange

As most of the messaging is done over exchanges, when creating producers or consumers the exchange settings can be specified with the 'ExchangeParameters'. This contains the exchange name and optionally an exchange type and the way the exchange is declared.

```
val default = ExchangeParameters("default_exchange")

val passiveDirect = ExchangeParameters("direct_exchange", Direct, PassiveDeclaration)

val activeDurableFanout = ExchangeParameters("fanout_exchange", Fanout, ActiveDeclaration(true, fa
```

Aside from using the predefined ExchangeTypes (`Direct`, `Fanout`, `Topic`, `Match`) also use `CustomExchange(...)`.

1.3.5 Producer

To create a basic producer, you can simply wrap the 'ExchangeParameters' in the 'ProducerParameters' and call the 'AMQP.newProducer' factory function. Optionally the 'ProducerParameters' takes a 'producerId' which will become the underlying actor id for lookup purposes in the 'ActorRegistry'.

Sending messages only takes a payload and a routingkey as a minumum, wrapped as a 'Message'.

```
val exchangeParameters = ExchangeParameters("my_topic_exchange", Topic)
val producer = AMQP.newProducer(connection, ProducerParameters(Some(exchangeParameters), producerI

producer ! Message("Some simple sting data".getBytes, "some.routing.key")
```

1.3.6 Consumer

A basic consumer does not take much more than a basic producer. Only addition is an actor that receives the eventual message deliveries. This delivery actor is specified via the 'ConsumerParameters'

```
val exchangeParameters = ExchangeParameters("my_topic_exchange", Topic)
val myConsumer = AMQP.newConsumer(connection, ConsumerParameters("some.routing.key", actorOf(new A
  case Delivery(payload, _, _, _, _, _) => log.info("Received delivery: %s", new String(payload))
}}), None, Some(exchangeParameters)))
```

Consumers are by default self acknowledging, but to be able to let the broker do the failover, you can overwrite the 'selfAcknowledging' property and send this acknowledgement yourself. This is done via both references in the 'Delivery' and a final confirmation that is send to the delivery handling actor.

```
val exchangeParameters = ExchangeParameters("my_topic_exchange", ExchangeType.Topic)
val myConsumer = AMQP.newConsumer(connection, ConsumerParameters("some.routing.key", actorOf(new A
  case Delivery(payload, _, deliveryTag, isRedeliver, _, sender) =>
    log.info("Received delivery: %s", new String(payload))
    sender ! Acknowledge(deliveryTag) // send the deliveryTag as acknowledgement to the sender (co
  case Acknowledged(deliveryTag) => () // tag acknowledged
}}), None, Some(exchangeParameters)))
```

N.B. 'selfAcknowledging=true' here still only means that the consuming actor does the acknowledgement for you. It is NOT auto acknowledgement on the amqp level, this is always disabled. A delivered message will alway get state 'message_unacknowledged' on the broker until successful processing. So making the consuming actor crash while handling the 'Delivery' will still put the message back on the queue. In addition one can look at the 'isRedeliver' property to check if the broker already tried to deliver the message before.

To check the message states on the broker, in a shell type: rabbitmqctl list_queues name messages messages_ready messages_unacknowledged

1.3.7 Load balancing

See this Gist: https://gist.github.com/858476

1.4 OSGi Support

Akka currently provides a certain kind of OSGi support which we call **//OSGi enabled//**. What does that mean?

First, all the Akka modules are **OSGi bundles**, i.e. they have got OSGi headers like *Bundle-SymbolicName*, *Bundle-Version*, *Import-Package* etc. in the manifest file (*META-INF/MANIFEST.MF*) of the the respective JAR archives.

Second, all necessary dependencies that are not already OSGi bundles (e.g. *commons-io-1.4.jar*) are wrapped into a **big dependencies bundle** exporting all their packages. While this is not the most modular approach, it is an easy path towards running Akka inside an OSGi container. This dependencies bundle is the artifact of the *akka-osgi-dependencies-bundle* module which itself is a subproject of the *akka-osgi* module.

