Camel

Essential Components

BY CHRISTIAN POSTA

WHAT IS APACHE CAMEL?

Camel is an open-source, lightweight integration library that allows your applications to accomplish intelligent routing, message transformation, and protocol mediation using the established Enterprise Integration Patterns and out-of-the-box components with a highly expressive Domain Specific Language (Java, XML, or Scala). With Camel you can implement integration solutions as part of a microservices architecture (ingress routing, legacy integration, bounded context inter-communication) or as a more centralized integration platform if that's what you need.

WHY USE CAMEL?

Camel simplifies service integrations with an easy-to-use DSL to create routes that **clearly** identify the integration intentions and endpoints. Camel's out-of-the-box integration components are modeled after the Enterprise Integration Patterns cataloged in Gregor Hohpe and Bobby Wolf's book (eaipatterns.com). You can use these EIPs as pre-packaged units, along with any custom processors or external adapters you may need, to easily assemble otherwise complex routing and transformation routes. For example, this route takes a JSON message from a queue, does some processing, and publishes to another queue:

```
from( "jms:incomingQueue" )
  .unmarshal().json()
  .process(new Processor() {

    @Override
    public void process( Exchange exchange ) throws
      Exception {
        .... Custom Logic If Needed ....
    }
})
  .marshal().json()
  .to( "jms:outgoingQueue" );
```

Camel allows you to integrate with quite a few protocols and systems out of the box (about 200 right now) using Camel components, and the community is constantly adding more. Each component is highly flexible and can be easily configured using Camel's consistent URI syntax. This Refcard introduces you to the following popular and widely used components and their configurations:

- Log/Metrics
- JMS/ActiveMQ
- REST components
- File
- SEDA/Direct
- Mock

CONFIGURING CAMEL COMPONENTS

All Camel components can be configured with a familiar URI syntax. Usually components are specified in the from() clause (beginning of a route) or the to() clause (typically termination points in the route, but not always). A component "creates" specific "endpoint" instances. Think of a component as an "endpoint factory" where an endpoint is the actual object listening for a message (in the "from" scenario) or sending a message (in the "to" scenario).

An endpoint URI usually has the form:

```
<component:><//endpoint-local-name>
<?config=value&config=value>
```

The component is listed first, followed by some local name specific to the endpoint, and finally some endpoint-specific configuration specified as key-value pairs.

Example:

```
jms:test-queue?preserveMessageQos=true
```

This URI creates a JMS endpoint that listens to the "test-queue" and sets the `preserveMessageQos` option to true.

There are around 200 Camel components and each one has configuration options to more finely tune its behavior. See the Camel components list to see if a component is available for the types of integrations you might need: camel.apache.org/components.html

ESSENTIAL COMPONENTS

BEAN COMPONENT

Camel implements the Service Activator pattern, which allows you to plug a POJO service into your route through Camel's binding mechanism responsible for mediating

between Camel and your service. Camel implements this with its "bean" component as we explore below:

BASIC USAGE

Define your bean in the Camel registry (usually the Spring context if you're instantiating from a Spring application).

```
@Bean
public YourPojo yourPojo(){
    return new YourPojo();
}
```

Then, from your route, you can invoke a method on the bean:

```
from("direct:incoming").beanRef("yourPojo", "methodName")
```

Instead of relying on the registry to have the bean, you could just let Camel create and manage the bean by supplying the class:

```
from("direct:incoming").bean(YourPojo.class, "methodName")
```

HOW IS A METHOD CHOSEN?

You don't always have to supply an explicit method name, but if you do, it makes it easier on both Camel and reading the route. If you don't supply an explicit method name, Camel will do it's best to resolve the correct method. Follow these rules to be safe, but for a more in-depth treatment of the algorithm, please see the Camel documentation for more (camel.apache.org/bean-binding.html):

- If there is only a single method, Camel will try to use that.
- If there are multiple methods, Camel will try to select one that has only one parameter.
- Camel will try to match a method's parameters by type to the incoming message body.

HOW DOES CAMEL BIND PARAMETERS?

