WildFly Camel

wildflyext

Table of Contents

1. Introduction
2. Getting Started
    i. Compatibility
3. Features
    i. Camel Context Definitions
    ii. Camel Context Deployments
    iii. Management Console
    iv. Arquillian Test Support
4. Configuration
    i. Subsystem
    ii. Deployment
5. JavaEE Integration
    i. CDI
    ii. JAXB
    iii. JAX-RS
    iv. JAX-WS
    v. JMS
    vi. JMX
    vii. JNDI
    viii. JPA
6. Camel Components
    i. camel-activemq
    ii. camel-atom
    iii. camel-bindy
    iv. camel-cdi
    v. camel-cxf
    vi. camel-dozer
    vii. camel-ejb
    viii. camel-file2
    ix. camel-ftp
    x. camel-http4
    xi. camel-hl7
    xii. camel-jaxb
    xiii. camel-jms
    xiv. camel-jmx
    xv. camel-jpa
    xvi. camel-lucene
    xvii. camel-mail
    xviii. camel-mina2
    xix. camel-mqtt
    xx. camel-mvel
    xxi. camel-netty4

    xxii. camel-ognl
    xxiii. camel-quartz2
    xxiv. camel-rest
    xxv. camel-rss
    xxvi. camel-saxon
    xxvii. camel-script
    xxviii. camel-servlet
    xxix. camel-sql
    xxx. camel-weather
    xxxi. camel-xstream
7. Data Formats
8. Languages
9. Cloud
    i. Docker
    ii. OpenShift
    iii. Beanstalk
10. Security
    i. JAX-RS
    ii. JAX-WS
    iii. JMS
    iv. Policy
11. Developer Guide

WildFly Camel

Version: 2.3.0

Provides Apache Camel integration with the WildFly Application Server.

The WildFly-Camel Subsystem allows you to add Camel Routes as part of the WildFly configuration. Routes can be deployed as part of JavaEE applications. JavaEE components can access the Camel Core API and various Camel Component APIs.

Your Enterprise Integration Solution can be architected as a combination of JavaEE and Camel functionality.

Getting Started

This chapter takes you through the first steps of getting WildFly-Camel and provides the initial pointers to get up and running.

Download the Distribution

WildFly Camel is distributed as

1. WildFly Patch - wildfly-camel-patch
2. Docker Image - wildflyext/wildfly-camel

Installing the Camel Subsystem

Simply apply the patch to the respective wildfly version. For possible WildFly target versions, see the compatibility page

Standalone Server

In your WildFly home directory run ...

```
$ bin/standalone.sh -c standalone-camel.xml
...
10:50:02,833 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 31) JBAS018559: De
10:50:02,834 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 31) JBAS018559: De
10:50:02,848 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961: Http management in
10:50:02,849 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951: Admin console list
10:50:02,849 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874: WildFly 8.1.0.Fina
```

Domain Mode

Similarly, for the WildFly Domain Mode run ...

```
$ bin/domain.sh -c domain-camel.xml
```

Docker Image

The easiest and most portable way to run WildFly-Camel is to use the wildflyext/wildfly-camel distribution.

```
$ docker run --rm -ti -p 8080:8080 -p 9990:9990 -e WILDFLY_MANAGEMENT_USER=admin -e WILDF
```

WildFly Compatibility

The WildFly compatibility matrix

8.2.0.Final	2.1, 2.2, 2.3
8.1.0.Final	2.0

Features

This chapter provides more detailed information about WildFly-Camel features.

- Camel Context Definitions
- Camel Context Deployments
- Management Console
- Arquillian Support
- Deployment Configuration

Camel Context Definitions

Camel Contexts can be configured in standalone-camel.xml and domain.xml as part of the subsystem definition like this

```
<subsystem xmlns="urn:jboss:domain:camel:1.0">
   <camelContext id="system-context-1">
     <![CDATA[
     <route>
       <from uri="direct:start"/>
       <transform>
         <simple>Hello #{body}</simple>
       </transform>
     </route>
     ]]>
   </camelContext>
</subsystem>
```

Camel Context Deployments

Camel contexts can be deployed to WildFly with a **-camel-context.xml** suffix.

1. As a standalone XML file
2. As part of another supported deployment

A deployment may contain multiple **-camel-context.xml** files.

A deployed Camel context is CDI injectable like this

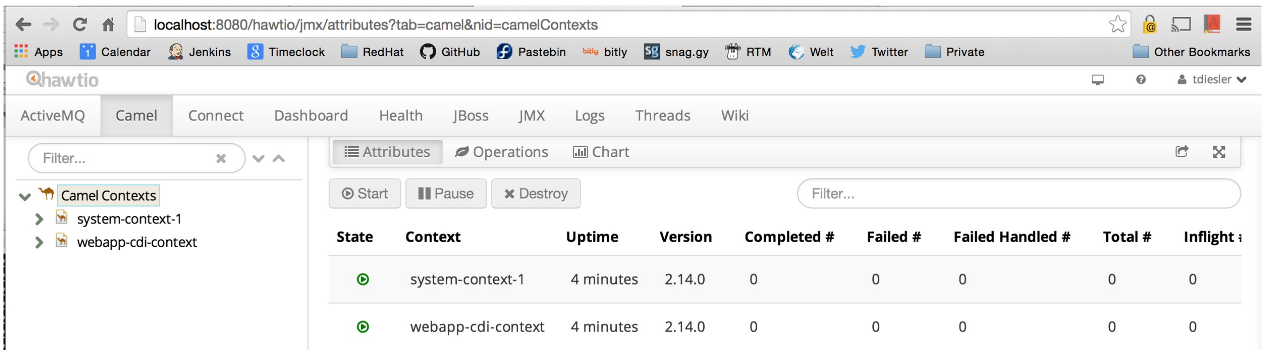
```
@Resource(name = "java:jboss/camel/context/mycontext")
CamelContext camelContext;
```

Management Console

By default, access to management consoles is secured. We therefore need to setup a Management User first.

```
$ bin/add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
```

The Hawt.io console should show the camel context from subsystem configuration.



State	Context	Uptime	Version	Completed #	Failed #	Failed Handled #	Total #	Inflight
●	system-context-1	4 minutes	2.14.0	0	0	0	0	0
●	webapp-cdi-context	4 minutes	2.14.0	0	0	0	0	0

Arquillian Test Support

The WildFly Camel test suite uses the WildFly Arquillian managed container. This can connect to an already running WildFly instance or alternatively start up a standalone server instance when needed.

A number of test enrichers have been implemented that allow you have these WildFly Camel specific types injected into your Arquillian test cases.

```
@ArquillianResource
CamelContextFactory contextFactory;

@ArquillianResource
CamelContextRegistry contextRegistry;
```

Configuration

This chapter details information about Camel subsytem and deployment configuration.

- Subsystem
- Deployment

Camel Subsystem Configuration

The subsystem configuration may contain static system routes

```
<subsystem xmlns="urn:jboss:domain:camel:1.0">
   <camelContext id="system-context-1">
     <![CDATA[
     <route>
       <from uri="direct:start"/>
       <transform>
         <simple>Hello #{body}</simple>
       </transform>
     </route>
     ]]>
   </camelContext>
</subsystem>
```

These routes are started automatically.

Camel Deployment Configuration

If you want to fine tune the default configuration of your Camel deployment, you can edit either the `WEB-INF/jboss-all.xml` or `META-INF/jboss-all.xml` configuration file in your deployment.

Use a `<jboss-camel>` XML element within the `jboss-all.xml` file to control the camel configuration.

Disabling the Camel Subsystem

If you don't want the camel subsystem to be added into your deployment, set the `enabled="false"` attribute on the `jboss-camel` XML element.

Example `jboss-all.xml` file:

```
<jboss umlns="urn:jboss:1.0">
  <jboss-camel xmlns="urn:jboss:jboss-camel:1.0" enabled="false"/>
</jboss>
```

Selecting Components

If you add nested `<component>` or `<component-module>` XML elements, then instead of adding the default list of Camel components to your deployment, only the specified components will be added to your deployment.

Example `jboss-all.xml` file:

```
<jboss umlns="urn:jboss:1.0">
  <jboss-camel xmlns="urn:jboss:jboss-camel:1.0">
    <component name="camel-ftp"/>
    <component-module name="org.apache.camel.component.rss"/>
  </jboss-camel>
</jboss>
```

JavaEE Integration

This chapter details information about integration points with JavaEE.

- CDI
- JAX-RS
- JAX-WS
- JMS
- JMX
- JNDI
- JPA

Integration with CDI

CDI integration is provided by camel-cdi.

A Context with an associated route can be provided like this

```
@Startup
@ApplicationScoped
@ContextName("cdi-context")
public class MyRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("direct:start").transform(body().prepend("Hi"));
    }
}
```

and consumed like this

```
    @Inject
    @ContextName("cdi-context")
    private CamelContext camelctx;
```

Integration with JAXB

JAXB support is provided through the Camel JAXB data format.

Camel supports unmarshalling XML data to JAXB annotated classes and marshalling from classes to XML. What follows demonstrates a simple Camel route for marshalling and unmarshalling with the Camel JAXB data format class.

JAXB Annotated class

```
@XmlRootElement(name = "customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer implements Serializable {

    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

JAXB Class XML representation

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customer xmlns="http://org/wildfly/test/jaxb/model/Customer">
    <firstName>John</firstName>
    <lastName>Doe</lastName>
</customer>
```

Camel JAXB Unmarshalling

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext(getClass().getClassLoade

final JaxbDataFormat jaxb = new JaxbDataFormat();
jaxb.setContextPath("org.wildfly.camel.test.jaxb.model");

camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .unmarshal(jaxb);
    }
});
camelctx.start();

ProducerTemplate producer = camelctx.createProducerTemplate();

// Send an XML representation of the customer to the direct:start endpoint
Customer customer = producer.requestBody("direct:start", readCustomerXml(), Customer.clas
Assert.assertEquals("John", customer.getFirstName());
Assert.assertEquals("Doe", customer.getLastName());
```

Camel JAXB Marshalling

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext();

final JaxbDataFormat jaxb = new JaxbDataFormat();
jaxb.setContextPath("org.wildfly.camel.test.jaxb.model");

camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .marshal(jaxb);
    }
});
camelctx.start();

ProducerTemplate producer = camelctx.createProducerTemplate();
Customer customer = new Customer("John", "Doe");
String customerXML = producer.requestBody("direct:start", customer, String.class);
Assert.assertEquals(readCustomerXml(), customerXML);
```

Integration with JAX-RS

JAX-RS consumer support is provided by two mechanisms. The Camel REST DSL and by leveraging the WildFly RESTEasy JAX-RS subsystem to publish RESTful services in conjunction with the CamelProxy.

JAX-RS producer endpoint support is provided by the camel-restlet component.

> **IMPORTANT: At present the WildFly Camel Subsytem does not support JAX-RS CXF or Restlet consumers. E.g endpoints defined as from("cxfrs://...") or from("restlet://"). Other JAX-RS consumer endpoint solutions are described later in this document.**

JAX-RS Restlet Producer

The following code example uses the camel-restlet component to consume a JAX-RS service which has been deployed by the WildFly JAX-RS subsystem.

JAX-RS service

This RESTful service desribes a simple interface which will return a list of Customer POJOs as a JSON string. The service will be published to the path `GET /rest/customer`.

```
// Service interface
@Path("/customer")
public interface CustomerService {
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  Response getCustomers();
}

// Service implementation
public class CustomerServiceImpl implements CustomerService {
  @Override
  public Response getCustomers() {
    List<Customer> customers = customerService.getCustomers();
    return Response.ok(customers).build();
  }
}

// Application bootstrap class
@ApplicationPath("/rest")
public class RestApplication extends Application {
  @Override
  public Set<Class<?>> getClasses() {
    final Set<Class<?>> classes = new HashSet<>();
    classes.add(CustomerServiceImpl.class);
    return classes;
  }
}
```

Camel route configuration

The Restlet component can be used to consume RESTful services as shown in the following Camel RouteBuilder example. CDI in conjunction with the camel-cdi component is used to bootstrap the RouteBuilder and CamelContext.

```
@Startup
@ApplicationScoped
@ContextName("rest-camel-context")
public class RestProducerRouteBuilder extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("direct:start")
    .to("restlet://http://localhost:8080/rest/customer")
    .process(new Processor() {
      @Override
      public void process(Exchange exchange) throws Exception {
        // Do something useful with the REST service response
      }
    });
  }
}
```

JAX-RS Consumer with the Camel REST DSL

The Camel REST DSL gives the capability to write Camel routes that act as JAX-RS consumers. The following RouteBuilder class demonstrates this.

```
@Startup
@ApplicationScoped
@ContextName("rest-camel-context")
public class RestConsumerRouteBuilder extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    // Use the camel-servlet component to provide REST integration
    restConfiguration().component("servlet")
      .contextPath("/rest").port(8080).bindingMode(RestBindingMode.json);

    rest("/customer")
      // GET /rest/customer
      .get()
        .produces(MediaType.APPLICATION_JSON)
        .to("direct:getCustomers")
        // GET /rest/customer/1
      .get("/{id}")
        .produces(MediaType.APPLICATION_JSON)
        .to("direct:getCustomer")
      // POST /rest/customer
      .post()
        .type(Customer.class)
        .to("direct:createCustomer");
      // PUT /rest/customer
      .put()
        .type(Customer.class)
        .to("direct:updateCustomer");
      // DELETE /rest/customer/1
```

```
        .delete("/{id}")
          .to("direct:deleteCustomer");
  }
}
```

Note that the REST DSL configuration starts with `restConfiguration().component("servlet")`. **The WildFly Camel Subsystem only supports camel-servlet for use with the REST DSL. Attempts to configure other components will not work.**

By setting the binding mode, Camel can marshal and unmarshal JSON data either by specifying a 'produces()' or 'type()' configuration step.

JAX-RS Consumer with CamelProxy

JAX-RS consumer endpoints can be configured using the CamelProxy. The proxy enables you to proxy a producer sending to an Endpoint by a regular interface. When clients invoke methods on this interface, the proxy performs a request / reply to a specified endpoint.