Third, the *akka-osgi-assembly* module which also is a subproject of the *akka-osgi* module will **assemble everything you need** to run Akka inside an OSGi container. In its *target/<scala-version>/bundles* directory you will find the Akka bundles and all dependency bundles.

Last but not least, there is a simple OSGi example for Akka Core in the *akka-sample-osgi* module which itself is a subproject of the *akka-samples* module. It will start an *EchoActor* and send a message to it on bundle activation and shut it down on bundle deactivation. In order to run this example all you have to run an OSGi container like Eclipse Equinox or Apache Felix and install all the above mentioned bundles as well as this example bundle. An easy way to achieve this is using Pax Runner, step into the *target/<scala-version>/bundles* directory of the *akka-osgi-assembly* module and enter the following on the console:

```
pax-run.sh --p=equinox --profiles=log scan-dir:.@update file:../../../../../akka-samples/akka-samp
```

1.5 Spring Integration

Module stability: **STABLE**

Akkas integration with the Spring Framework supplies the Spring way of using the Typed Actor Java API and for CamelService configuration for *Standalone Spring applications*. It uses Spring's custom namespaces to create Typed Actors, supervisor hierarchies and a CamelService in a Spring environment.

To use the custom name space tags for Akka you have to add the XML schema definition to your spring configuration. It is available at http://repo.akka.io/akka-1.3.1.xsd. The namespace for Akka is:

```
xmlns:akka="http://repo.akka.io/schema/akka"
```

Example header for Akka Spring configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://repo.akka.io/schema/akka"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://repo.akka.io/schema/akka
       http://repo.akka.io/akka-1.3.1.xsd">
```

-

1.5.1 Actors

Actors in Java are created by extending the 'UntypedActor' class and implementing the 'onReceive' method.

Example how to create Actors with the Spring framework:

```
<akka:untyped-actor id="myActor"
                    implementation="com.biz.MyActor"
                    scope="singleton"
                    autostart="false"
                    depends-on="someBean"> <!-- or a comma-separated list of beans -->
        <property name="aProperty" value="somePropertyValue"/>
        <property name="aDependency" ref="someBeanOrActorDependency"/>
</akka:untyped-actor>
```

Supported scopes are singleton and prototype. Dependencies and properties are set with Springs `<property/>` element. A dependency can be either a `<akka:untyped-actor/>` or a regular `<bean/>`.

Get the Actor from the Spring context:

```
ApplicationContext context = new ClassPathXmlApplicationContext("akka-spring-config.xml");
ActorRef actorRef = (ActorRef) context.getBean("myActor");
```

1.5.2 Typed Actors

Here are some examples how to create Typed Actors with the Spring framework:

Creating a Typed Actor:

```
<beans>
  <akka:typed-actor id="myActor"
                    interface="com.biz.MyPOJO"
                    implementation="com.biz.MyPOJOImpl"
```

```
                    transactional="true"
                    timeout="1000"
                    scope="singleton"
                    depends-on="someBean"> <!-- or a comma-separated list of beans -->
        <property name="aProperty" value="somePropertyValue"/>
        <property name="aDependency" ref="someBeanOrActorDependency"/>
  </akka:typed-actor>
</beans>
```

Supported scopes are singleton and prototype. Dependencies and properties are set with Springs `<property/>` element. A dependency can be either a `<akka:typed-actor/>` or a regular `<bean/>`.

Get the Typed Actor from the Spring context:

```
ApplicationContext context = new ClassPathXmlApplicationContext("akka-spring-config.xml");
MyPojo myPojo = (MyPojo) context.getBean("myActor");
```

1.5.3 Remote Actors

For details on server managed and client managed remote actors see Remote Actor documentation.

Configuration for a client managed remote Actor

