Book Getting Started

Getting Started with Apache Camel

The *Enterprise Integration Patterns* (EIP) book

The purpose of a "patterns" book is not to advocate new techniques that the authors have invented, but rather to document existing best practices within a particular field. By doing this, the authors of a patterns book hope to spread knowledge of best practices and promote a vocabulary for discussing architectural designs.
One of the most famous patterns books is *Design Patterns: Elements of Reusable Object-oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, commonly known as the "Gang of Four" (GoF) book. Since the publication of *Design Patterns*, many other pattern books, of varying quality, have been written. One famous patterns book is called *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf. It is common for people to refer to this book by its initials *EIP*. As the subtitle of EIP suggests, the book focuses on design patterns for asynchronous messaging systems. The book discusses 65 patterns. Each pattern is given a textual name and most are also given a graphical symbol, intended to be used in architectural diagrams.

The Camel project

Camel (http://camel.apache.org) is an open-source, Java-based project that helps the user implement many of the design patterns in the EIP book. Because Camel implements many of the design patterns in the EIP book, it would be a good idea for people who work with Camel to have the EIP book as a reference.

Online documentation for Camel

The documentation is all under the Documentation category on the right-side menu of the Camel website (also available in PDF form. Camel-related books are also available, in particular the Camel in Action book, presently serving as the Camel bible--it has a free Chapter One (pdf), which is highly recommended to read to get more familiar with Camel.

A useful tip for navigating the online documentation

The breadcrumbs at the top of the online Camel documentation can help you navigate between parent and child subsections.
For example, If you are on the "Languages" documentation page then the left-hand side of the reddish bar contains the following links.

```
Apache Camel > Documentation > Architecture > Languages
```

As you might expect, clicking on "Apache Camel" takes you back to the home page of the Apache Camel project, and clicking on "Documentation" takes you to the main documentation page. You can interpret the "Architecture" and "Languages" buttons as indicating you are in the "Languages" section of the "Architecture" chapter. Adding browser bookmarks to pages that you frequently reference can also save time.

Online Javadoc documentation

The Apache Camel website provides Javadoc documentation. It is important to note that the Javadoc documentation is spread over several *independent* Javadoc hierarchies rather than being all contained in a single Javadoc hierarchy. In particular, there is one Javadoc hierarchy for the *core* APIs of Camel, and a separate Javadoc hierarchy for each component technology supported by Camel. For example, if you will be using Camel with ActiveMQ and FTP then you need to look at the Javadoc hierarchies for the core API and Spring API.

Concepts and terminology fundamental to Camel

In this section some of the concepts and terminology that are fundamental to Camel are explained. This section is not meant as a complete Camel tutorial, but as a first step in that direction.

Endpoint

The term *endpoint* is often used when talking about inter-process communication. For example, in client-server communication, the client is one endpoint and the server is the other endpoint. Depending on the context, an endpoint might refer to an *address*, such as a host:port pair for TCP-based communication, or it might refer to a *software entity* that is contactable at that address. For example, if somebody uses "www.example.com:80" as an example of an endpoint, they might be referring to the actual port at that host name (that is, an address), or they might be referring to the web server (that is, software contactable at that address). Often, the distinction between the address and software contactable at that address is not an important one.
Some middleware technologies make it possible for several software entities to be contactable at the same physical address. For example, CORBA is an object-oriented, remote-procedure-call (RPC) middleware standard. If a CORBA server process contains several objects then a client can communicate with any of these objects at the same *physical* address (host:port), but a client communicates with a particular object via that object's *logical* address (called an *IOR* in CORBA terminology), which consists of the physical address (host:port) plus an id that uniquely identifies the object within its server process. (An IOR contains some additional information that is not relevant to this present discussion.) When talking about CORBA, some people may use the term "endpoint" to refer to a CORBA server's *physical address*, while other people may use the term to refer to the *logical address* of a single CORBA object, and other people still might use the term to refer to any of the following:

- The physical address (host:port) of the CORBA server process
- The logical address (host:port plus id) of a CORBA object.
- The CORBA server process (a relatively heavyweight software entity)
- A CORBA object (a lightweight software entity)

Because of this, you can see that the term *endpoint* is ambiguous in at least two ways. First, it is ambiguous because it might refer to an address or to a software entity contactable at that address. Second, it is ambiguous in the *granularity* of what it refers to: a heavyweight versus lightweight software entity, or physical address versus logical address. It is useful to understand that different people use the term *endpoint* in slightly different (and hence ambiguous) ways because Camel's usage of this term might be different to whatever meaning you had previously associated with the term.
Camel provides out-of-the-box support for endpoints implemented with many different communication technologies. Here are some examples of the Camel-supported endpoint technologies.

- A JMS queue.
- A web service.
- A file. A file may sound like an unlikely type of endpoint, until you realize that in some systems one application might write information to a file and, later, another application might read that file.
- An FTP server.
- An email address. A client can send a message to an email address, and a server can read an incoming message from a mail server.
- A POJO (plain old Java object).