Below is an example JAX-RS service interface and implementation.

```
// Service interface
@Path("/customer")
public interface CustomerService {
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  Response getCustomers();

  @PUT
  Response updateCustomer();
}

// Service implementation
public class CustomerServiceImpl implements CustomerService {
  @Inject
  @ContextName("rest-camel-context")
  private CamelContext context;

  @Produce(uri="direct:rest")
  private CustomerService customerServiceProxy;

  @Override
  public Response getCustomers() {
    return customerServiceProxy.getCustomers();
  }

  @Override
  public Response updateCustomer(Customer customer) {
    return customerServiceProxy.updateCustomer(customer);
  }
}
```

Notice in the above code example that `CustomerServiceImpl` delegates all method calls to a customerServiceProxy object which has been annotated with `@Produce`. This annotation is important as it configures a proxy for the `direct:rest` endpoint against the `CustomerService` interface. Whenever any of the REST service methods are invoked by clients, the `direct:rest` camel route is triggered.

The RouteBuilder class implements logic for each REST service method invocation.

```
from("direct:rest")
  .process(new Processor() {
      @Override
      public void process(Exchange exchange) throws Exception {
        BeanInvocation beanInvocation = exchange.getIn().getBody(BeanInvocation.class);
        String methodName = beanInvocation.getMethod().getName();

        if (methodName.equals("getCustomers")) {
          List<Customer> customers = customerService.findAllCustomers();
          exchange.getOut().setBody(Response.ok(customers).build());
        } else if(methodName.equals("updateCustomer")) {
          Customer updatedCustomer = (Customer) beanInvocation.getArgs()[0];
          customerService.updateCustomer(updatedCustomer);
          exchange.getOut().setBody(Response.ok().build());
        }
      }
  });
```

In the above RouteBuilder a `Processor` handles REST service method invocations that have been proxied through the `direct:rest` endpoint. The exchange message body will be an instance of `BeanInvocation`. This can be used to determine which web service method was invoked and what arguments were passed to it. In this example some simple logic is used to return results to the client based on the name of the method that was called.

Security

Refer to the JAX-RS security section.

Code examples on GitHub

An example camel-rest application is available on GitHub.

Integration with JAX-WS

WebService support is provided through the camel-cxf component which integrates with the WildFly WebServices subsystem that also uses Apache CXF.

> **IMPORTANT: At present the WildFly Camel Subsytem does not support CXF consumers. E.g endpoints defined as from("cxf://..."). Although, it is possible to mimic CXF consumer behaviour using the CamelProxy.**

JAX-WS CXF Producer

The following code example uses CXF to consume a web service which has been deployed by the WildFly web services subsystem.

JAX-WS web service

The following simple web service has a simple 'greet' method which will concatenate two string arguments together and return them.

When the WildFly web service subsystem detects classes containing JAX-WS annotations, it bootstraps a CXF endpoint. In this example the service endpoint will be located at http://hostname:port/context-root/greeting.

```
// Service interface
@WebService(name = "greeting")
public interface GreetingService {
    @WebMethod(operationName = "greet", action = "urn:greet")
    String greet(@WebParam(name = "message") String message, @WebParam(name = "name") Str
}

// Service implementation
public class GreetingServiceImpl implements GreetingService{
    public String greet(String message, String name) {
        return message + " " + name ;
    }
}
```

Camel route configuration

This RouteBuilder configures a CXF producer endpoint which will consume the 'greeting' web service defined above. CDI in conjunction with the camel-cdi component is used to bootstrap the RouteBuilder and CamelContext.

```
@Startup
@ApplicationScoped
@ContextName("cxf-camel-context")
public class CxfRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .to("cxf://http://localhost:8080/example-camel-cxf/greeting?serviceClass=" + Gree
    }
}
```

The greeting web service 'greet' requires two parameters. These can be supplied to the above route by way of a `ProducerTemplate`. The web service method argument values are configured by constructing an object array which is passed as the exchange body.

```
String message = "Hello"
String name = "Kermit"

ProducerTemplate producer = camelContext.createProducerTemplate();
Object[] serviceParams = new Object[] {message, name};
String result = producer.requestBody("direct:start", serviceParams, String.class);
```

JAX-WS Consumer with CamelProxy

JAX-WS consumer endpoints can be configured using the CamelProxy. The proxy enables you to proxy a producer sending to an Endpoint by a regular interface. When clients invoke methods on this interface, the proxy performs a request / reply to a specified endpoint.

Below is an example JAX-WS web service interface and implementation.

```
// Service interface
@WebService(name = "greeting")
public interface GreetingService {
    @WebMethod(operationName = "greet", action = "urn:greet")
    String greet(@WebParam(name = "name") String name);

    @WebMethod(operationName = "greetWithMessage", action = "urn:greetWithMessage")
    String greetWithMessage(@WebParam(name = "message") String message, @WebParam(name =
}

// Service implementation
@WebService(serviceName="greeting", endpointInterface = "org.wildfly.camel.examples.jaxws
public class GreetingServiceImpl {
    @Produce(uri="direct:start")
    GreetingService greetingService;

    @WebMethod(operationName = "greet")
    public String greet(@WebParam(name = "name") String name) {
        return greetingService.greet(name);
    }

    @WebMethod(operationName = "greetWithMessage")
    public String greetWithMessage(@WebParam(name = "message") String message, @WebParam(
        return greetingService.greetWithMessage(message, name);
    }
}
```

Notice in the above code example that `GreetingServiceImpl` delegates all method calls to a greetingService object which has been annotated with `@Produce`. This annotation is important as it configures a proxy for the `direct:start` endpoint against the `GreetingService` interface. Whenever any of the web service methods are invoked by clients, the `direct:start` camel route is triggered.

The RouteBuilder class implements logic for each web service method invocation.

```
@Startup
@ApplicationScoped
@ContextName("jaxws-camel-context")
public class JaxwsRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .process(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                BeanInvocation beanInvocation = exchange.getIn().getBody(BeanInvocation.c
                String methodName = beanInvocation.getMethod().getName();

                if(methodName.equals("greet")) {
                    String name = exchange.getIn().getBody(String.class);
                    exchange.getOut().setBody("Hello " + name);
                } else if(methodName.equals("greetWithMessage")) {
                    String message = (String) beanInvocation.getArgs()[0];
                    String name = (String) beanInvocation.getArgs()[1];
                    exchange.getOut().setBody(message + " " + name);
                } else {
                    throw new IllegalStateException("Unknown method invocation " + method
                }
            }
        });
    }
```

In the above RouteBuilder a `Processor` handles web service method invocations that have been proxied through the `direct:start` endpoint. The exchange message body will be an instance of `BeanInvocation`. This can be used to determine which web service method was invoked and what arguments were passed to it. In this example some simple logic is used to return results to the client based on the name of the method that was called.

Security

Refer to the JAX-WS security section.

Code examples on GitHub

Example JAX-WS applications are available on GitHub.

- camel-cxf application
- camel-jaxws application

Integration with JMS

Messaging support is provided through the camel-jms component which integrates with the WildFly Messaging (HornetQ) subsystem.

Integration with other JMS implementations is possible through configuration of vendor specific resource adapters, or if not available, by using the JBoss Generic JMS resource adapter.

WildFly JMS configuration

The WildFly messaging subsystem can be configured from within the standard WildFly XML configuration files, standalone.xml or domain.xml.

For the examples that follow we use the embedded HornetQ in memory instance. We first configure a new JMS queue on the messaging subsystem by adding the following XML configuration to the jms-destinations section.

```
<jms-queue name="WildFlyCamelQueue">
  <entry name="java:/jms/queue/WildFlyCamelQueue"/>
</jms-queue>
```

Alternatively you could use a CLI script to add the queue.

```
jms-queue add --queue-address=WildFlyCamelQueue --entries=queue/WildFlyCamelQueue,java:/j
```

Or, you could create a `messaging-deployment` configuration within a custom jms.xml deployment descriptor. See section 'Deployment of -jms.xml files' within the WildFly messaging subsystem documentation for more information.

Camel route configuration

The following JMS producer and consumer examples make use of WildFly's embedded HornetQ sever to publish and consume messages to and from destinations.

The examples also use CDI in conjunction with the camel-cdi component. JMS ConnectionFactory instances are injected into the Camel RouteBuilder through JNDI lookups.

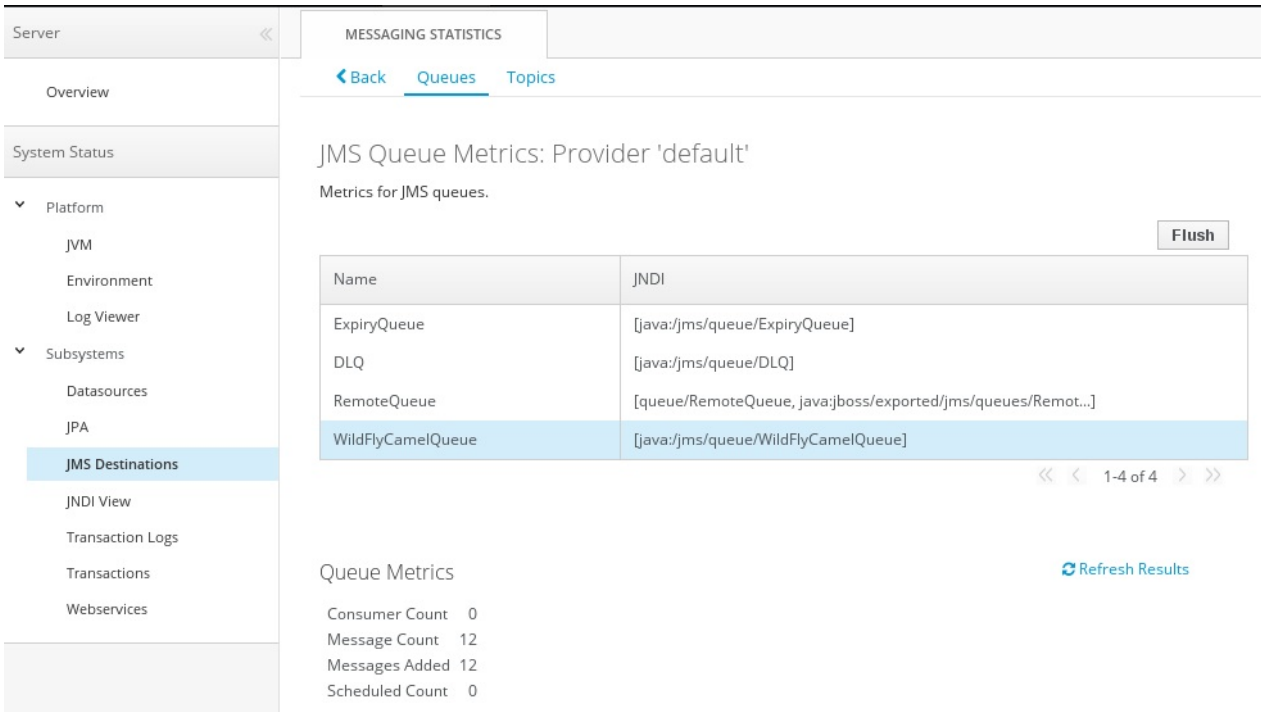
JMS Producer

The RouteBuilder begins by injecting the `DefaultJMSConnectionFactory` connection factory from JNDI. If you're wondering where the connection factory is defined, look at the WildFly XML configuration and you'll see it is defined within the messaging subsystem.

Next a timer endpoint runs every 10 seconds to send an XML payload to the WildFlyCamelQueue destination that we configured earlier.

```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JmsRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:jboss/DefaultJMSConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Override
  public void configure() throws Exception {
    JmsComponent component = new JmsComponent();
    component.setConnectionFactory(connectionFactory);

    getContext().addComponent("jms", component);

    from("timer://sendJMSMessage?fixedRate=true&period=10000")
    .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting></me
    .to("jms:queue:WildFlyCamelQueue")
    .log("JMS Message sent");
  }
}
```

A log message will be output to the console each time a JMS message is added to the WildFlyCamelQueue destination. To verify that the messages really are being placed onto the queue, we can use the WildFly administration console.



JMS Queue Metrics: Provider 'default'

Metrics for JMS queues.

Name	JNDI
ExpiryQueue	[java:/jms/queue/ExpiryQueue]
DLQ	[java:/jms/queue/DLQ]
RemoteQueue	[queue/RemoteQueue, java:jboss/exported/jms/queues/Remot...]
WildFlyCamelQueue	[java:/jms/queue/WildFlyCamelQueue]

Queue Metrics

Consumer Count	0
Message Count	12
Messages Added	12
Scheduled Count	0

JMS Consumer

To consume JMS messages the Camel RouteBuilder implementation is similar to the producer example.

As before, the connection factory is discovered from JNDI, injected and set on the JMSComponent instance.

When the JMS endpoint consumes messages from the WildFlyCamelQueue destination, the content is logged to the console.

```
@Override
public void configure() throws Exception {
  JmsComponent component = new JmsComponent();
  component.setConnectionFactory(connectionFactory);

  getContext().addComponent("jms", component);

  from("jms:queue:WildFlyCamelQueue")
  .to("log:jms?showAll=true");
}
```

JMS Transactions

To enable Camel JMS routes to participate in JMS transactions, some additional configuration is required. Since camel-jms is built around spring-jms, we need to configure some Spring classes to enable them to work with WildFly's transaction manager and connection factory. The following code example demonstrates how to use CDI to configure a transactional JMS Camel route.

The camel-jms component requires a transaction manager of type `org.springframework.transaction.PlatformTransactionManager`. Therefore, we begin by creating a bean extending `JtaTransactionManager`. Note that the bean is annotated with `@Named` to allow the bean to be registered within the Camel bean registry. Also note that the WildFly transaction manager and user transaction instances are injected using CDI.

```
@Named("transactionManager")
public class CdiTransactionManager extends JtaTransactionManager {

  @Resource(mappedName = "java:/TransactionManager")
  private TransactionManager transactionManager;

  @Resource
  private UserTransaction userTransaction;

  @PostConstruct
  public void initTransactionManager() {
    setTransactionManager(transactionManager);
    setUserTransaction(userTransaction);
  }
}
```

Next we need to declare the transaction policy that we want to use. Again we use the `@Named` annotation to make the bean available to Camel. The transaction manager is also injected so that a TransactionTemplate can be created with the desired transaction policy. PROPAGATION_REQUIRED in this instance.

```
@Named("PROPAGATION_REQUIRED")
public class CdiRequiredPolicy extends SpringTransactionPolicy {
  @Inject
  public CdiRequiredPolicy(CdiTransactionManager cdiTransactionManager) {
    super(new TransactionTemplate(cdiTransactionManager,
      new DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_REQUIRED)));
  }
}
```

Now we can configure our Camel RouteBuilder class and inject the dependencies needed for the Camel JMS component. The WildFly XA connection factory is injected together with the transaction manager we configured earlier.