```
<akka:untyped-actor id="remote-untyped-actor"
                    implementation="com.biz.MyActor"
                    timeout="2000">
    <akka:remote host="localhost" port="9992" managed-by="client"/>
</akka:untyped-actor>
```

The default for 'managed-by' is "client", so in the above example it could be left out.

Configuration for a server managed remote Actor

Server side

```
<akka:untyped-actor id="server-managed-remote-untyped-actor"
                    implementation="com.biz.MyActor">
    <akka:remote host="localhost" port="9990" managed-by="server"/>
</akka:untyped-actor>

<!-- register with custom service name -->
<akka:untyped-actor id="server-managed-remote-untyped-actor-custom-id"
                    implementation="com.biz.MyActor">
    <akka:remote host="localhost" port="9990" service-name="my-service"/>
</akka:untyped-actor>
```

If the server specified by 'host' and 'port' does not exist it will not be registered.

Client side

```
<!-- service-name could be custom name or class name -->
<akka:actor-for id="client-1" host="localhost" port="9990" service-name="my-service"/>
```

Configuration for a client managed remote Typed Actor

```
<akka:typed-actor id="remote-typed-actor"
                  interface="com.biz.MyPojo"
                  implementation="com.biz.MyPojoImpl"
                  timeout="2000">
    <akka:remote host="localhost" port="9999" />
</akka:typed-actor>
```

Configuration for a server managed remote Typed Actor

Sever side setup

```
<akka:typed-actor id="server-managed-remote-typed-actor-custom-id"
                  interface="com.biz.IMyPojo"
                  implementation="com.biz.MyPojo"
                  timeout="2000">
     <akka:remote host="localhost" port="9999" service-name="mypojo-service"/>
</akka:typed-actor>
```

Client side setup

```
<!-- always specify the interface for typed actor -->
<akka:actor-for id="typed-client"
                interface="com.biz.MyPojo"
                host="localhost"
                port="9999"
                service-name="mypojo-service"/>
```

1.5.4 Dispatchers

Configuration for a Typed Actor or Untyped Actor with a custom dispatcher

If you don't want to use the default dispatcher you can define your own dispatcher in the spring configuration. For more information on dispatchers have a look at Dispatchers documentation.

```
<akka:typed-actor id="remote-typed-actor"
                  interface="com.biz.MyPOJO"
                  implementation="com.biz.MyPOJOImpl"
                  timeout="2000">
  <akka:dispatcher id="my-dispatcher" type="executor-based-event-driven" name="myDispatcher">
    <akka:thread-pool queue="unbounded-linked-blocking-queue" capacity="100" />
  </akka:dispatcher>
</akka:typed-actor>

<akka:untyped-actor id="untyped-actor-with-thread-based-dispatcher"
                    implementation="com.biz.MyActor">
      <akka:dispatcher type="thread-based" name="threadBasedDispatcher"/>
</akka:untyped-actor>
```

If you want to or have to share the dispatcher between Actors you can define a dispatcher and reference it from the Typed Actor configuration:

```
<akka:dispatcher id="dispatcher-1"
                 type="executor-based-event-driven"
                 name="myDispatcher">
  <akka:thread-pool queue="bounded-array-blocking-queue"
```

```
                    capacity="100"
                    fairness="true"
                    core-pool-size="1"
                    max-pool-size="20"
                    keep-alive="3000"
                    rejection-policy="caller-runs-policy"/>
</akka:dispatcher>