Camel will automatically try to bind the first parameter to the body of the incoming message. Otherwise, if you have parameters of these types, they will automatically be bound, regardless of order:

- **CamelContext** – use when you need access to the full context, components defined in the context, or to create objects that require the context
- **Registry** – the registry is the object that holds all of the beans that might be used in a CamelContext; usually the Spring BeanFactory/ApplicationFactory (but not always)
- **Exchange** – contains headers, body, properties, and overall state for processing a message unit including a reply
- **Message** – the incoming message
- **TypeConverter** – part of Camel's type conversion internals, can be used to explicitly convert types

You can also use annotations to specifically bind certain parts of the Exchange to parameters in your methods. See the Camel documentation (camel.apache.org/parameter-binding-annotations.html) for more.

FURTHER READING

For more information on the bean component, see camel.apache.org/bean.html

LOG AND METRICS COMPONENTS

In a services architecture, it's important to know what's happening at all times (happy path or failure). One way to do that is through diligent logging. Another is by exposing metrics. Camel's Log Component and Metrics component come into play here.

LOGGING

To log an exchange at debug level:

```
from("direct:start")
  .to("log:com.fusesource.examples?level=DEBUG")
```

This will output the exchange type (`InOnly`/`InOut`) and the body of the **In** message. You do have control over what is logged; for example, to see the headers along with the exchange:

```
from("direct:start")
  .to("log:com.fusesource.examples?showHeaders=true")
```

To make the Exchange logging easier to read, consider enabling multiline printing:

```
from("direct:start").to("log:com.fusesource
  .examples?showHeaders=true&multiline=true")
from("direct:start").streamCaching().to("log:com.fusesource
  .examples?showHeaders=true&showStreams=true")
```

LOG FORMATTING

You can add configurations to fine-tune exactly what's logged; see the following options that can be enabled/disabled:

OPTION	DESCRIPTION
`showAll`	Turns all options on, such as headers, body, out, stackTrace, etc.
`showExchangeId`	Prints out the exchange ID
`showHeaders`	Shows all headers in the in message
`showBodyType`	Shows the Java type for the body
`showBody`	Shows the actual contents of the body
`showOut`	Shows the out message
`showException`	If there's an exception in the exchange, show the exception. Doesn't show the full stack trace
`showStackTrace`	Print the stack trace from the exception
`multiline`	Log each part of the exchange and its details on separate lines for better readability
`maxChars`	The maximum number of characters logged per line

METRICS

You can use the metrics component to report fine-grained metrics using counters, histograms, meters, and timers. The metrics component is implemented with Dropwizard/Codahale Metrics. You can retrieve the metric values by plugging in different metrics reporters or via JMX. For example, to count the occurrence of certain patterns in a route:

```
from("direct:start").to("metrics:counter:simple.counter")
.to("jms:inventory")
```

Or to add values to a histogram where 700 is the value we want to increment in the histogram:

```
from(“direct:start”)
  .to(“metrics:histogram:simple.histo?value=700”)
  .to(“jms:inventory”)
```

You can override the value to use for the histogram with a header:

```
from(“direct:start”)
  .setHeader(MetricsConstant.HEADER\_HISTOGRAM\_VALUE)
  .to(“metrics:historgram:simple.histo”)
  .to(“jms:inventory”)
```

FURTHER READING

For more information on the log and metrics component, see camel.apache.org/log.html and camel.apache.org/metrics-component.html.

JMS AND ACTIVEMQ COMPONENT

The Camel JMS component allows you to produce or consume from a JMS provider. Most commonly, JMS and associated routes are used for asynchronous `inOnly` style messaging; however, camel-jms understands request-reply or `inOut` style messaging also and hides the details. Use the activemq-camel component for JMS endpoints that rely on an ActiveMQ provider as it provides some simple optimizations.