In this example RouteBuilder, whenever any messages are consumed from queue1, they are routed to another JMS queue named queue2. Messages consumed from queue2 result in JMS transaction being rolled back using the rollback() DSL method. This results in the original message being placed onto the dead letter queue(DLQ).

```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JMSRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:/JmsXA")
  private ConnectionFactory connectionFactory;

  @Inject
  CdiTransactionManager transactionManager;

  @Override
  public void configure() throws Exception {
    // Creates a JMS component which supports transactions
    JmsComponent jmsComponent = JmsComponent.jmsComponentTransacted(connectionFactory, tr
    getContext().addComponent("jms", jmsComponent);

    from("jms:queue:queue1")
      .transacted("PROPAGATION_REQUIRED")
      .to("jms:queue:queue2");

    // Force the transaction to roll back. The message will end up on the WildFly 'DLQ' m
    from("jms:queue:queue2")
      .to("log:end")
      .rollback();
  }
```

Remote JMS destinations

It's possible for one WildFly instance to send messages to HornetQ destinations configured on another WildFly instance through remote JNDI.

Some additional WildFly configuration is required to achieve this. First an exported JMS queue is configured.

Only JNDI names bound in the `java:jboss/exported` namespace are considered as candidates for remote clients, so the queue is named appropriately.

Note that the queue must be configured on the WildFly client application server **and** the WildFly remote server.

```
<jms-queue name="RemoteQueue">
  <entry name="java:jboss/exported/jms/queues/RemoteQueue"/>
</jms-queue>
```

Before the client can connect to the remote server, user access credentials need to be configured. On the remote server run the add user utility to create a new application user within the 'guest' group. This example has a user with the name 'admin' and a password of 'secret'.

The RouteBuilder implementation is different to the previous examples. Instead of injecting the connection factory, we need to configure an InitalContext and retrieve it from JNDI ourselves.

The `configureInitialContext` method creates this InitialContext. Notice that we need to set a provider URL which should reference your remote WildFly instance host name and port number. This example uses the WildFly JMS http-connector, but there are alternatives documented here.

Finally the route is configured to send an XML payload every 10 seconds to the remote destination configured earlier - 'RemoteQueue'.

```
@Override
public void configure() throws Exception {
  Context initialContext = configureInitialContext();
  ConnectionFactory connectionFactory = (ConnectionFactory) initialContext.lookup("java:j

  JmsComponent component = new JmsComponent();
  component.setConnectionFactory(connectionFactory);

  getContext().addComponent("jms", component);

  from("timer://foo?fixedRate=true&period=10000")
  .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting></mess
  .to("jms:queue:RemoteQueue?username=admin&password=secret")
  .to("log:jms?showAll=true");
}

private Context configureInitialContext() throws NamingException {
  final Properties env = new Properties();
  env.put(Context.INITIAL_CONTEXT_FACTORY, "org.jboss.naming.remote.client.InitialContext
  env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL, "http-remoting:/
  env.put(Context.SECURITY_PRINCIPAL, System.getProperty("username", "admin"));
  env.put(Context.SECURITY_CREDENTIALS, System.getProperty("password", "secret"));
  return new InitialContext(env);
}
```

Security

Refer to the JMS security section.

Code examples on GitHub

An example camel-jms application is available on GitHub.

Integration with JMX

Management support is provided through the camel-jmx component which integrates with the WildFly JMX subsystem.

```
CamelContext camelctx = contextFactory.createWildflyCamelContext(getClass().getClassLoade
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        String host = InetAddress.getLocalHost().getHostName();
        from("jmx:platform?format=raw&objectDomain=org.apache.camel&key.context=" + host
        "&monitorType=counter&observedAttribute=ExchangesTotal&granularityPeriod=500").
        to("direct:end");
    }
});
camelctx.start();

ConsumerTemplate consumer = camelctx.createConsumerTemplate();
MonitorNotification notifcation = consumer.receiveBody("direct:end", MonitorNotification.
Assert.assertEquals("ExchangesTotal", notifcation.getObservedAttribute());
```

Integration with JNDI

JNDI integration is provided by a WildFly specific CamelContext, which is can be obtained like this

```
InitialContext inictx = new InitialContext();
CamelContextFactory factory = inictx.lookup("java:jboss/camel/CamelContextFactory");
WildFlyCamelContext camelctx = factory.createCamelContext();
```

From a `WildFlyCamelContext` you can obtain a preconfigured Naming Context

```
Context context = camelctx.getNamingContext();
context.bind("helloBean", new HelloBean());
```

which can then be referenced from Camel routes.

```
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").beanRef("helloBean");
    }
});
camelctx.start();
```

Integration with JPA

JPA integration is provided by the Camel JPA component. Camel JPA applications are developed as per normal by providing a persistence.xml configuration file together with some JPA annotated classes.

Example persistence.xml

In this example we use the Wildfly in-memory ExampleDS datasource which is configured within the Wildfly standalone.xml configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/
             xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="camel">
        <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
        <class>org.wildfly.camel.test.jpa.model.Customer</class>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Example JPA entitiy

```
@Entity
@Table(name = "customer")
public class Customer implements Serializable {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Long getId() {
        return id;
    }

    public void setId(final Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Camel JPA endpoint / route configuration

Having configured JPA, you can make use of CDI to inject an EntityManager and UserTransaction instance into your RouteBuilder class or test case:

```
@PersistenceContext
EntityManager em;

@Inject
UserTransaction userTransaction;
```

Now to configure the Camel routes and JPA endpoint:

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext(getClass().getClassLoade

EntityManagerFactory entityManagerFactory = em.getEntityManagerFactory();

// Configure a transaction manager
JtaTransactionManager transactionManager = new JtaTransactionManager();
transactionManager.setUserTransaction(userTransaction);
transactionManager.afterPropertiesSet();

// Configure the JPA endpoint to use the correct EntityManagerFactory and JtaTransactionM
final JpaEndpoint jpaEndpoint = new JpaEndpoint();
jpaEndpoint.setCamelContext(camelctx);
jpaEndpoint.setEntityType(Customer.class);
jpaEndpoint.setEntityManagerFactory(entityManagerFactory);
jpaEndpoint.setTransactionManager(transactionManager);

camelctx.addRoutes(new RouteBuilder() {
@Override
public void configure() throws Exception {
    from("direct:start")
    .to(jpaEndpoint);
}
});

camelctx.start();
```

Finally, we can send a 'Customer' entity to the 'direct:start' endpoint and then query the ExampleDS datasource to verify that a record was saved.

```
Customer customer = new Customer("John", "Doe");
ProducerTemplate producer = camelctx.createProducerTemplate();
producer.sendBody("direct:start", customer);

// Query the in memory database customer table to verify that a record was saved
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<Long> query = criteriaBuilder.createQuery(Long.class);
query.select(criteriaBuilder.count(query.from(Customer.class)));

long recordCount = em.createQuery(query).getSingleResult();

Assert.assertEquals(1L, recordCount);
```

Camel Components

This chapter details information about supported camel components

Supported Components

The following lists the set of supported Camel components that are available to JavaEE deployments:

- camel-activemq
- camel-atom
- camel-bindy
- camel-cdi
- camel-cxf
- camel-dozer
- camel-ejb
- camel-file2
- camel-ftp
- camel-hl7
- camel-http4
- camel-jaxb
- camel-jms
- camel-jmx
- camel-jpa
- camel-lucene
- camel-mail
- camel-mina2
- camel-mqtt
- camel-mvel
- camel-netty4
- camel-ognl
- camel-quartz2
- camel-rest
- camel-restlet
- camel-rss
- camel-saxon
- camel-script
- camel-servlet
- camel-sql
- camel-swagger
- camel-velocity
- camel-weather
- camel-xstream

Adding Components

Adding support for additional Camel Components is easy

Add your modules.xml definition

A modules.xml descriptor defines the class loading behavior for your component. It should be placed together with the component's jar in `modules/system/layers/fuse/org/apache/camel/component`. Module dependencies should be setup for direct compile time dependencies.

Here is an example for the camel-ftp component

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.camel.component.ftp">
  <resources>
    <resource-root path="camel-ftp-2.14.0.jar" />
  </resources>
  <dependencies>
    <module name="com.jcraft.jsch" />
    <module name="javax.xml.bind.api" />
    <module name="org.apache.camel.core" />
    <module name="org.apache.commons.net" />
  </dependencies>
</module>
```

Please make sure you don't duplicate modules that are already available in WildFly and can be reused.

Add a reference to the component

To make this module visible by default to arbitrary JavaEE deployments add a reference to `modules/system/layers/fuse/org/apache/camel/component/main/module.xml`

```
<module xmlns="urn:jboss:module:1.3" name="org.apache.camel.component">

  <dependencies>
    ...
    <module name="org.apache.camel.component.ftp" export="true" services="export"/>
  </dependencies>

</module>
```

camel-activemq

Camel ActiveMQ integration is provided by the camel-activemq component.

The component can be configured to work with an embedded or external broker. For Wildfly / EAP container managed connection pools and XA-Transaction support, the ActiveMQ Resource Adapter can be configured into the container configuration file.

WildFly ActiveMQ resource adapter configuration

An ActiveMQ WildFly module and the ActiveMQ resource adapter rar file is provided as part of the WildFly Camel subsystem distribution. Therefore there is no need to download and manually configure all of these components yourself.

The following steps outline how to configure the ActiveMQ resource adapter.

1) Make sure your running WildFly instance is stopped. Open a terminal session and change into the WildFly installation root directory

2) Change into the ActiveMQ module directory

`cd modules/system/layers/fuse/org/apache/activemq/main`

3) Extract broker-config and ra.xml files from the resource adapter.

```
unzip activemq-rar-5.11.1.rar broker-config.xml
unzip activemq-rar-5.11.1.rar META-INF/ra.xml
```

4) Modify module.xml and add an additional `<resource-root>` value to the top of the `<resources>` section for `<resource-root path="." />`.

After making the modification, the `<resources>` section should look like this:

```
<resources>
  <resource-root path="." />
  <resource-root path="activemq-broker-5.11.1.jar" />
  <resource-root path="activemq-client-5.11.1.jar" />
  <resource-root path="activemq-jms-pool-5.11.1.jar" />
  <resource-root path="activemq-kahadb-store-5.11.1.jar" />
  <resource-root path="activemq-openwire-legacy-5.11.1.jar" />
  <resource-root path="activemq-pool-5.11.1.jar" />
  <resource-root path="activemq-protobuf-1.1.jar" />
  <resource-root path="activemq-ra-5.11.1.jar" />
  <resource-root path="activemq-rar-5.11.1.rar" />
  <resource-root path="activemq-spring-5.11.1.jar" />
</resources>
```

5) Modify broker-config and META-INF/ra.xml as per your requirements

6) Configure the WildFly resource adapters subsystem for the ActiveMQ adapter. Instructions for doing this can be found in section 'Setup WildFly standalone configuration' in the guide - How to Use Out of Process ActiveMQ with WildFly

7) Start WildFly. If everything is configured correctly, you should see a message within the WildFly server.log like.

```
13:16:08,412 INFO [org.jboss.as.connector.deployment] (MSC service thread 1-5) JBAS010406:
Registered connection factory java:/AMQConnectionFactory
```

Camel route configuration

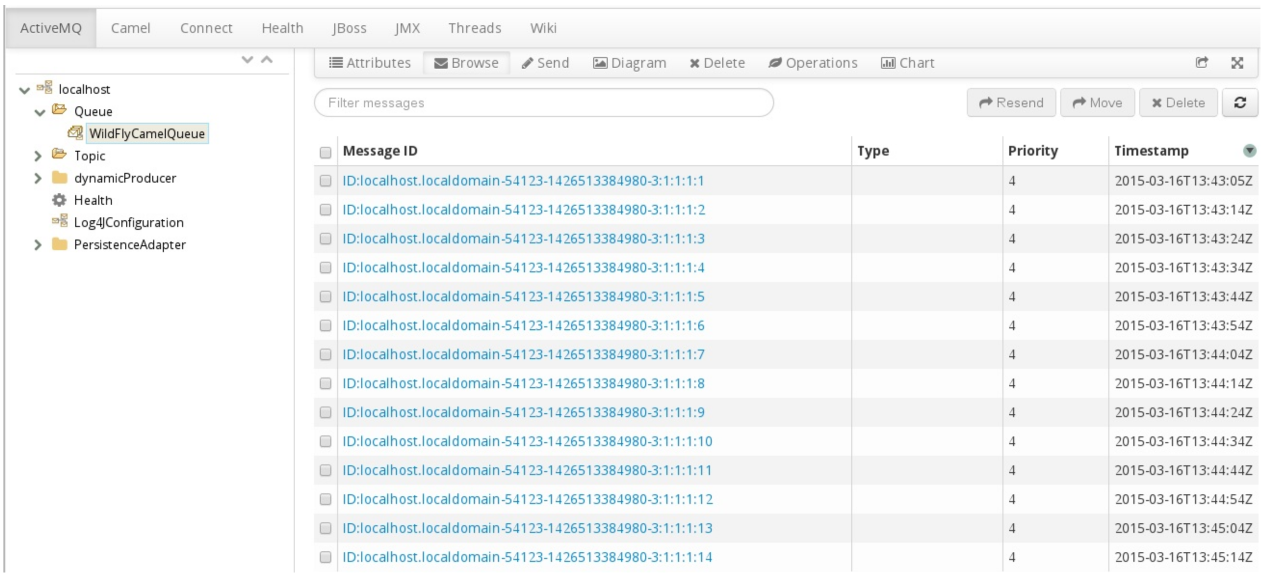
The following ActiveMQ producer and consumer examples make use of the ActiveMQ embedded broker and the 'vm' transport (thus avoiding the need for an external ActiveMQ broker).

The examples use CDI in conjunction with the camel-cdi component. JMS ConnectionFactory instances are injected into the Camel RouteBuilder through JNDI lookups.

ActiveMQ Producer

```
@Startup
@ApplicationScoped
@ContextName("activemq-camel-context")
public class ActiveMQRouteBuilder extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("timer://sendJMSMessage?fixedRate=true&period=10000")
    .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting></me
    .to("activemq:queue:WildFlyCamelQueue?brokerURL=vm://localhost")
    .log("JMS Message sent");
  }
}
```

A log message will be output to the console each time a message is added to the WildFlyCamelQueue destination. To verify that the messages really are being placed onto the queue, we can use the Hawtio console provided by the WildFly Camel subsystem.



Message ID	Type	Priority	Timestamp
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:1		4	2015-03-16T13:43:05Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:2		4	2015-03-16T13:43:14Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:3		4	2015-03-16T13:43:24Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:4		4	2015-03-16T13:43:34Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:5		4	2015-03-16T13:43:44Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:6		4	2015-03-16T13:43:54Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:7		4	2015-03-16T13:44:04Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:8		4	2015-03-16T13:44:14Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:9		4	2015-03-16T13:44:24Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:10		4	2015-03-16T13:44:34Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:11		4	2015-03-16T13:44:44Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:12		4	2015-03-16T13:44:54Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:13		4	2015-03-16T13:45:04Z
ID:localhost.localdomain-54123-1426513384980-3:1:1:1:14		4	2015-03-16T13:45:14Z

ActiveMQ Consumer

To consume ActiveMQ messages the Camel RouteBuilder implementation is similar to the producer example.

When the ActiveMQ endpoint consumes messages from the WildFlyCamelQueue destination, the content is logged to the console.