<akka:typed-actor id="typed-actor-with-dispatcher-ref"
                  interface="com.biz.MyPOJO"
                  implementation="com.biz.MyPOJOImpl"
                  timeout="1000">
    <akka:dispatcher ref="dispatcher-1"/>
</akka:typed-actor>
```

The following dispatcher types are available in spring configuration:

- executor-based-event-driven
- executor-based-event-driven-work-stealing
- thread-based

The following queue types are configurable for dispatchers using thread pools:

- bounded-linked-blocking-queue
- unbounded-linked-blocking-queue
- synchronous-queue
- bounded-array-blocking-queue

If you have set up your IDE to be XSD-aware you can easily write your configuration through auto-completion.

1.5.5 Stopping Typed Actors and Untyped Actors

Actors with scope singleton are stopped when the application context is closed. Actors with scope prototype must be stopped by the application.

1.5.6 Supervisor Hierarchies

The supervisor configuration in Spring follows the declarative configuration for the Java API. Have a look at Akka's approach to fault tolerance.

Example spring supervisor configuration

```
<beans>
  <akka:supervision id="my-supervisor">

    <akka:restart-strategy failover="AllForOne"
                           retries="3"
                           timerange="1000">
      <akka:trap-exits>
        <akka:trap-exit>java.io.IOException</akka:trap-exit>
      </akka:trap-exits>
    </akka:restart-strategy>

    <akka:typed-actors>
      <akka:typed-actor interface="com.biz.MyPOJO"
                        implementation="com.biz.MyPOJOImpl"
                        lifecycle="permanent"
```

```
                        timeout="1000"/>
      <akka:typed-actor interface="com.biz.AnotherPOJO"
                        implementation="com.biz.AnotherPOJOImpl"
                        lifecycle="temporary"
                        timeout="1000"/>
      <akka:typed-actor interface ="com.biz.FooBar"
                        implementation ="com.biz.FooBarImpl"
                        lifecycle="permanent"
                        transactional="true"
                        timeout="1000" />
    </akka:typed-actors>
  </akka:supervision>

  <akka:supervision id="supervision-untyped-actors">
    <akka:restart-strategy failover="AllForOne" retries="3" timerange="1000">
      <akka:trap-exits>
        <akka:trap-exit>java.io.IOException</akka:trap-exit>
        <akka:trap-exit>java.lang.NullPointerException</akka:trap-exit>
      </akka:trap-exits>
    </akka:restart-strategy>
    <akka:untyped-actors>
      <akka:untyped-actor implementation="com.biz.PingActor"
                        lifecycle="permanent"/>
      <akka:untyped-actor implementation="com.biz.PongActor"
                        lifecycle="permanent"/>
      </akka:untyped-actors>
  </akka:supervision>

</beans>
```

Get the TypedActorConfigurator from the Spring context

```
TypedActorConfigurator myConfigurator = (TypedActorConfigurator) context.getBean("my-supervisor");
MyPojo myPojo = (MyPOJO) myConfigurator.getInstance(MyPojo.class);
```

1.5.7 Property Placeholders

The Akka configuration can be made available as property placeholders by using a custom property placeholder configurer for Configgy:

```
<akka:property-placeholder location="akka.conf"/>

<akka:untyped-actor id="actor-1" implementation="com.biz.MyActor" timeout="${akka.actor.timeout}"
  <akka:remote host="${akka.remote.server.hostname}" port="${akka.remote.server.port}"/>
</akka:untyped-actor>
```

1.5.8 Camel configuration

For details refer to the *Camel* documentation:

- CamelService configuration for *Standalone Spring applications*
- Access to Typed Actors *Using Spring*

1.6 Scalaz

This is a work in progress and is mostly an outline at this point. More detailed information to come soon.

1.6.1 Introduction

The akka-scalaz module provides implementations of most Scalaz type classes. The intended audience of this documentation is someone who is not familiar with Scalaz, as the methods are the same as long as the relevant type classes are implemented. At the moment only Future, functions returning Future, and Actors that return a Future benefit from this module and should behave similarily to Scalaz's Promise.

1.6.2 Futures

TODO: Add examples

To use this module, you must import scalaz and the type classes for Future:

```
import scalaz._
import Scalaz._
import akka.scalaz.futures._
```

Note: Whenever an additional collection is required to explain the use of a method, List is used. Any other monad/functor/foldable/etc can be used in it's placed, as long as the applicable type classes are defined in scalaz or are in scope elsewhere.

map