In a Camel-based application, you create (Camel wrappers around) some endpoints and connect these endpoints with *routes*, which I will discuss later in Section 4.8 ("Routes, RouteBuilders and Java DSL"). Camel defines a Java interface called `Endpoint`. Each Camel-supported endpoint has a class that implements this `Endpoint` interface. As I discussed in Section 3.3 ("Online Javadoc documentation"), Camel provides a separate Javadoc hierarchy for each communications technology supported by Camel. Because of this, you will find documentation on, say, the `JmsEndpoint` class in the JMS Javadoc hierarchy, while documentation for, say, the `FtpEndpoint` class is in the FTP Javadoc hierarchy.

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") – 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.

Note that the `CamelContext.start()` operation does not block indefinitely. Rather, it starts threads internal to each `Component` and `Endpoint` and then `start()` returns. Conversely, `CamelContext.stop()` waits for all the threads internal to each `Endpoint` and `Component` to terminate and then `stop()` returns.
If you neglect to call `CamelContext.start()` in your application then messages will not be processed because internal threads will not have been created.
If you neglect to call `CamelContext.stop()` before terminating your application then the application may terminate in an inconsistent state. If you neglect to call `CamelContext.stop()` in a JUnit test then the test may fail due to messages not having had a chance to be fully processed.

CamelTemplate

Camel used to have a class called `CamelClient`, but this was renamed to be `CamelTemplate` to be similar to a naming convention used in some other open-source projects, such as the `TransactionTemplate` and `JmsTemplate` classes in Spring.
The `CamelTemplate` class is a thin wrapper around the `CamelContext` class. It has methods that send a `Message` or `Exchange` – both discussed in Section 4.6 ("Message and Exchange")) – to an `Endpoint` – discussed in Section 4.1 ("Endpoint"). This provides a way to enter messages into source endpoints, so that the messages will move along routes – discussed in Section 4.8 ("Routes, RouteBuilders and Java DSL") – to destination endpoints.

The Meaning of URL, URI, URN and IRI

Some Camel methods take a parameter that is a *URI* string. Many people know that a URI is "something like a URL" but do not properly understand the relationship between URI and URL, or indeed its relationship with other acronyms such as IRI and URN.
Most people are familiar with *URLs* (uniform resource locators), such as "http://...", "ftp://...", "mailto:...". Put simply, a URL specifies the *location* of a resource.
A *URI* (uniform resource identifier) is a URL *or* a URN. So, to fully understand what URI means, you need to first understand what is a URN.
URN is an acronym for *uniform resource name*. There are may "unique identifier" schemes in the world, for example, ISBNs (globally unique for books), social security numbers (unique within a country), customer numbers (unique within a company's customers database) and telephone numbers. Each "unique identifier" scheme has its own notation. A URN is a wrapper for different "unique identifier" schemes. The syntax of a URN is "urn:<scheme-name>:<unique-identifier>". A URN uniquely identifies a *resource*, such as a book, person or piece of equipment. By itself, a URN does not specify the *location* of the resource. Instead, it is assumed that a *registry* provides a mapping from a resource's URN to its location. The URN specification does not state what form a registry takes, but it might be a database, a server application, a wall chart or anything else that is convenient. Some hypothetical examples of URNs are "urn:employee:08765245", "urn:customer:uk:3458:hul8" and "urn:foo:0000-0000-9E59-0000-5E-2". The <scheme-name> ("employee", "customer" and "foo" in these examples) part of a URN implicitly defines how to parse and interpret the <unique-identifier> that follows it. An arbitrary URN is meaningless unless: (1) you know the semantics implied by the <scheme-name>, and (2) you have access to the registry appropriate for the <scheme-name>. A registry does not have to be public or globally accessible. For example, "urn:employee:08765245" might be meaningful only within a specific company.
To date, URNs are not (yet) as popular as URLs. For this reason, URI is widely misused as a synonym for URL.
IRI is an acronym for *internationalized resource identifier*. An IRI is simply an internationalized version of a URI. In particular, a URI can contain letters and digits in the US-ASCII character set, while a IRI can contain those same letters and digits, and *also* European accented characters, Greek letters, Chinese ideograms and so on.

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 `Component` objects. 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 `MailComponent` object 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"), 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") 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. 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`.

Critique of Java DSL

The online Camel documentation compares Java DSL favourably against the alternative of configuring routes and endpoints in a XML-based Spring configuration file. In particular, Java DSL is less verbose than its XML counterpart. In addition, many integrated development environments (IDEs) provide an auto-completion feature in their editors. This auto-completion feature works with Java DSL, thereby making it easier for developers to write Java DSL. However, there is another option that the Camel documentation neglects to consider: that of writing a parser that can process DSL stored in, say, an external file. Currently, Camel does not provide such a DSL parser, and I do not know if it is on the "to do" list of the Camel maintainers. I think that a DSL parser would offer a significant benefit over the current Java DSL. In particular, the DSL would have a syntactic definition that could be expressed in a relatively short BNF form. The effort required by a Camel user to learn how to use DSL by reading this BNF would almost certainly be significantly less than the effort currently required to study the API of the `RouterBuilder` classes.

Continue Learning about Camel

Return to the main Getting Started page for additional introductory reference information.