CONSUME FROM A QUEUE

Note, you can specify "`jms:queue:incomingOrders`" but "`queue`" is default so can be left off:

```
from("jms:incomingOrders").process(<some processing>)
  .to(“jms:inventoryCheck”);
```

CONSUME FROM A TOPIC

Here we specify “topic” in the URI:

```
from("jms:topic:incomingOrders").process(<some processing>)
  .to(“jms:inventoryCheck”);
```

SET UP POOLED RESOURCES

JMS resources are often expensive to create. Connections, Sessions, Producers, Consumers should be cached where it makes sense, and setting up the appropriate caching starts with a pooled connection factory. If you're using ActiveMQ, you can use the org.apache.activemq.pool.PooledConnectionFactory. Alternatively, for general purpose JMS connection pooling, you can use the Spring's CachingConnectionFactory. See the documentation for using the CachingConnectionFactory.

When using ActiveMQ:

```
<bean id="connectionFactory" class="org.apache.activemq.
ActiveMQConnectionFactory">
    <property name="brokerURL"
      value="tcp://localhost:61616" />
</bean>

<bean id="jmsPooledConnectionFactory"
  class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory"
      ref="connectionFactory">
    </property>
</bean>
```

REQUEST/REPLY

By default, Camel will use a Message Exchange Pattern of `InOnly` when dealing with JMS. However, it will be changed to `InOut` if there is a `JMSReplyTo` header or if explicitly set:

```
from("direct:incoming").inOut().to("jms:outgoingOrders")
  .process(process jms response here and continue your route)
```

In the previous example, the MEP was set to `InOut`, which means before sending to the `outgoingOrders` queue, Camel will create a temporary destination, listen to it, and set the `JMSReplyTo` header to the temporary destination before it sends the JMS message. Some other consumer will need to listen to `outgoingOrders` queue and send a reply to the destination named in the `JMSReplyTo` header. The response received over the temporary destination will be used in further processing in the route. Camel will listen for a default 20s on the temporary destination before it times out. If you prefer to use named queues instead of temporary ones, you can set the `replyTo` configuration option:

```
from("direct:incoming").inOut()
  .to("jms:outgoingOrders?replyTo=outgoingOrdersReply")
  .process(process jms response here and continue your route)
```

TRANSACTIONS

To use transactions with camel-jms, you should set up a pooled connection factory, a transaction manager, and configure the camel-jms configuration in a `JmsConfiguration` object:

```
<!-- JmsConfiguration object -->
<bean id="jmsConfig" class="org.apache.camel.component.jms.
JmsConfiguration" >
    <property name="connectionFactory"
ref="jmsPooledConnectionFactory" />
    <property name="transacted" value="true" />
    <property name="transactionManager"
ref="jmsTransactionManager" />
    <property name="cacheLevelName" value="CACHE\_CONSUMER"
/>
</bean>

<!-- Spring Transaction Manager -->
<bean id="jmsTransactionManager" class="org.springframework.
jms.connection.JmsTransactionManager">
    <property name="connectionFactory"
ref="jmsPooledConnectionFactory" />
</bean>

<!-- Set up Pooled Connection Factory -->
<bean id="jmsPooledConnectionFactory" class="org.apache.
activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.
ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://
localhost:61616" />
        </bean>
    </property>
</bean>
```

Make sure to set cache level to `CACHE_CONSUMER` so that your consumers, sessions, and connections are cached between messages.

Updated for Camel 2.10, you can now use "local" transactions with the camel-jms consumer (you no longer have to specify an external transaction manager):

```
<bean id="jmsConfig"
  class="org.apache.camel.component.jms.JmsConfiguration" >
    <property name="connectionFactory"
      ref="jmsPooledConnectionFactory" />
    <property name="transacted" value="true" />
    <property name="cacheLevelName"
      value="CACHE\_CONSUMER" />
</bean>
```

COMMON CONFIGURATION OPTIONS

OPTION	DEFAULT	DESCRIPTION
concurrentConsumers	1	specifies the number of consumers to create to listen on the destination
disableReplyTo	false	set this endpoint to not do inOut messaging
durableSubscriptionName	null	explicitly set the name of a durable subscription (per JMS, must be used with clientId)
clientId	null	JMS client id for this endpoint. Must be unique
replyTo	null	specify the name of a destination to be included as a JMSReplyTo header on outgoing messages
selector	null	JMS selector to specify a predicate for what messages the broker should deliver to this endpoint consumer
timeToLive	null	time to live in milliseconds for the message as it travels through a broker (possible network of brokers)
transacted	false	specify whether to do send and receives in a transaction
acknowledgementModeName	AUTO_ACKNOWLEDGE	JMS acknowledgement mode
asyncConsumer	false	whether to process a message from a destination asynchronously. By default this is false
cacheLevelName	CACHE_AUTO	how to cache consumers, sessions, connections. e.g. CACHE_AUTO, CACHE_CONSUMER

FURTHER READING

For more information on the JMS component, see camel.apache.org/jms.html.

