**CamelContext**

A CamelContext object represents the Camel runtime system. You typically have one CamelContext object in an application. A typical application executes the following steps.

1. Create a CamelContext object.
2. Add endpoints – and possibly Components, which are discussed in [Section 4.5 ("Components")](http://camel.apache.org/book-getting-started.html#BookGettingStarted-components-and-uris) – to the CamelContext object.
3. Add routes to the CamelContext object to connect the endpoints.
4. Invoke the start() operation on the CamelContext object. This starts Camel-internal threads that are used to process the sending, receiving and processing of messages in the endpoints.
5. Eventually invoke the stop() operation on the CamelContext object. Doing this gracefully stops all the endpoints and Camel-internal threads.

……

### Components

Component is confusing terminology; EndpointFactory would have been more appropriate because a Component is a factory for creating Endpoint instances. For example, if a Camel-based application uses several JMS queues then the application will create one instance of the JmsComponent class (which implements the Component interface), and then the application invokes the createEndpoint() operation on this JmsComponent object several times. Each invocation of JmsComponent.createEndpoint() creates an instance of the JmsEndpoint class (which implements the Endpoint interface). Actually, application-level code does not invoke Component.createEndpoint() directly. Instead, application-level code normally invokes CamelContext.getEndpoint(); internally, the CamelContext object finds the desired Component object (as I will discuss shortly) and then invokes createEndpoint() on it.  
Consider the following code.

|  |
| --- |
| myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword"); |

The parameter to getEndpoint() is a URI. The URI prefix (that is, the part before ":") specifies the name of a component. Internally, the CamelContext object maintains a mapping from names of components to Componentobjects. For the URI given in the above example, the CamelContext object would probably map the pop3 prefix to an instance of the MailComponent class. Then the CamelContext object invokes createEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword") on that MailComponent object. The createEndpoint() operation splits the URI into its component parts and uses these parts to create and configure an Endpoint object.

In the previous paragraph, I mentioned that a CamelContext object maintains a mapping from component names to Component objects. This raises the question of how this map is populated with named Component objects. There are **two ways** of populating the map. The first way is for application-level code to invoke CamelContext.addComponent(String componentName, Component component). The example below shows a single MailComponentobject being registered in the map under 3 different names.

|  |
| --- |
| Component mailComponent = new org.apache.camel.component.mail.MailComponent();  myCamelContext.addComponent("pop3", mailComponent);  myCamelContext.addComponent("imap", mailComponent);  myCamelContext.addComponent("smtp", mailComponent); |

The second (and preferred) way to populate the map of named Component objects in the CamelContext object is to let the CamelContext object perform lazy initialization. This approach relies on developers following a convention when they write a class that implements the Component interface. I illustrate the convention by an example. Let's assume you write a class called com.example.myproject.FooComponent and you want Camel to automatically recognize this by the name "foo". To do this, you have to write a properties file called "META-INF/services/org/apache/camel/component/foo" (without a ".properties" file extension) that has a single entry in it called class, the value of which is the fully-scoped name of your class. This is shown below.

**META-INF/services/org/apache/camel/component/foo**

|  |
| --- |
| class=com.example.myproject.FooComponent |

If you want Camel to also recognize the class by the name "bar" then you write another properties file in the same directory called "bar" that has the same contents. Once you have written the properties file(s), you create a jar file that contains the com.example.myproject.FooComponent class and the properties file(s), and you add this jar file to your CLASSPATH. Then, when application-level code invokes createEndpoint("foo:...") on a CamelContext object, Camel will find the "foo"" properties file on the CLASSPATH, get the value of the class property from that properties file, and use reflection APIs to create an instance of the specified class.

As I said in [Section 4.1 ("Endpoint")](http://camel.apache.org/book-getting-started.html#BookGettingStarted-endpoint), Camel provides out-of-the-box support for numerous communication technologies. The out-of-the-box support consists of classes that implement the Component interface plus properties files that enable a CamelContext object to populate its map of named Component objects.

Earlier in this section I gave the following example of calling CamelContext.getEndpoint().

|  |
| --- |
| myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword"); |

When I originally gave that example, I said that the parameter to getEndpoint() was a URI. I said that because the online Camel documentation and the Camel source code both claim the parameter is a URI. In reality, the parameter is restricted to being a URL. This is because when Camel extracts the component name from the parameter, it looks for the first ":", which is a simplistic algorithm. To understand why, recall from [Section 4.4 ("The Meaning of URL, URI, URN and IRI")](http://camel.apache.org/book-getting-started.html#BookGettingStarted-url-uri-urn-iri) that a URI can be a URL or a URN. Now consider the following calls to getEndpoint.

|  |
| --- |
| myCamelContext.getEndpoint("pop3:...");  myCamelContext.getEndpoint("jms:...");  myCamelContext.getEndpoint("urn:foo:...");  myCamelContext.getEndpoint("urn:bar:..."); |

Camel identifies the components in the above example as "pop3", "jms", "urn" and "urn". It would be more useful if the latter components were identified as "urn:foo" and "urn:bar" or, alternatively, as "foo" and "bar" (that is, by skipping over the "urn:" prefix). So, in practice you must identify an endpoint with a URL (a string of the form "<scheme>:...") rather than with a URN (a string of the form "urn:<scheme>:..."). This lack of proper support for URNs means the you should consider the parameter to getEndpoint() as being a URL rather than (as claimed) a URI.

……

### Message and Exchange

The Message interface provides an abstraction for a single message, such as a request, reply or exception message.