```
@Override
public void configure() throws Exception {
  from("activemq:queue:WildFlyCamelQueue?brokerURL=vm://localhost")
  .to("log:jms?showAll=true");
}
```

ActiveMQ Transactions

ActiveMQ Resource Adapter Configuration

The ActiveMQ resource adapter is required as we will want to leverage XA transaction support, connection pooling etc.

The XML snippet below shows how the resource adapter is configured within the WildFly server XML configuration. Notice that the `ServerURL` is set to use an embedded broker. The connection factory is bound to the JNDI name 'java:/ActiveMQConnectionFactory'. This will be looked up in the RouteBuilder example that follows.

Finally, two queues are configured named 'queue1' and 'queue2'.

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:2.0">
  <resource-adapters>
    <resource-adapter id="activemq-rar.rar">
      <module slot="main" id="org.apache.activemq" />
      <transaction-support>XATransaction</transaction-support>
      <config-property name="ServerUrl">
```

```
            vm://localhost?jms.rmIdFromConnectionId=true
        </config-property>
        <connection-definitions>
          <connection-definition class-name="org.apache.activemq.ra.ActiveMQManagedConnecti
            <xa-pool>
              <min-pool-size>1</min-pool-size>
              <max-pool-size>20</max-pool-size>
              <prefill>false</prefill>
              <is-same-rm-override>false</is-same-rm-override>
              </xa-pool>
            </connection-definition>
        </connection-definitions>
        <admin-objects>
          <admin-object class-name="org.apache.activemq.command.ActiveMQQueue" jndi-name="j
            <config-property name="PhysicalName">queue1</config-property>
          </admin-object>
          <admin-object class-name="org.apache.activemq.command.ActiveMQQueue" jndi-name="j
            <config-property name="PhysicalName">queue2</config-property>
          </admin-object>
        </admin-objects>
      </resource-adapter>
    </resource-adapters>
  </subsystem>
```

Transaction Manager

The camel-active component requires a transaction manager of type `org.springframework.transaction.PlatformTransactionManager`. Therefore, we begin by creating a bean extending `JtaTransactionManager`. Note that the bean is annotated with `@Named` to allow the bean to be registered within the Camel bean registry. Also note that the WildFly transaction manager and user transaction instances are injected using CDI.

```
@Named("transactionManager")
public class CdiTransactionManager extends JtaTransactionManager {

  @Resource(mappedName = "java:/TransactionManager")
  private TransactionManager transactionManager;

  @Resource
  private UserTransaction userTransaction;

  @PostConstruct
  public void initTransactionManager() {
    setTransactionManager(transactionManager);
    setUserTransaction(userTransaction);
  }
}
```

Transaction Policy

Next we need to declare the transaction policy that we want to use. Again we use the `@Named` annotation to make the bean available to Camel. The transaction manager is also injected so that a

TransactionTemplate can be created with the desired transaction policy. PROPAGATION_REQUIRED in this instance.

```
@Named("PROPAGATION_REQUIRED")
public class CdiRequiredPolicy extends SpringTransactionPolicy {
  @Inject
  public CdiRequiredPolicy(CdiTransactionManager cdiTransactionManager) {
    super(new TransactionTemplate(cdiTransactionManager,
      new DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_REQUIRED)));
  }
}
```

Route Builder

Now we can configure our Camel RouteBuilder class and inject the dependencies needed for the Camel ActiveMQ component. The ActiveMQ connection factory that we configured on the resource adapter configutation is injected together with the transaction manager we configured earlier.

In this example RouteBuilder, whenever any messages are consumed from queue1, they are routed to another JMS queue named queue2. Messages consumed from queue2 result in JMS transaction being rolled back using the rollback() DSL method. This results in the original message being placed onto the dead letter queue(DLQ).

```
@Startup
@ApplicationScoped
@ContextName("activemq-camel-context")
public class ActiveMQRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:/ActiveMQConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Inject
  private CdiTransactionManager transactionManager;

  @Override
  public void configure() throws Exception {
    ActiveMQComponent activeMQComponent = ActiveMQComponent.activeMQComponent();
    activeMQComponent.setTransacted(false);
    activeMQComponent.setConnectionFactory(connectionFactory);
    activeMQComponent.setTransactionManager(transactionManager);

    getContext().addComponent("activemq", activeMQComponent);

      errorHandler(deadLetterChannel("activemq:queue:ActiveMQ.DLQ")
      .useOriginalMessage()
      .maximumRedeliveries(0)
      .redeliveryDelay(1000));

    from("activemq:queue:queue1")
      .transacted("PROPAGATION_REQUIRED")
      .to("activemq:queue:queue2");

    from("activemq:queue:queue2")
      .to("log:end")
```

```
        .rollback();
    }
  }
```

Security

Refer to the JMS security section.

Code examples on GitHub

An example camel-activemq application is available on GitHub.

camel-atom

Atom feed consumption in Camel is provided by the camel-atom component.

The following example configures an Atom endpoint to consume the recent activity GitHub feed of user 'wildflyext'. The raw content of each feed entry is then written out as a file within directory 'feed/entries'.

```
CamelContext camelContext = new DefaultCamelContext();

camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("atom://https://github.com/wildflyext.atom?splitEntries=true")
        .process(new Processor() {
            @Override
            public void process(final Exchange exchange) throws Exception {
                Entry entry = exchange.getIn().getBody(Entry.class);
                exchange.getOut().setBody(entry.getContent());
            }
        })
        .to("file:///feed/entries/");
    }
});
```

camel-bindy

The goal of camel-bindy is to allow the parsing/binding of non-structured data to/from Java Beans that have binding mappings defined with annotations.

Here we have a annotated domain model class

```
@CsvRecord(separator = ",")
public class Customer {

    @DataField(pos = 1)
    private String firstName;

    @DataField(pos = 2)
    private String lastName;

    ...
}
```

We can use the `BindyCsvDataFormat` data format unmarshall CSV data like `John,Doe` to the domain model.

```
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .unmarshal(new BindyCsvDataFormat(Customer.class))
        .to("mock:result");
    }
});
camelctx.start();
```

camel-cdi

Covered by JavaEE CDI .

camel-cxf

Covered by JavaEE JAXWS .

camel-dozer

The camel-dozer component provides the ability to map between Java beans using the Dozer mapping framework. Camel also supports the ability to trigger Dozer mappings as a type converter.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").convertBodyTo(CustomerB.class);
    }
});

DozerBeanMapperConfiguration mconfig = new DozerBeanMapperConfiguration();
mconfig.setMappingFiles(Arrays.asList(new String[] { "mappings.xml" }));
new DozerTypeConverterLoader(camelctx, mconfig);
```

camel-ejb

The camel-ejb component binds EJBs to Camel message exchanges.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").to("ejb:java:module/HelloBean");
    }
});
```

camel-file

The camel-file component provides access to file systems, allowing files to be processed by any other Camel Components or messages from other components to be saved to disk.

Here is a simple route that prepends the message with 'Hello' and writes the result to a file in WildFly's data directory.

```
final String datadir = System.getProperty("jboss.server.data.dir");

CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").transform(body().prepend("Hello ")).
        to("file:" + datadir + "?fileName=camel-file.txt");
    }
});
```

camel-ftp

The camel-ftp component provides access to remote file systems over the FTP and SFTP protocols.

```
CamelContext camelctx = new DefaultCamelContext();
Endpoint endpoint = camelctx.getEndpoint("ftp://localhost:21000/foo?username=admin&passwo

camelctx.createProducerTemplate().sendBodyAndHeader(endpoint, "Hello", "CamelFileName", "
```

camel-http

The camel-http component provides HTTP based endpoints for calling external HTTP resources.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .to("http4://somehost:8080/simple/myservlet");
    }
});
```

camel-hl7

The camel-hl7 component is used for working with the HL7 MLLP protocol and HL7 v2 messages using the HAPI library.

```
final String msg = "MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|12
final HL7DataFormat format = new HL7DataFormat();

CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .marshal(format)
        .unmarshal(format)
        .to("mock:result");
    }
});
camelctx.start();

HapiContext context = new DefaultHapiContext();
Parser p = context.getGenericParser();
Message hapimsg = p.parse(msg);

ProducerTemplate producer = camelctx.createProducerTemplate();
Message result = (Message) producer.requestBody("direct:start", hapimsg);
Assert.assertEquals(hapimsg.toString(), result.toString());
```

camel-jaxb

Covered by JavaEE JAXB .

camel-jms

Covered by JavaEE JMS .

camel-jmx

Covered by JavaEE JMX .

camel-jpa

Covered by JavaEE JPA .

camel-lucene

The camel-lucene component is based on the Apache Lucene project. Apache Lucene is a powerful high-performance, full-featured text search engine library written entirely in Java.

camel-mail

Interaction with email is provided by the camel-mail component.

By default, Camel will create its own mail session and use this to interact with your mail server. Since WildFly already provides a mail subsystem with all of the relevant support for secure connections, username / password encryption etc, it is recommended to configure your mail sessions within the WildFly configuration and use JNDI to wire them into your Camel endpoints.

WildFly configuration

First we configure the WildFly mail subsystem for our Mail server. This example adds configuration for Google Mail IMAP and SMTP .

An additional mail-session is configured after the 'default' session.

```
<subsystem xmlns="urn:jboss:domain:mail:2.0">
    <mail-session name="default" jndi-name="java:jboss/mail/Default">
      <smtp-server outbound-socket-binding-ref="mail-smtp"/>
    </mail-session>

    <mail-session debug="true" name="gmail" jndi-name="java:jboss/mail/gmail">
      <smtp-server outbound-socket-binding-ref="mail-gmail-smtp" ssl="true" username="you
      <imap-server outbound-socket-binding-ref="mail-gmail-imap" ssl="true" username="you
    </mail-session>
</subsystem>
```

Note that we configured `outbound-socket-binding-ref` values of 'mail-gmail-smtp' and 'mail-gmail-imap'. The next step is to configure these socket bindings. Add aditional bindings to the `socket-binding-group` configuration like the following.

```
<outbound-socket-binding name="mail-gmail-smtp">
  <remote-destination host="smtp.gmail.com" port="465"/>
</outbound-socket-binding>

<outbound-socket-binding name="mail-gmail-imap">
  <remote-destination host="imap.gmail.com" port="993"/>
</outbound-socket-binding>
```

This configures our mail session to connect to host smtp.gmail.com on port 465 and imap.gmail.com on port 993. If you're using a different mail host, then this detail will be different.

POP3 Configuration

If you need to configure POP3 sessions, the principals are the same as defined in the examples above.

```
<!-- Server configuration -->
```

```
<pop3-server outbound-socket-binding-ref="mail-pop3" ssl="true" username="your-username-h

<!-- Socket binding configuration -->
<outbound-socket-binding name="mail-gmail-imap">
  <remote-destination host="pop3.gmail.com" port="993"/>
</outbound-socket-binding>
```

Camel route configuration

Mail producer

This example uses the SMTPS protocol, together with CDI in conjunction with the camel-cdi component. The Java mail session that we configured within the WildFly configuration is injected into a Camel RouteBuilder through JNDI.

Route builder SMTPS example

The GMail mail session is injected into our RouteBuilder class using the `@Resource` annotation with a reference to the `jndi-name` attribute that we previously configured.

The `configureMailEndpoint` method takes care of some required configuration for the SMTP `MailEndpoint`. This is done so as not to duplicate the username & password details that we already defined within the WildFly configuration.

```
@Startup
@ApplicationScoped
@ContextName("mail-camel-context")
public class MailRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:jboss/mail/gmail")
  private Session mailSession;

  @Override
  public void configure() throws Exception {
    MailEndpoint mailEndpoint = (MailEndpoint) getContext().getEndpoint("smtps://smtp.gma
    configureMailEndpoint(mailEndpoint);

    from("direct:start")
      .to(mailEndpoint);
  }

  private void configureMailEndpoint(MailEndpoint endpoint) throws UnknownHostException {
    MailConfiguration configuration = endpoint.getConfiguration();

    // Wildfly seems to configure things under the SMTP / IMAP and not SMTPS / IMAPS
    String protocol = configuration.getProtocol();
    if(protocol.equals("smtps")) {
      protocol = "smtp";
    } else if(protocol.equals("imaps")) {
      protocol = "imap";
    }

    // Fetch mail session credentials from the session
```

```
      String host = mailSession.getProperty("mail." + protocol + ".host");
      String user = mailSession.getProperty("mail." + protocol + ".user");

      int port = Integer.parseInt(mailSession.getProperty("mail." + protocol + ".port"));
      InetAddress address = InetAddress.getByName(host);

      PasswordAuthentication auth = mailSession.requestPasswordAuthentication(address, port
      configuration.setPort(port);
      configuration.setUsername(auth.getUserName());
      configuration.setPassword(auth.getPassword());
    }
  }
```

To send an email we can create a ProducerTemplate and send an appropriate body together with the necessary email headers.

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("To", "destination@test.com");
headers.put("From", "sender@example.com");
headers.put("Subject", "Camel on Wildfly rocks");

String body = "Hi,\n\nCamel on Wildfly rocks!.";

ProducerTemplate template = camelContext.createProducerTemplate();
template.sendBodyAndHeaders("direct:start", body, headers);
```

Mail consumer

To receive email we use an IMAP MailEndpoint. The Camel route configuration looks like the following.

```
public void configure() throws Exception {
   MailEndpoint mailEndpoint = (MailEndpoint) getContext().getEndpoint("imaps://imap.gmai
   configureMailEndpoint(mailEndpoint);

   from(mailEndpoint)
    .to("log:email");
}
```

Security

SSL configuration

WildFly can be configured to manage Java mail sessions and their associated transports using SSL / TLS. When configuring mail sessions you can configure SSL or TLS on server types:

- smtp-server
- imap-server
- pop-server

By setting attributes `ssl="true"` or `tls="true"`.