REST DSL

You're probably asking "why did you include the REST DSL in a Refcard about components"? Well, the REST DSL actually provides a great way to expose REST endpoints using a multitude of different backend components that implement the REST endpoint. So in a way, we're able to leverage any of the following Camel components (and others) to expose REST endpoints:

- camel-netty4-http
- camel-jetty
- camel-restlet
- camel-undertow
- camel-spark-rest

The REST DSL allows you to use a human-readable DSL to expose REST endpoints. To communicate with REST endpoints that live outside your Camel route, use the http component (or any of the above) directly in a to() clause. Here's an example of how to expose a REST API:

```
rest("/api")
  .get("/customer/{id}").to("direct:getCustomer")
  .post("/customer/{id}").to("direct:createCustomer")
  .put("/customer/{id}").to("direct:updateCustomer")
  .delete("/custome/{id}").to("direct:deleteCustomer")
```

Here we just use the DSL to expose the endpoints we want and send to the appropriate Camel to handle the integration (e.g., maybe we're integrating with a backend legacy SOAP service, or Mainframe, etc.). We could inline the route directly if we wanted:

```
rest("/api")
  .get("/customer/{id}")
  .route().to("ibatis:getCustomer").marshal()
  .json().endRest()
```

In the above examples, we explicitly marshaled to JSON, but we didn't have to do that. We can configure the REST DSL to automatically bind to XML or JSON. The default of this binding is disabled. See the configuration for more information.

DOCUMENTING YOUR API WITH SWAGGER

The Camel REST DSL automatically integrates with Swagger API documentation so you can have nicely described APIs for your clients (and further auto-generate clients, etc.). Here's how you'd do that. First, include the `camel-swagger-java` dependency:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
</dependency>
```

Then we need to add some docs to our API:

```
rest("/api")
  .get("/customer/{id}")
  .description("Retrieve an existing customer")
  .outType(Customer.class)
  .param()
  .name("id")
  .type(path).dataType("int").endParam()
  .to("direct:getCustomer")
```

CONFIGURATION

You can configure the binding mode and what component to use under the covers of the REST DSL with the REST DSL configuration options. Here's an example of configuring the DSL:

```
restConfiguration().component("netty4-http")
  .host("localhost").port(8080)
  .bindingMode(RestBindingMode.json)
```

Note, you'd need to verify the camel-netty4-http component/dependency was on the classpath for this to work properly:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
</dependency>
```

You can configure the underlying component directly with the

`.componentProperty()` configuration for the REST DSL. Some other useful configurations can be found:

OPTION	DEFAULT	DESCRIPTION
`scheme`	http	Specifies whether to use http or https
`contextPath`		What HTTP context path should be prefixed to the REST endpoints. Example "/foo"
`enableCORS`	false	Whether to enable CORS headers in HTTP response
`bindingMode`		auto, json, xml, json_xml

FURTHER READING

For more information on the REST DSL and associated components, see camel.apache.org/rest-dsl.html.

FILE COMPONENT

When reading or writing files (CSV, XML, fixed-format, for example) to or from the file system, use the Camel File component. File-based integrations are fairly common with legacy applications but can be tedious to interact with. With a file consumer endpoint, you can consume files from the file system and work with them as exchanges in your camel route. Conversely, you can also persist the content of your exchanges to disk.