There are concrete classes that implement the Message interface for each Camel-supported communications technology. For example, the JmsMessage class provides a JMS-specific implementation of the Message interface. The public API of the Message interface provides get- and set-style methods to access the message id, body and individual header fields of a messge.

The Exchange interface provides an abstraction for an exchange of messages, that is, a request message and its corresponding reply or exception message. In Camel terminology, the request, reply and exception messages are called **in**, **out** and **fault** messages.

There are concrete classes that implement the Exchange interface for each Camel-supported communications technology. For example, the JmsExchange class provides a JMS-specific implementation of the Exchange interface. The public API of the Exchange interface is quite limited. This is intentional, and it is expected that each class that implements this interface will provide its own technology-specific operations.  
Application-level programmers rarely access the Exchange interface (or classes that implement it) directly. However, many classes in Camel are generic types that are instantiated on (a class that implements) Exchange. Because of this, the Exchange interface appears a lot in the generic signatures of classes and methods.

……

### Processor

The Processor interface represents a class that processes a message. The signature of this interface is shown below.

**Processor**

|  |
| --- |
| package org.apache.camel;  public interface Processor {      void process(Exchange exchange) throws Exception;  } |

Notice that the parameter to the process() method is an Exchange rather than a Message. This provides flexibility. For example, an implementation of this method initially might call exchange.getIn() to get the input message and process it. If an error occurs during processing then the method can call exchange.setException().

An application-level developer might implement the Processor interface with a class that executes some business logic. However, there are many classes in the Camel library that implement the Processor interface in a way that provides support for a design pattern in the [EIP book](http://camel.apache.org/book-getting-started.html#BookGettingStarted-eip-book). For example, ChoiceProcessor implements the message router pattern, that is, it uses a cascading if-then-else statement to route a message from an input queue to one of several output queues. Another example is the FilterProcessor class which discards messages that do not satisfy a stated predicate (that is, condition).

### Routes, RouteBuilders and Java DSL

A route is the step-by-step movement of a Message from an input queue, through arbitrary types of decision making (such as filters and routers) to a destination queue (if any). Camel provides **two ways** for an application developer to specify routes. One way is to specify route information in an XML file. A discussion of that approach is outside the scope of this document. The other way is through what Camel calls a Java DSL (domain-specific language).

#### Introduction to Java DSL

For many people, the term "domain-specific language" implies a compiler or interpreter that can process an input file containing keywords and syntax specific to a particular domain. This is not the approach taken by Camel. Camel documentation consistently uses the term "Java DSL" instead of "DSL", but this does not entirely avoid potential confusion. The Camel "Java DSL" is a class library that can be used in a way that looks almost like a DSL, except that it has a bit of Java syntactic baggage. You can see this in the example below. Comments afterwards explain some of the constructs used in the example.

**Example of Camel's "Java DSL"**

|  |
| --- |
| RouteBuilder builder = new RouteBuilder() {      public void configure() {          from("queue:a").filter(header("foo").isEqualTo("bar")).to("queue:b");          from("queue:c").choice()                  .when(header("foo").isEqualTo("bar")).to("queue:d")                  .when(header("foo").isEqualTo("cheese")).to("queue:e")                  .otherwise().to("queue:f");      }  };  CamelContext myCamelContext = new DefaultCamelContext();  myCamelContext.addRoutes(builder); |

The first line in the above example creates an object which is an instance of an anonymous subclass of RouteBuilder with the specified configure() method.  
The CamelContext.addRoutes(RouterBuilder builder) method invokes builder.setContext(this) – so the RouteBuilder object knows which CamelContext object it is associated with – and then invokes builder.configure(). The body of configure() invokes methods such as from(), filter(), choice(), when(), isEqualTo(), otherwise() and to().  
The RouteBuilder.from(String uri) method invokes getEndpoint(uri) on the CamelContext associated with the RouteBuilder object to get the specified Endpoint and then puts a FromBuilder "wrapper" around this Endpoint. The FromBuilder.filter(Predicate predicate) method creates a FilterProcessor object for the Predicate (that is, condition) object built from the header("foo").isEqualTo("bar") expression. In this way, these operations incrementally build up a Route object (with a RouteBuilder wrapper around it) and add it to the CamelContext object associated with the RouteBuilder.

<http://camel.apache.org/book-getting-started.html#BookGettingStarted-message-and-exchange>

At the core of the Camel framework is a routing engine, or more precisely a routing engine

builder. It allows you to define your own routing rules, decide from which

sources to accept messages, and determine how to process and send those messages to

other destinations. Camel uses an integration language that allows you to define complex

routing rules, akin to business processes.

……

Camel offers higher-level abstractions that allow you to **interact with** various **systems**

using the **same API** regardless of the **protocol** or **data type** the systems are using. **Components** in Camel provide specific implementations of the API that target different

protocols and data types. Out of the box, Camel comes with support for over 80

protocols and data types. Its extensible and modular architecture allows you to implement

and seamlessly plug in support for your own protocols, proprietary or not.

These architectural choices eliminate the need for unnecessary conversions and make

Camel not only faster but also very lean. As a result, it’s suitable for embedding into

other projects that require Camel’s rich processing capabilities. Other open source

projects, such as Apache ServiceMix and ActiveMQ, already use Camel as a way to

carry out enterprise integration.

We should also mention what Camel isn’t. Camel isn’t an enterprise service bus

(ESB), although some call Camel a lightweight ESB because of its support for routing,

transformation, monitoring, orchestration, and so forth. Camel doesn’t have a

container or a reliabled message bus, but it can be deployed in one, such as Open-

ESB or the previously mentioned ServiceMix.

***Camel In Action 1.1.1***