Securing passwords

Writing passwords in clear text within configuration files is never a good idea for obvious reasons. The WildFly Vault provides tooling to mask sensitive data.

Camel security

Camel endpoint security documentation can be found on the camel-mail component guide. Camel also has a security summary page.

Code examples on GitHub

An example camel-mail application is available on GitHub for you to try out sending / receiving email.

camel-mina2

The camel-mina2 component is a transport for working with Apache MINA

camel-mqtt

The camel-mqtt component is used for communicating with MQTT compliant message brokers, like Apache ActiveMQ or Mosquitto

camel-mvel

The camel-mvel component allows you to process a message using an MVEL template.

A simple template

```
Hello @{request.body}
```

can be used like this

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").to("mvel:template.mvel");
    }
});
camelctx.start();
ProducerTemplate producer = camelctx.createProducerTemplate();
String result = producer.requestBody("direct:start", "Kermit", String.class);
Assert.assertEquals("Hello Kermit", result);
```

camel-netty4

Netty client / server support in Camel is provided by the camel-netty4 component.

WildFly 8 and EAP 6.4 are bundled with module libraries supporting the Netty project version 4. Therefore, the standard camel-netty component will not work since it is compatible with Netty version 3 only.

Simple Netty Client / Server Test

```
CamelContext camelContext = new DefaultCamelContext();

camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("netty4:tcp://localhost:7666?textline=true")
                .transform(simple("Hello ${body}"))
                .to("direct:end");
    }
});

camelContext.start();

PollingConsumer pollingConsumer = camelContext.getEndpoint("direct:end").createPollingCon
pollingConsumer.start();

Socket socket = new Socket("localhost", 7666);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

try {
    out.write("Kermit\n");
} finally {
    out.close();
    socket.close();
}

String result = pollingConsumer.receive().getIn().getBody(String.class);

Assert.assertEquals("Hello Kermit", result);

camelContext.stop();
```

camel-ognl

OGNL expression support in Camel is provided by the camel-ognl component.

Simple Camel OGNL Use Case

```
CamelContext camelContext = new DefaultCamelContext();

camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
            .choice()
                .when()
                    .ognl("request.body.name == 'Kermit'").transform(simple("Hello ${body
                .otherwise()
                    .to("mock:dlq");
    }
});

camelContext.start();

Person person = new Person();
person.setName("Kermit");

ProducerTemplate producer = camelContext.createProducerTemplate();
String result = producer.requestBody("direct:start", person, String.class);

Assert.assertEquals("Hello Kermit", result);

camelContext.stop();
```

camel-quartz

The camel-quartz component provides a scheduled delivery of messages using the Quartz Scheduler 2.x.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        from("quartz2://mytimer?trigger.repeatCount=3&trigger.repeatInterval=100")
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                latch.countDown();
            }})
        .to("mock:result");
    }
});
```

camel-rest

The camel-rest component allows you to define REST endpoints using the Rest DSL and plugin to other Camel components as the REST transport.

> **The WildFly Camel Subsystem only supports camel-servlet for use with the REST DSL. Attempts to configure other components will not work**

```
    CamelContext camelctx = new DefaultCamelContext();
    camelctx.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            restConfiguration().component("servlet").contextPath("camel/rest").port(8080)
            rest("/hello").get("/{name}").to("direct:hello");
            from("direct:hello").transform(simple("Hello ${header.name}"));
        }
    });
```

camel-rss

The camel-rss component is used for polling RSS feeds.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        from("rss://https://developer.jboss.org/blogs/feeds/posts?splitEntries=true&c
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                latch.countDown();
            }})
        .to("mock:result");
    }
});
```

camel-saxon

The camel-saxon component supports XQuery to allow an Expression or Predicate to be used in the DSL or Xml Configuration.

```
    CamelContext camelctx = new DefaultCamelContext();
    camelctx.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            Namespaces ns = new Namespaces("ns", "http://org/wildfly/test/jaxb/model/Cust
            from("direct:start").transform().xquery("/ns:customer/ns:firstName", String.c
            .to("mock:result");
        }
    });
```

camel-script

The camel-script component supports a number of scripting languages which can be used to create an Expression or Predicate via the standard JSR 223

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").choice()
        .when(script("beanshell", "request.getHeaders().get(\"foo\").equals(\"bar\")"
        .otherwise().transform(body().append(" unmatched")).to("mock:unmatched");
    }
});
```

camel-servlet

The camel-servlet component provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint that is bound to a published Servlet.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("servlet://hello?servletName=CamelServletTest&matchOnUriPrefix=true")
        .process(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                exchange.getOut().setBody("Hello Kermit");
            }
        });
    }
});
```

camel-sql

The camel-sql component allows you to work with databases using JDBC queries. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("sql:select name from information_schema.users?dataSource=java:jboss/dat
        .to("direct:end");
    }
});
```

camel-weather

The camel-weather component provides integration with the Open Weather Map API.

As an example, we can consume the current weather for Madrid in Spain and make some decisions based upon the humidity percentage:

```
CamelContext camelContext = new DefaultCamelContext();

camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("weather:foo?location=Madrid,Spain")
        .choice()
          .when().jsonpath("$..[?(@humidity > 90)]")
            .to("direct:veryhumid")
          .when().jsonpath("$..[?(@humidity > 70)]")
            .to("direct:humid")
          .otherwise()
            .to("direct:nothumid");
    }
});

camelContext.start();
```

camel-xstream

The camel-xstream component provides the XStream Data Format which uses the XStream library to marshal and unmarshal Java objects to and from XML.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .marshal().xstream();
    }
});

camelctx.start();
try {
    ProducerTemplate producer = camelctx.createProducerTemplate();
    String customer = producer.requestBody("direct:start", new Customer("John", "Doe"), S
} finally {
    camelctx.stop();
}
```

Data Formats

The following lists supported Data Formats

- Bindy
- Castor
- Crypto
- CSV
- Flatpack
- GZip
- HL7
- JSON
- JAXB
- Protobuf
- SOAP
- Serialization
- TidyMarkup
- XmlBeans
- XMLSecurity
- XStream
- Zip

Languages

The following lists supported scripting languages

- BeanShell
- Groovy
- Ruby
- Python
- JavaScript

Cloud Integration

This chapter details information about cloud integration

- Docker
- OpenShift
- Beanstalk

Docker Integration

This chapter gets you started on WildFly-Camel on Docker

Installing Docker

SSH into your instance and type

```
$ sudo yum install -y docker
$ sudo service docker start
```

to install and start Docker.

Giving non-root access

The docker daemon always runs as the root user, and the docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user root, and so, by default, you can access it with sudo.

If you have a Unix group called `docker` and add users to it, then the docker daemon will make the ownership of the Unix socket read/writable by the `docker` group when the daemon starts. The docker daemon must always run as the root user, but if you run the docker client as a user in the `docker` group then you don't need to add sudo to all the client commands.

```
$ sudo usermod -a -G docker ec2-user
$ sudo service docker restart
```

> You may have to logout/login for this change to take effect

In my case I also had to add a host mapping like this

```
[ec2-user@ip-172-30-0-233 ~]$ sudo cat /etc/hosts
127.0.0.1       localhost localhost.localdomain
172.30.0.233    ip-172-30-0-233
```

and make the docker daemon bind to a number of sockets

```
[ec2-user@ip-172-30-0-217 ~]$ cat /etc/sysconfig/docker
# Additional startup options for the Docker daemon
OPTIONS="-H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock"
```

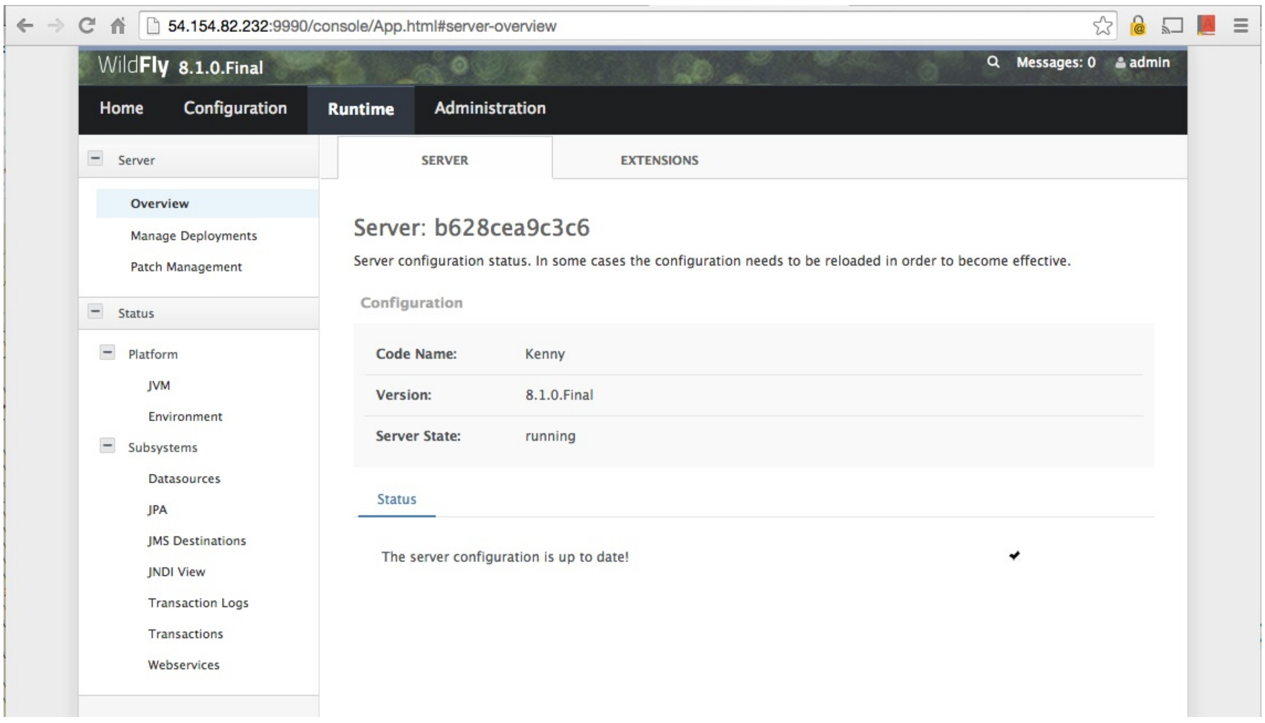
Standalone Server

With every WildFly-Camel release we also publish the latest wildflyext/wildfly-camel image.

You can run the standalone container like this

```
$ docker run --rm -ti -e WILDFLY_MANAGEMENT_USER=admin -e WILDFLY_MANAGEMENT_PASSWORD=adm
```

and access the admin console like this: http://54.154.82.232:9990/console



Server: b628cea9c3c6

Server configuration status. In some cases the configuration needs to be reloaded in order to become effective.

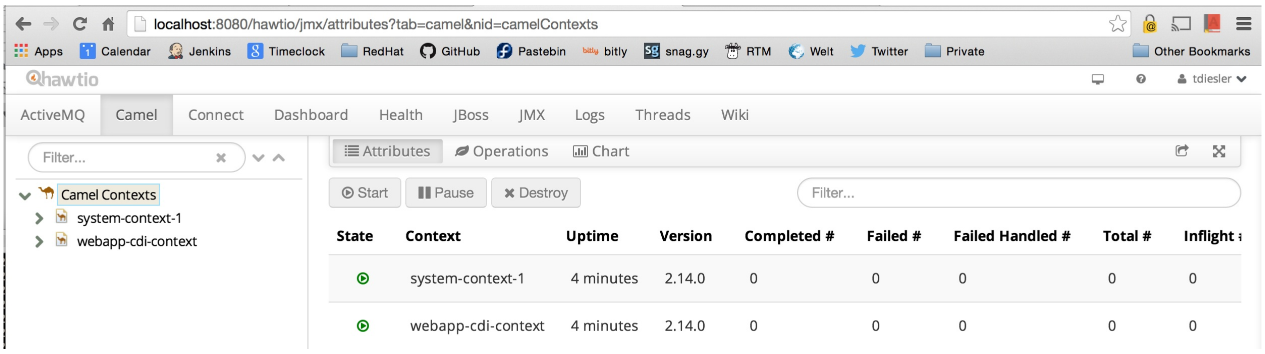
Configuration

Code Name:	Kenny
Version:	8.1.0.Final
Server State:	running

Status

The server configuration is up to date! ✔

The Hawt.io console is available at: http://54.154.82.232:8080/hawtio



State	Context	Uptime	Version	Completed #	Failed #	Failed Handled #	Total #	Inflight
●	system-context-1	4 minutes	2.14.0	0	0	0	0	0
●	webapp-cdi-context	4 minutes	2.14.0	0	0	0	0	0

Domain Setup

Running multiple server containers in a cloud environment is often only useful when these containers can also be managed. Without the management interfaces exposed it would be virtually impossible to adjust configurations for these individual servers or the whole set. As a minimum we would like to monitor the health state of these servers so that we can possibly replace containers if needed.

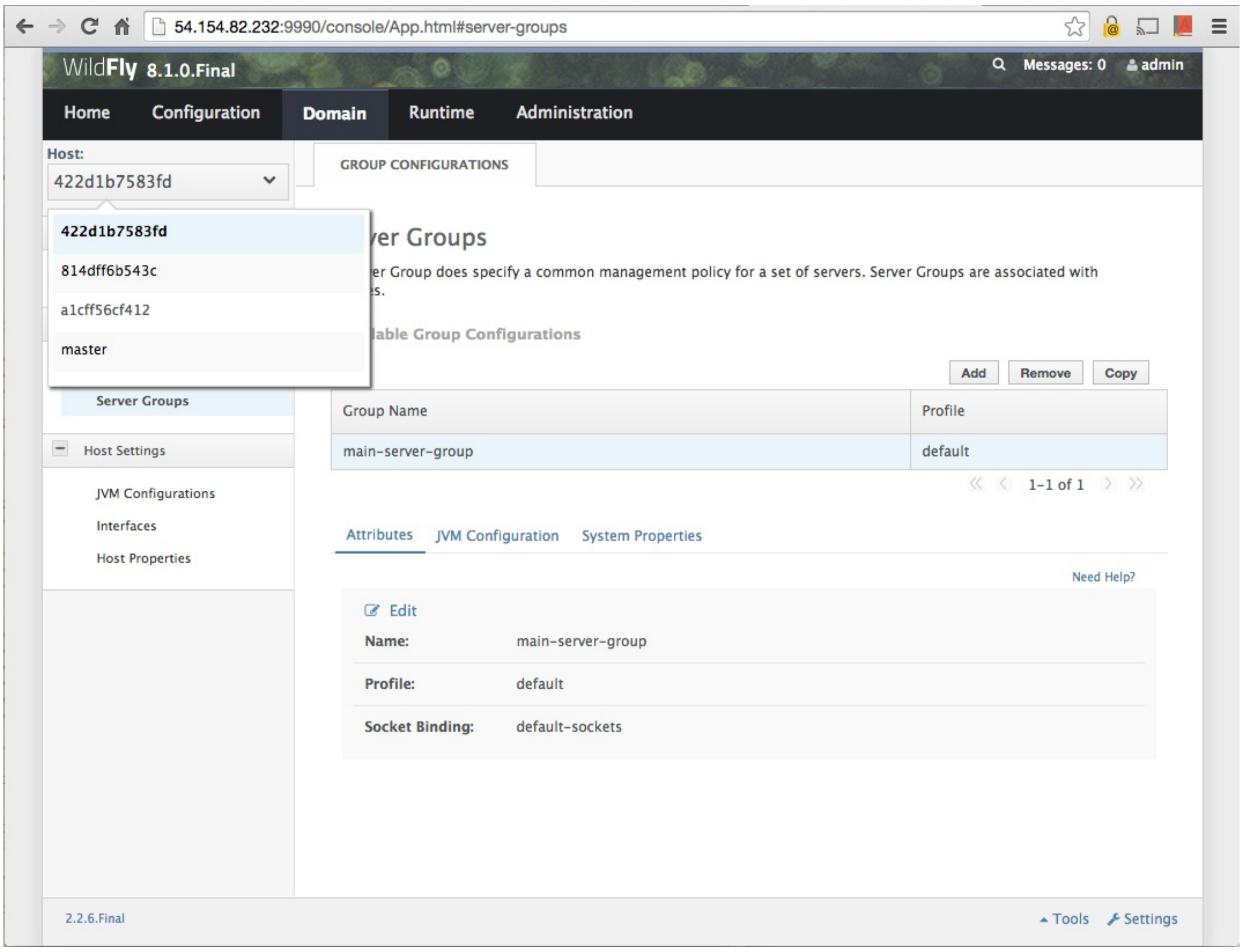
You can run the container that acts as a domain controller like this