READ FILES

When used in a "from" DSL clause, the file component will be used in "consumer" mode or "read":

```
from("file:/location/of/files?configs").log("\${body}")
```

OPTION	DEFAULT	DESCRIPTION
`delay`	500ms	how long (in milliseconds) to wait before polling the file system for the next file
`delete`	false	whether or not to delete the file after it has been processed
`doneFileName`	null	name of file that must exist before camel starts processing files from the folder
`exclude`	null	regex for file patterns to ignore
`filter`	null	a custom filter class used to filter out files/folders you don't want to process (specify as a bean name: #beanName)
`idempotent`	false	skip already process files following the Idempotent Consumer pattern
`include`	null	regex for file patterns to include
`move`	.camel	default place to move files after they've been processed
`noop`	false	if true, the file is not moved or deleted after it has been processed
`readLock`	markerFile	camel will only process files that it can get a read-lock for. specify a strategy for how to determine whether it has read-lock access
`readLockTimeout`	10000ms	timeout in milliseconds for how long to wait to acquire a read-lock
`recursive`	false	recurse through sub directories as well

THINGS TO WATCH OUT FOR

- Locking of files that are being processed (on reads) until **route** is complete (by default)
- Will ignore files that start with '.'
- By default will move processed files to '.camel' directory
- Moving/Deleting files will happen after routing

WRITE FILES

Writing to files is as simple as using the file component in a "to" clause:

```
from(<endpoint>).to(
  "file:/location/of/files?fileName=<filename>")
```

CONFIGURATION OPTIONS FOR WRITING FILES

OPTION	DESCRIPTION
`fileExist`	What to do if a file exists: Override, Append, Ignore, Fail
`fileName`	Name of file to be written. This can be a dynamic value determined by an expression (e.g., using the simple expression language. See camel.apache.org/file-language.html for more)
`tempFileName`	A temporary name given to the file as it's being written. Will change to real name when writing is finished

BIG FILES

Sometimes you need to process files that are too large to fit into memory, or would be too large to process efficiently if loaded into memory. A common approach to dealing with such files using the camel-file component is to stream them and process them in chunks. Generally the files are structured in such a way that it makes sense to process them in chunks, and we can leverage the power of camel's EIP processors to "split" a file into those chunks. Here's an example route:

```
from("file:/location/of/files").split(body()
  .tokenize("\\n")).streaming().log("\${body}").end()
```

This route splits on newlines in the file, streaming and processing one line at a time.

For more information on splitting large files, see davsclaus.com/2011/11/splitting-big-xml-files-with-apache.html.

HEADERS

Headers are automatically inserted (consumer) for use in your routes or can be set in your route and used by the file component (producer). Here are some of the headers that are set automatically by the file consumer endpoint:

- **CamelFileName** - name of file consumed as a relative location with the offset from the directory configured in the endpoint
- **CamelFileAbsolute** - a boolean that specifies whether the path is absolute or not
- **CamelFileAbsolutePath** - would be the absolute path if the CamelFileAbsolute header is true
- **CamelFilePath** - starting directory + relative filename
- **CamelFileRelativePath** - just the relative path
- **CamelFileParent** - just the parent path
- **CamelFileLength** - length of the file
- **CamelFileLastModified** - date of the last modified timestamp

Headers that, if set, will be used by the file producer endpoint:

- **CamelFileName** - name of the file to write to the output directory (usually an expression)
- **CamelFileNameProduced** - the actual filename produced (absolute file path)

FURTHER READING

For more information on the file component, see camel.apache.org/file2.html

SEDA AND DIRECT COMPONENT

You can still use messaging as a means of communicating with or between your routes without having to use an external messaging broker. Camel allows you to do in-memory messaging via synchronous or asynchronous queues. Use the "direct" component for synchronous processing and use the "seda" for asynchronous "staged" event-driven processing.