```
Future[A] map (A => B): Future[B]
Future[A] >| (=> B): Future[B]
Future[List[A]] map2 (A => B): Future[List[B]]
List[Future[A]] map2 (A => B): List[Future[B]]
```

flatMap

```
Future[A] flatMap (A => Future[B]): Future[B]
Future[A] >>= (A => Future[B]): Future[B]
Future[Future[A]] join: Future[A]
```

foreach

```
Future[A] foreach (A => Unit): Unit
Future[A] |>| (A => Unit): Unit
```

applicative

```
(Future[A] <**> Future[B])((A, B) => C): Future[C]
Future[A] <|*|> Future[B]: Future[(A, B)]
Future[A] |@| Future[B]: ApplicativeBuilder[Future, A, B]
```

traverse

```
List[A] traverse (A => Future[B]): Future[List[B]]
List[Future[A]] sequence: Future[List[A]]
```

fold

```
List[A].foldl(Future[B])((Future[B], A) => Future[B]): Future[B]
List[A] foldLeftM(B)((B, A) => Future[B]): Future[B]
List[Future[A]] foldl1 ((Future[A], Future[A]) => Future[A]): Option[Future[A]]
List[A].foldr(Future[B])((A, => Future[B]) => Future[B]): Future[B]
List[A] foldRightM(B)((B, A) => Future[B]): Future[B]
List[Future[A]] foldr1 ((Future[A], => Future[A]) => Future[A]): Option[Future[A]]
```

monoid

```
List[A] foldMapDefault (A => Future[B]): Future[B]
List[Future[A]] collapse: Future[A]
List[A] foldMap (A => Future[B]): Future[B]
List[Future[A]] sum: Future[A]
List[Future[A]] sumr: Future[A]
Future[A] |+| Future[A]: Future[A]
A +>: Future[A]: Future[A]
```

composition

```
(A => Future[B]) >=> (B => Future[C]): A => Future[C]
```

misc

```
Future[A] <+> Future[A]: Future[A]
Future[A] getOrElseM Future[Option[A]]: Future[A]
Future[A] copure: A
Future[A] fpure[List]: Future[List[A]]
```

1.6.3 Actors

An ActorRef can be implicitly converted into a function "Any => Future[Any]" and used wherever that function is accepted. For example:

```
ActorRef >=> ActorRef: Any => Future[Any]
Future[A] flatMap ActorRef: Future[Any]
List[A] traverse ActorRef: Future[List[Any]]
```

1.6.4 Concurrency

TODO: Explain when and where the given functions are applied to the value of a Future, and how to manipulate this. TODO: Configuration options

1.6.5 Type Classes

Future

Pure Functor Bind Each Monad (implicitly from Pure and Bind) Apply (implicitly from Functor and Bind) Applicative (implicitly from Pure and Apply) Cojoin Copure Comonad (implicitly from Functor, Cojoin, and Copure)

The following type classes are available if the Future's contained type also implements the same type class: Semigroup Zero Monoid (implicitly from Semigroup and Zero)

CHAPTER

TWO

INFORMATION FOR DEVELOPERS

2.1 Building Akka Modules

This section describes how to build and run Akka Modules from the latest source code.

2.1.1 Get the source code

Akka uses Git and is hosted at Github.

You first need Git installed on your machine. You can then clone the source repositories:

- Akka repository from http://github.com/akka/akka
- Akka Modules repository from http://github.com/akka/akka-modules

For example:

```
git clone git://github.com/akka/akka.git
git clone git://github.com/akka/akka-modules.git
```

If you have already cloned the repositories previously then you can update the code with `git pull`:

```
git pull origin master
```

2.1.2 SBT - Simple Build Tool

Akka is using the excellent SBT build system. So the first thing you have to do is to download and install SBT. You can read more about how to do that here .

The SBT commands that you'll need to build Akka are all included below. If you want to find out more about SBT and using it for your own projects do read the SBT documentation.

The Akka SBT build file is `project/build/AkkaProject.scala` with some properties defined in `project/build.properties`.

2.1.3 Building Akka

First make sure that you are in the akka code directory:

```
cd akka
```

Fetching dependencies

SBT does not fetch dependencies automatically. You need to manually do this with the `update` command:

```
sbt update
```

Once finished, all the dependencies for Akka will be in the `lib_managed` directory under each module: akka-actor, akka-stm, and so on.

Note: you only need to run update the first time you are building the code, or when the dependencies have changed.

Building

To compile all the Akka core modules use the `compile` command:

```
sbt compile
```

You can run all tests with the `test` command:

```
sbt test
```

If compiling and testing are successful then you have everything working for the latest Akka development version.

Publish to local Ivy repository

If you want to deploy the artifacts to your local Ivy repository (for example, to use from an SBT project) use the `publish-local` command:

```
sbt publish-local
```

Publish to local Maven repository

If you want to deploy the artifacts to your local Maven repository use:

```
sbt publish-local publish
```

SBT interactive mode

Note that in the examples above we are calling `sbt compile` and `sbt test` and so on. SBT also has an interactive mode. If you just run `sbt` you enter the interactive SBT prompt and can enter the commands directly. This saves starting up a new JVM instance for each command and can be much faster and more convenient.

For example, building Akka as above is more commonly done like this:

```
% sbt
[info] Building project akka 1.3.1 against Scala 2.9.0
[info]    using AkkaParentProject with sbt 0.7.6 and Scala 2.7.7
> update
[info]
[info] == akka-actor / update ==
...
[success] Successful.
[info]
[info] Total time ...
> compile
...
> test
...
```

SBT batch mode

It's also possible to combine commands in a single call. For example, updating, testing, and publishing Akka to the local Ivy repository can be done with:

```
sbt update test publish-local
```

2.1.4 Building Akka Modules

To build Akka Modules first build and publish Akka to your local Ivy repository as described above. Or using:

```
cd akka
sbt update publish-local
```

Then you can build Akka Modules using the same steps as building Akka. First update to get all dependencies (including the Akka core modules), then compile, test, or publish-local as needed. For example:

```
cd akka-modules
sbt update publish-local
```

Microkernel distribution

To build the Akka microkernel (the same as the Akka Modules distribution download) use the `dist` command:

```
sbt dist
```

The distribution can be found in the `dist/microkernel/target/dist` directory.

There is a start script in the `bin` directory that can be used to start up the microkernel.

The microkernel will boot up and install any applications that reside in the distribution's `deploy` directory. You can deploy your own applications into the `deploy` directory. There is a simple sample application included, see *Hello Microkernel*.

Configuration files are in the `config` directory. Modify these as needed.

2.1.5 Scripts

Linux/Unix init script

Here is a Linux/Unix init script that can be very useful:

http://github.com/akka/akka/blob/master/scripts/akka-init-script.sh

Copy and modify as needed.

Simple startup shell script

This little script might help a bit:

http://github.com/akka/akka/blob/master/scripts/run_akka.sh

Copy and modify as needed.

2.1.6 Dependencies

If you are managing dependencies by hand you can find the dependencies for each module by looking in the `lib_managed` directories. For example, this will list all compile dependencies (providing you have the source code and have run `sbt update`):

```
cd akka
ls -1 */lib_managed/compile
```

You can also look at the Ivy dependency resolution information that is created on `sbt update` and found in `~/.ivy2/cache`. For example, the `.ivy2/cache/se.scalablesolutions.akka-akka-kernel-compile.xml` file contains the resolution information for the akka-kernel module compile dependencies. If you open this file in a web browser you will get an easy to navigate view of dependencies.

CHAPTER

THREE

LINKS

- Downloads
- Source Code
- Scaladoc API
- Akka Core Documentation
- Issue Tracking