```
$ docker run --rm -ti -e WILDFLY_MANAGEMENT_USER=admin -e WILDFLY_MANAGEMENT_PASSWORD=adm
```

and various hosts that connect to the domain controller as daemons like this

```
$ docker run -d -e WILDFLY_MANAGEMENT_USER=admin -e WILDFLY_MANAGEMENT_PASSWORD=admin -p
```

As above, you can access the admin console like this: http://54.154.82.232:9990/console



ver Groups

er Group does specify a common management policy for a set of servers. Server Groups are associated with s.

lable Group Configurations

Group Name	Profile
main-server-group	default

Name: main-server-group

Profile: default

Socket Binding: default-sockets

Now we have a domain controller running that is reachable on `9990` and three hosts that expose their respective `8080` port on various public network ports.

```
$ docker ps
CONTAINER ID        IMAGE                                 COMMAND                CREATED
eac043e97abc        wildflyext/wildfly-camel:latest       "/opt/jboss/wildfly/   4 seconds ag
27a41d2b8a4e        wildflyext/wildfly-camel:latest       "/opt/jboss/wildfly/   5 seconds ag
6d47cae3be7c        wildflyext/wildfly-camel:latest       "/opt/jboss/wildfly/   6 seconds ag
6095cf80d3a8        wildflyext/wildfly-camel:latest       "/opt/jboss/wildfly/   15 seconds a
```

We retained WildFly manageability in a Docker environment.

The next level up would be a cloud management layer like Kubernetes or OpenShift.

The goal would be a setup that is highly available (HA) and scalable because containers that go bad can transparently replaced or new ones added. The public facing service should be reachable on a known address and request load should be balanced across the available hosts.

OpenShift Origin

This chapter gets you started on WildFly-Camel in OpenShift Origin

Starting OpenShift

We can start OpenShift Origin like this

```
$ mkdir /tmp/openshift
$ docker run --rm --name openshift-origin --net=host --privileged -v /var/run/docker.sock
```

Then verify the OpenShift version

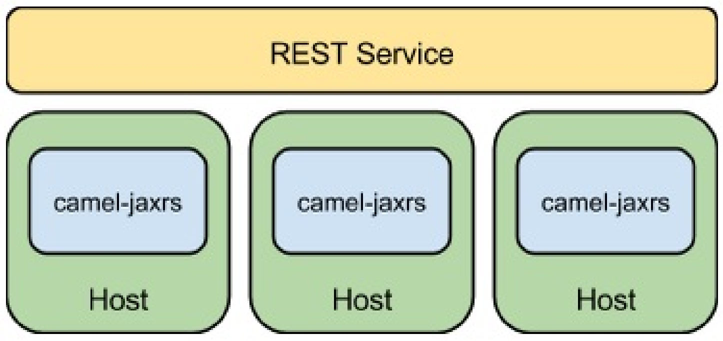
```
$ docker exec openshift-origin openshift version
openshift v0.5-22-g2b309a0
kubernetes v0.14.1-582-gb12d75d
```

We may also want to create an alias to OpenShift

```
$ alias openshift-cli="docker exec openshift-origin osc"
```

Standalone Servers

Here we run a set of WildFly Camel servers on OpenShift Origin. The target platform is Linux on an Amazon EC2 instance. When done, we have a set of portable Docker containers that can be deployed onto a production platform.



The example architecture consists of a set of three high available (HA) servers running REST endpoints. For server replication and failover we use Kubernetes. Each server runs in a dedicated Pod that we access via Services.

Log in to the OpenShift server

Before we start creating pods and services, we need to authenticate against the OpenShift server. We use the default 'admin' user credentials for this.

```
$ openshift-cli login -u admin -p admin
```

Create an OpenShift project

This allows us to create pods, services and replication controllers under our own custom 'wildfly-camel' namespace instead of the 'default' namespace.

```
$ openshift-cli new-project wildfly-camel
```

Running a single Pod

A simple Pod configuration for a WildFly Camel container might be defined as in wildfly-camel-step01.json

To create the Pod in OpenShift we do

```
$ openshift-cli create -f https://raw.githubusercontent.com/wildfly-extras/wildfly-camel-
```

You can see the running Pod like this

```
$ openshift-cli get pods -l name=camel

POD         IP           CONTAINER(S)   IMAGE(S)                        HOST
camel-pod   172.17.0.21  camel-cnt      wildflyext/example-camel-rest   localhost.locald
```

and delete it again with

```
$ openshift-cli delete pod -l name=camel
```

Adding a ReplicationController

To achieve high availability (HA), lets replicate this Pod using a ReplicationController as in wildfly-camel-step02.json

To create the replicated Pod in OpenShift we do

```
$ openshift-cli create -f https://raw.githubusercontent.com/wildfly-extras/wildfly-camel-
```

We now have three Pods each running an instance of our container

```
$ openshift-cli get pods

POD                      IP            CONTAINER(S)   IMAGE(S)                         HOST
rest-controller-39ywf    172.17.0.22   rest-cnt       wildflyext/example-camel-rest    loca
rest-controller-bfxg0    172.17.0.24   rest-cnt       wildflyext/example-camel-rest    loca
rest-controller-zlzfk    172.17.0.23   rest-cnt       wildflyext/example-camel-rest    loca
```

Adding a Service

The entry point into the system is a Kubernetes Service as in wildfly-camel-step03.json

To create a Service that accesses replicated Pods do

```
$ openshift-cli create -f https://raw.githubusercontent.com/wildfly-extras/wildfly-camel-
```

We now have a service

```
$ openshift-cli get services -l name=camel-srv
NAME           LABELS           SELECTOR         IP              PORT(S)
rest-service   name=camel-srv   name=camel-pod   172.30.140.33   8080/TCP
                                                 172.30.0.21
```

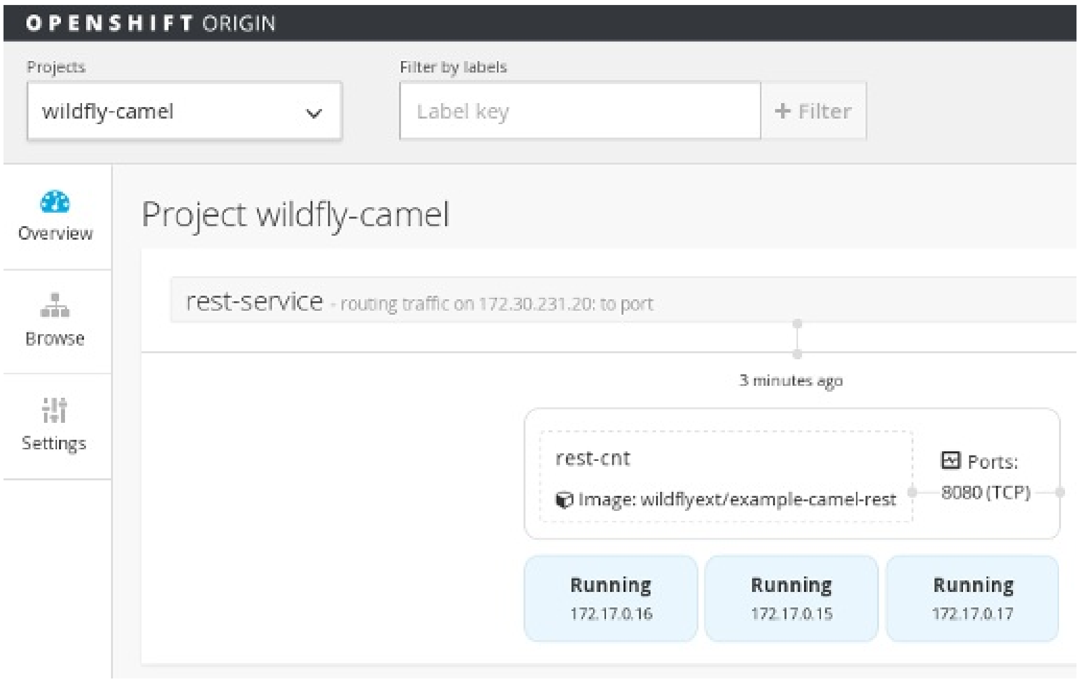
> Note, this uses a hard coded mapping in wildfly-camel-step03.json for publicIPs, which would have to be replaced according to your EC2 setup.

From a remote client, you should now be able to access the service like this

```
$ curl http://54.154.239.169:8080/example-camel-rest/rest/greet/hello/Kermit
Hello Kermit from 172.17.0.51
```

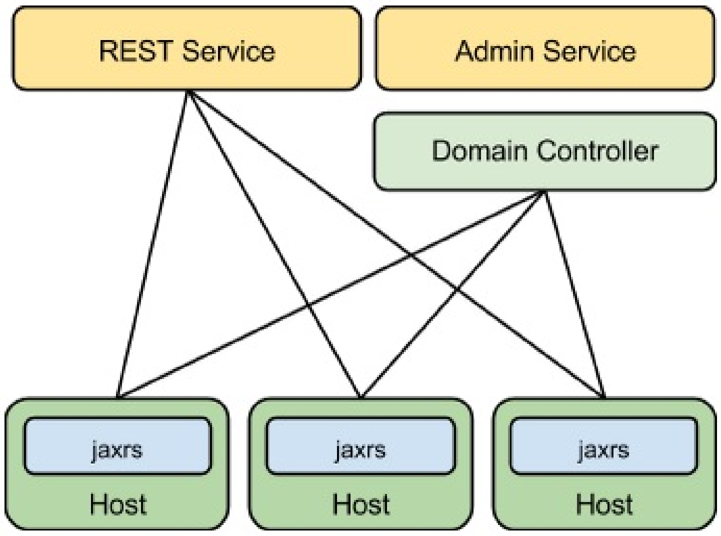
OpenShift Console

You can see a diagramatic overview of replication controllers, services and pods from the OpenShift console. Use a web browser to navigate to https://localhost:8443/console. Then log in using default username 'admin' with a password of 'admin'. Then click on the wildfly-camel project and you should see a diagram like this.



Domain Setup

Running multiple server containers in a cloud environment is often only useful when these containers can also be managed. In the previous example we had three servers that each exposed an HTTP service reachable through a Kubernetes Service. The management interface of these servers were not exposed. It would be virtually impossible to adjust configurations for these individual servers or the whole set. As a minimum we would like to monitor the health state of these servers so that we can possibly replace containers if needed.



Starting the Domain

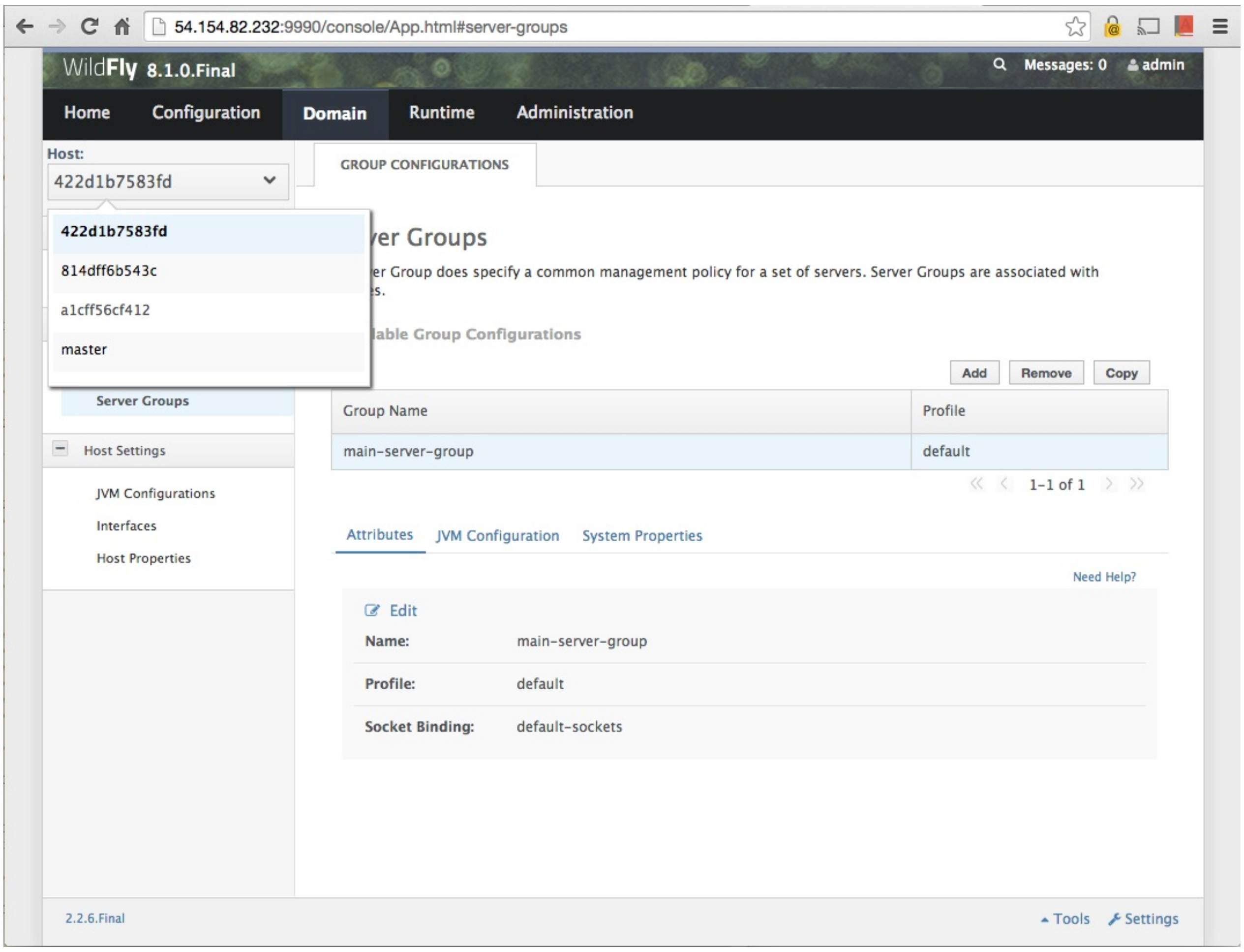
The WildFly-Camel domain can be configured as in wildfly-camel-domain.json