DIRECT

Use the direct component to break up routes using synchronous messaging:

```
from("direct:channelName").process(<process something>)
  .to("direct:outgoingChannel");
from("direct:outgoingChannel")
  .transform(<transform something>).to("jms:outgoingOrders")
```

SEDA

SEDA endpoints and their associated routes will be run in separate threads and process exchanges asynchronously. Although the pattern of usage is similar to the "direct" component, the semantics are quite different.

```
from("<any component>").choice()
  .when(header("accountType").endsWith("Consumer"))
  .to("seda:consumerAccounts")
  .when(header("accountType").endsWith("Business"))
  .to("seda:businessAccounts")
from("seda:consumerAccounts").process(<process logic>)
from("seda:businessAccounts").process(<process logic>)
```

You can set up the SEDA endpoint to use multiple threads to do its processing by adding the `concurrentConsumers` configuration:

```
from("seda:consumerAccounts?concurrentConsumers=20")
  .process(<process logic>)
```

Keep in mind that using SEDA (or any asynchronous endpoint) will behave differently in a transaction, i.e., the consumers of the SEDA queue will not participate in the transaction as they are in different threads.

COMMON CONFIGURATION OPTIONS

The "direct" component does not have any configuration options.

SEDA COMMONLY USED OPTIONS:

OPTION	DEFAULT	DESCRIPTION
`size`	Unbounded	Max capacity of the in-memory queue
`concurrentConsumers`	1	The number of concurrent threads that can process exchanges
`multipleConsumers`	false	Determine whether multiple consumers are allowed
`blockWhenFull`	false	Block a thread that tries to write to a full SEDA queue instead of throwing an Exception

FURTHER READING

For more information on the direct and SEDA component, see camel.apache.org/direct.html and camel.apache.org/seda.html respectively.

MOCK COMPONENT

Testing your routes is an important aspect to integration development and camel makes it easier with the mock component. You can write your Camel routes in JUnit or TestNG to use mock components. Then you can declare a set of expectations on the mock such as how many messages were processed, or that certain headers were present at an endpoint, and then run your route. At the completion of the route, you can verify that the intended expectations were met and fail the test if they were not.

MOCKS

You start by obtaining the mock endpoint from the route:

```
MockEndpoint mock = context.getEndpoint("mock:outgoing",
  MockEndpoint.class);
```

Next, you declare expectations. Methods that declare expectations start with "expect", for example:

```
mock.expectedMessageCount( 1 )
```

Then you run your route.

Finally, you assert that the declared expectations were met:

```
mock.assertIsSatisfied()
```

Here are a few of the methods you can use to set up expectations and others for verifying your mock endpoints (see camel.apache.org/mock.html for more)

Expectation Methods:

- `expectedMessageCount(int)`
- `expectedMinimumMessageCount(int)`
- `expectedBodiesReceived()`
- `expectedHeadersReceived()`
- `expectsNoDuplicate(Expression)`

AssertionMethods:

- `assertIsSatisfied()`
- `assertIsNotSatisfied()`

MOCKING EXISTING ENDPOINTS

Sometimes you'll have routes with live endpoints that you cannot change (or don't want to change) for test. For such

routes, you can insert mocks where the live mocks are to do some testing.

```
RouteDefinition route = context.getRouteDefinition("advice");
route.adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
        mockEndpoints(<pattern>);
    }
});
```

`<pattern>` allows you to specify which endpoints to mock. For example, to mock out only the JMS components, you would do `mockEndpoints("jms*")`. To mock all endpoints, leave off the pattern completely.

NOTE: inserting mocks at the location of the endpoints does not replace the endpoints, i.e., the live endpoints will still exist. Updated for Camel 2.10, you can now skip sending the exchange to the live endpoint:

```
RouteDefinition route = context.getRouteDefinition("advice");
route.adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
        mockEndpointsAndSkip(<pattern>);
    }
});
```

FURTHER READING

For more information on the mock component, see http://camel.apache.org/mock.html

ABOUT THE AUTHOR

CHRISTIAN POSTA (@christianposta) is a Principal Middleware Specialist/Architect at Red Hat and well known for being an author (Microservices for Java Developers, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast and committer on Apache ActiveMQ, Apache Camel, Fabric8 and others. Christian has spent time at web-scale companies and now helps companies creating and deploying large-scale distributed architectures - many of what are now called Microservices based. He enjoys mentoring, training and leading teams to be successful with distributed systems concepts, microservices, devops, and cloud-native application design.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

BROUGHT TO YOU IN PARTNERSHIP WITH
redhat