```
$ openshift-cli create -f https://raw.githubusercontent.com/wildfly-extras/wildfly-camel-
```

and verify the resulting servies like this

```
$ openshift-cli get services

NAME                 LABELS    SELECTOR        IP               PORT(S)
domain-controller    <none>    name=ctrl-pod   172.30.237.233   9999/TCP
                                               172.30.0.21
http-service         <none>    name=http-pod   172.30.21.77     8080/TCP
                                               172.30.0.21
management-service   <none>    name=ctrl-pod   172.30.189.153   9990/TCP
                                               172.30.0.21
```

Now, you should be able to access the admin console like this: http://54.154.82.232:9990/console



ver Groups

er Group does specify a common management policy for a set of servers. Server Groups are associated with s.

lable Group Configurations

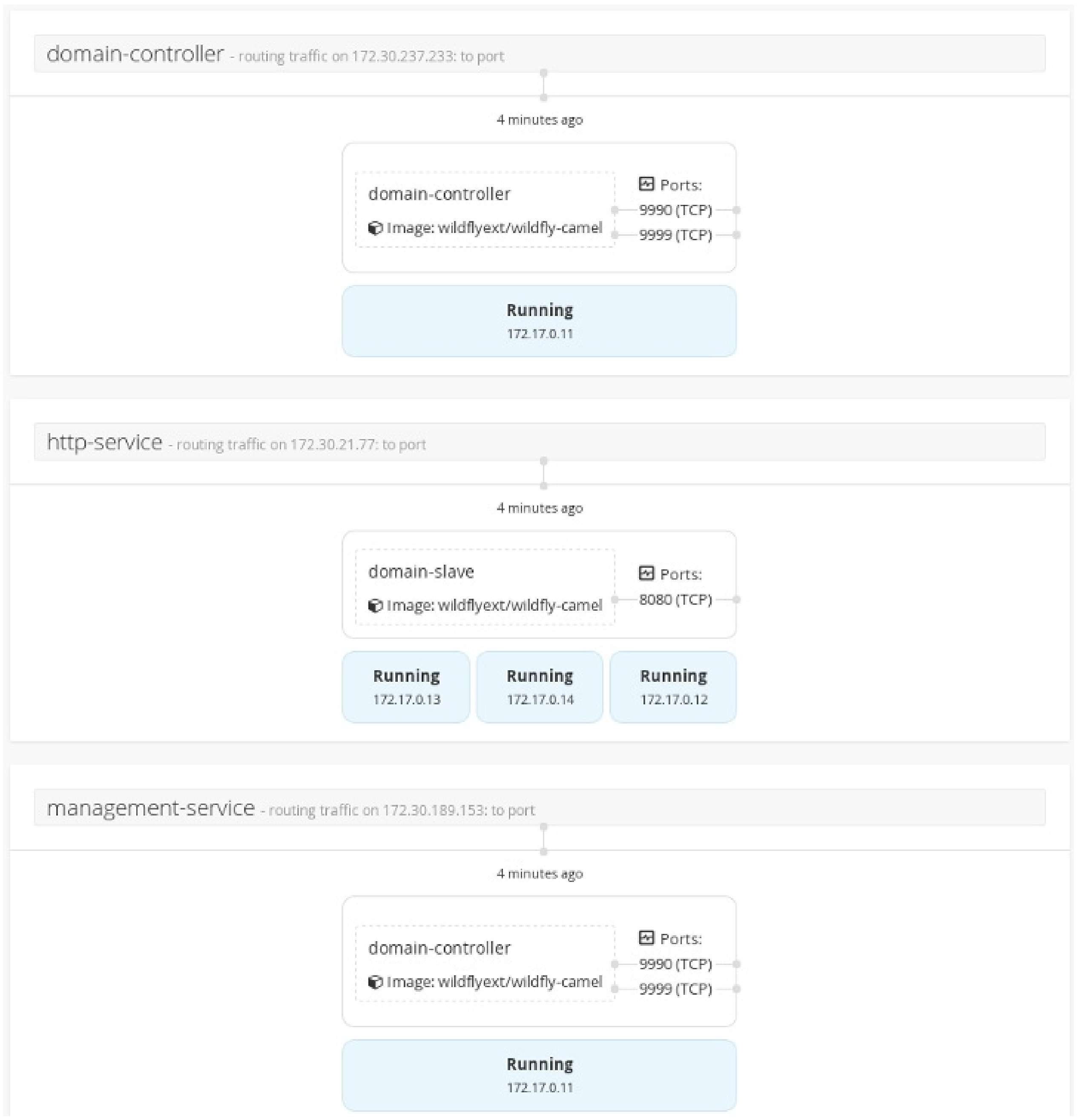
Group Name	Profile
main-server-group	default

Name: main-server-group

Profile: default

Socket Binding: default-sockets

The OpenShift console should display a diagram like this:


domain-controller - routing traffic on 172.30.237.233: to port
4 minutes ago
domain-controller
Image: wildflyext/wildfly-camel
Ports:
9990 (TCP)
9999 (TCP)
Running
172.17.0.11
http-service - routing traffic on 172.30.21.77: to port
4 minutes ago
domain-slave
Image: wildflyext/wildfly-camel
Ports:
8080 (TCP)
Running
172.17.0.13
Running
172.17.0.14
Running
172.17.0.12
management-service - routing traffic on 172.30.189.153: to port
4 minutes ago
domain-controller
Image: wildflyext/wildfly-camel
Ports:
9990 (TCP)
9999 (TCP)
Running
172.17.0.11

We believe that managing deployments through the WildFly admin interface does not make much sense. Instead, deployments should already be backed into containers that you spin up in the various Pods.

There is a wide spectrum of opinion on whether this also applies to configuration. Here we retain the WildFly domain configurability (i.e. mutable containers for configuration).

Feedback is welcome.

AWS Elastic Beanstalk

WildFly Camel also comes as docker image. This allows you to run the certified WildFly JavaEE server with Camel Integration in any managed environment that supports Docker.

Here is an easy 3-Step process to run WildFly Camel on Elastic Beanstalk

Define the Dockerrun.aws.json

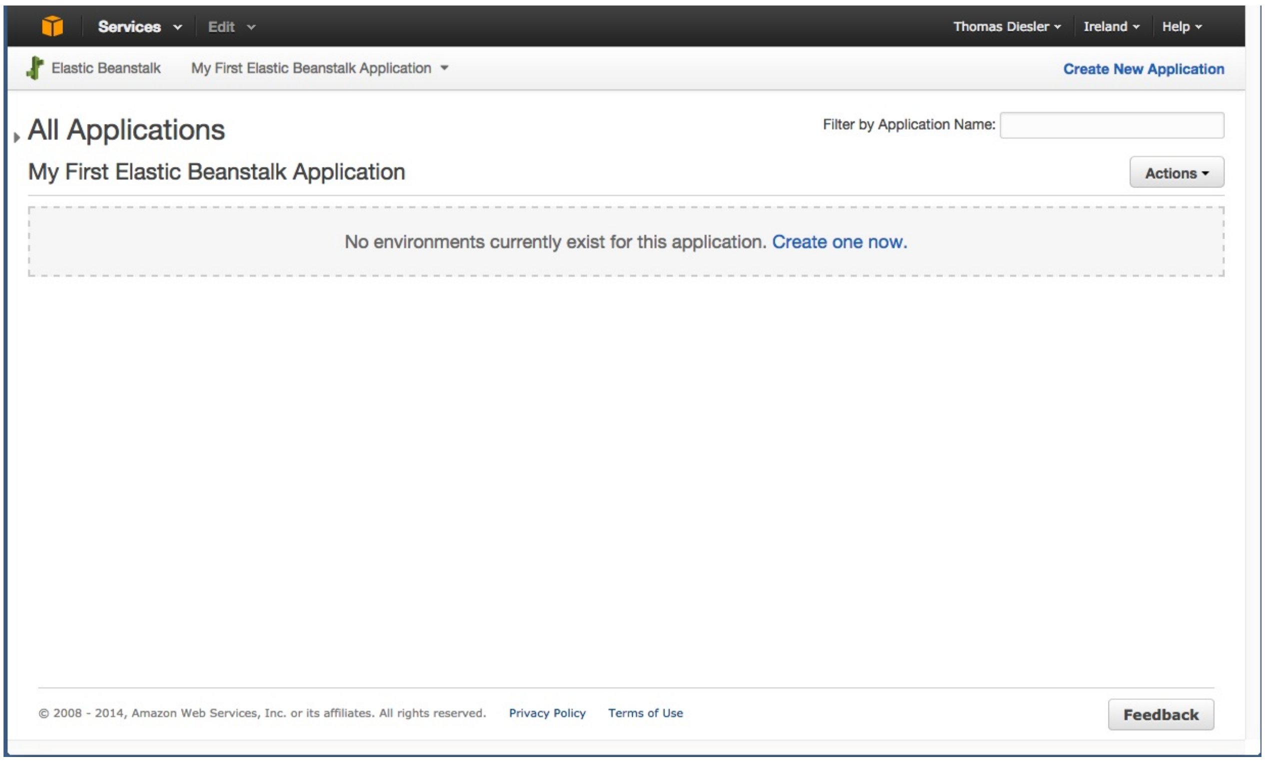
A `Dockerrun.aws.json` file describes how to deploy a Docker container as an AWS Elastic Beanstalk application.

Here is a simple example that uses a plain wildfly-camel distro image

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "wildflyext/wildfly-camel",
    "Update": "false"
  },
  "Ports": [
    {
      "ContainerPort": "8080"
    }
  ]
}
```

Create the Elastic Beanstalk Application

Navigate to Elastic Beanstalk and start creating an application.



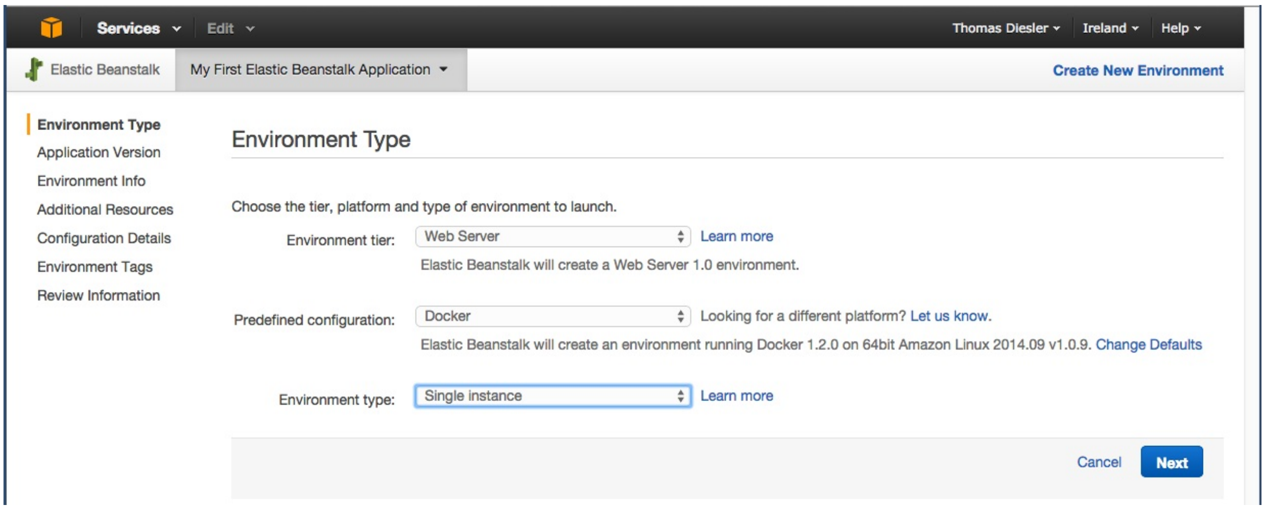
All Applications

My First Elastic Beanstalk Application

No environments currently exist for this application. Create one now.

© 2008 - 2014, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Select the environment tpe. In this case we use a simple WebServer tier with docker as a single instance.



Environment Type

Choose the tier, platform and type of environment to launch.

Environment tier: Web Server

Elastic Beanstalk will create a Web Server 1.0 environment.

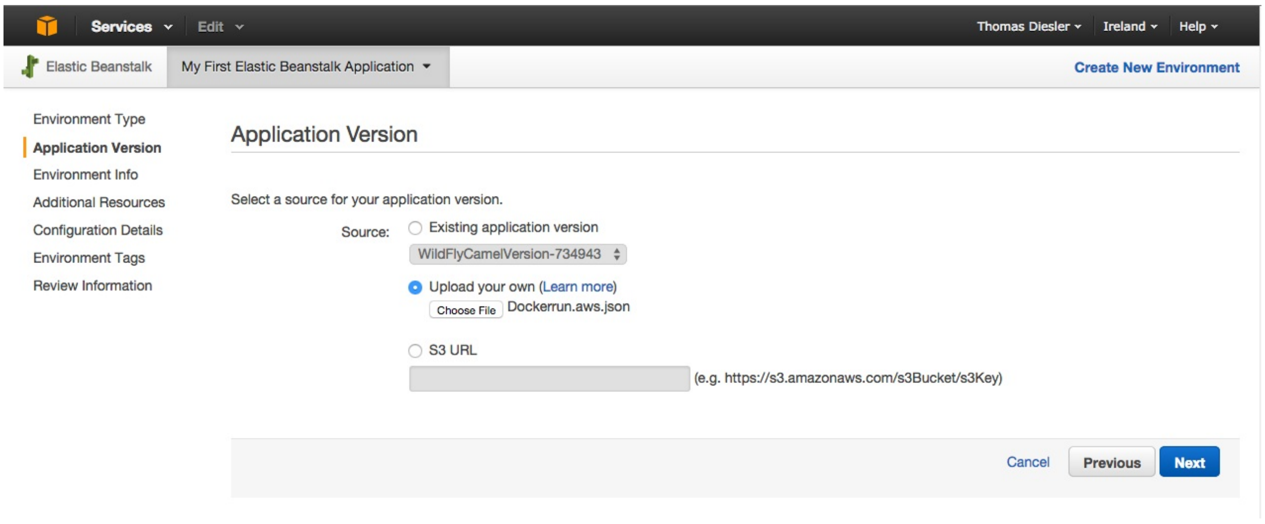
Predefined configuration: Docker

Looking for a different platform? Let us know.

Elastic Beanstalk will create an environment running Docker 1.2.0 on 64bit Amazon Linux 2014.09 v1.0.9. Change Defaults

Environment type: Single instance

Upload the `Dockerrun.aws.json` file from above.

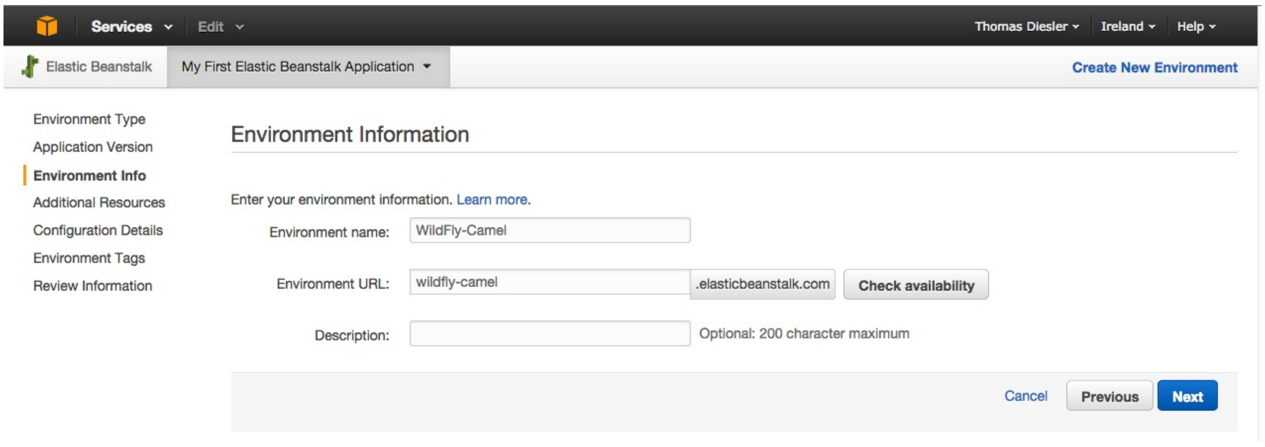


Application Version

Select a source for your application version.

Source: ○ Existing application version
WildFlyCamelVersion-734943
● Upload your own (Learn more)
Choose File Dockerrun.aws.json
○ S3 URL
(e.g. https://s3.amazonaws.com/s3Bucket/s3Key)

Give the environment a name.



Environment Information

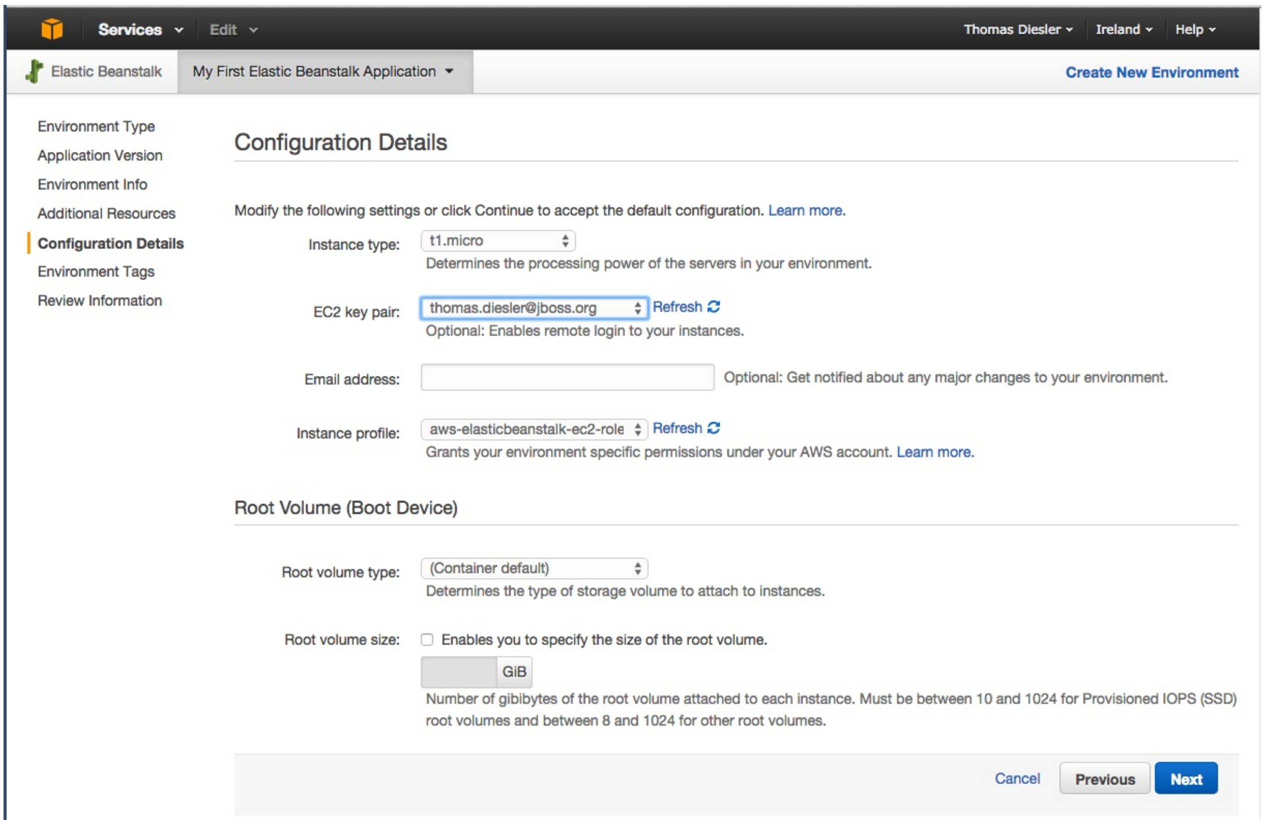
Enter your environment information. Learn more.

Environment name: WildFly-Camel

Environment URL: wildfly-camel .elasticbeanstalk.com

Description: Optional: 200 character maximum

Add more configuration. Here I select my EC2 key pair so that I can SSH into the EC2 instance if needed.



Configuration Details

Modify the following settings or click Continue to accept the default configuration. Learn more.

Instance type: t1.micro
Determines the processing power of the servers in your environment.

EC2 key pair: thomas.diesler@jboss.org Refresh
Optional: Enables remote login to your instances.

Email address: Optional: Get notified about any major changes to your environment.

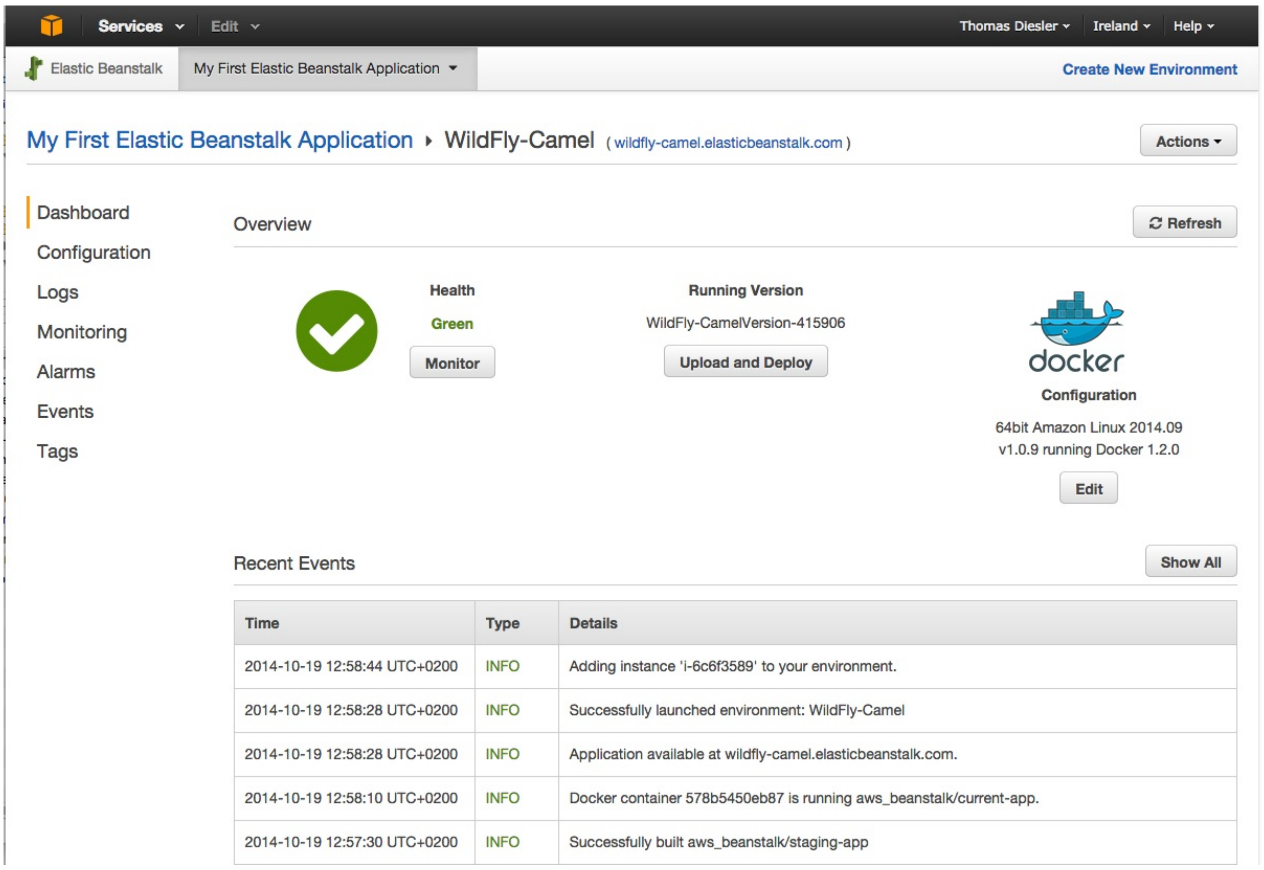
Instance profile: aws-elasticbeanstalk-ec2-role Refresh
Grants your environment specific permissions under your AWS account. Learn more.

Root Volume (Boot Device)

Root volume type: (Container default)
Determines the type of storage volume to attach to instances.

Root volume size: ☐ Enables you to specify the size of the root volume.
GiB
Number of gibibytes of the root volume attached to each instance. Must be between 10 and 1024 for Provisioned IOPS (SSD) root volumes and between 8 and 1024 for other root volumes.

Launch the the application - this may take few minutes.



My First Elastic Beanstalk Application ▸ WildFly-Camel (wildfly-camel.elasticbeanstalk.com)

Overview

Health: Green

Running Version: WildFly-CamelVersion-415906

Configuration: 64bit Amazon Linux 2014.09 v1.0.9 running Docker 1.2.0

Recent Events

Time	Type	Details
2014-10-19 12:58:44 UTC+0200	INFO	Adding instance 'i-6c6f3589' to your environment.
2014-10-19 12:58:28 UTC+0200	INFO	Successfully launched environment: WildFly-Camel
2014-10-19 12:58:28 UTC+0200	INFO	Application available at wildfly-camel.elasticbeanstalk.com.
2014-10-19 12:58:10 UTC+0200	INFO	Docker container 578b5450eb87 is running aws_beanstalk/current-app.
2014-10-19 12:57:30 UTC+0200	INFO	Successfully built aws_beanstalk/staging-app

Accessing the WildFly Camel Application

Finally, you should be able to access the wildfly-camel application on http://wildfly-camel.elasticbeanstalk.com.

Security

Security in WildFly is a broad topic. Both WildFly and Camel have well documented, stadardised methods of securing configuration, endpoints and payloads.

For details please see the security related documentation for:

- JAX-RS Security
- JAX-WS Security
- JMS Security

In addition to that, we use Camel's notion of Route Policies to integrate with the WildFly security system.

- RoutePolicy

JAX-RS Security

The following topics explain how to secure JAX-RS endpoints.

- WildFly HTTP basic authentication
- Security Realms & SSL
- Securing EJBs

JAX-WS Security

The following topics explain how to secure JAX-WS endpoints.

- WildFly HTTP basic authentication
- WS-Security
- CXF Security
- Security Realms & SSL
- Securing EJBs

JMS Security

The following topics explain how to secure JMS endpoints.

- HornetQ security documentation
- Security settings for HornetQ addresses and JMS destinations
- HornetQ security domain configuration
- ActiveMQ Security

Developer Guide

Source

https://github.com/wildflyext/wildfly-camel

Issues

https://github.com/wildflyext/wildfly-camel/issues

Jenkins

https://fabric8-ci.fusesource.com/job/wildfly-camel/

Downloads

https://github.com/wildflyext/wildfly-camel/releases

Forums, Lists, IRC

#wildfly-camel channel on irc.freenode.net