19

Camel tooling

This chapters covers

- Eclipse-based graphical Camel editor from JBoss
- IDEA plugin for Camel code editor
- Maven plugin for source code validation
- Debugging Camel routes from Eclipse or a web browser
- hawtio web console with Camel plugin
- Camel Catalog—information archive for Camel tools

This chapter walks you through some of the most powerful Camel tools that you can find on the internet. The first tool covered is an Eclipse plugin that provides a graphical editor that enables users to design Camel routes using a drag-and-drop style of editing. For IDEA users, we have an alternative tool that allows in-place code assistance for editing Camel endpoints in a *type-safe* manner. We'll show you a Maven tool that's capable of scanning your source code and reporting any Camel configuration mistakes. You can also find tools to debug Camel applications by stepping through the routes and inspecting the Camel messages.

The last tool we demonstrate can inspect running Camel applications and help you visualize the Camel routes with real-time metrics. It also provides management functionality that allows you to control your Camel applications—for example, starting and stopping routes. You'll learn the secret of the Camel Catalog, which exposes a wealth of information about a Camel release that allows tooling to know everything there is to know about all the components, data formats, EIPs, and so on.

Let's get started with the traditional developer tools that build Camel routes using a graphical editor.

19.1 Camel editors

Building Camel tools for graphical Camel development is a large task and could explain why we see this kind of tool only from commercial vendors such as Red Hat and Talend. This chapter covers the tools we know best: those from Red Hat JBoss, Apache Camel, and hawtio projects.

19.1.1 JBoss Fuse Tooling

The JBoss Fuse Tooling (formerly known as Fuse IDE—see http://tools.jboss.org/features/fusetools.html) is a plugin that can be installed in Eclipse and that provides the following Camel development capabilities:

- Graphical route editor
- Graphical data mapper
- Type-safe endpoint editor
- Integrated Camel tracer and debugger
- Easy deployment to application servers

NO VENDOR LOCK-IN

The tools covered in this chapter are all open source and have no vendor lock-in. You can start to use a tool and can stop at any point. These tools all operate on your project's source code as is and don't use tooling-specific metadata to be saved with your source code.

We'll now look at some of the features this tool provides. You can find the sample project in the chapter19/eclipse-camel-editor directory.

Graphical Camel editor

The Camel project that's loaded into Eclipse in figure 19.1 is a simple Camel route from a Spring XML file. You can click each EIP in the route and edit its properties in a type-safe manner, as shown in figure 19.2.

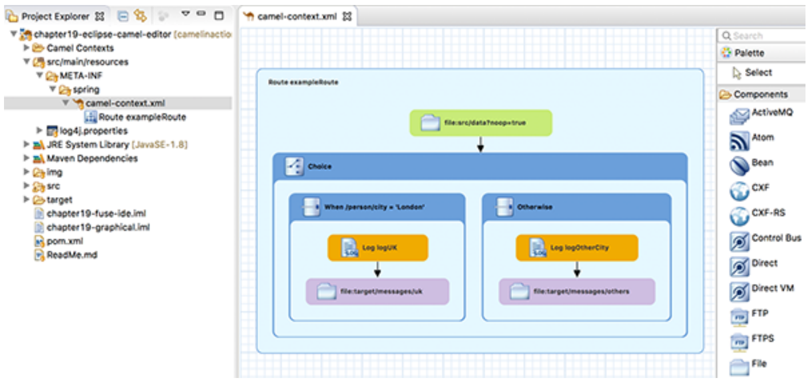
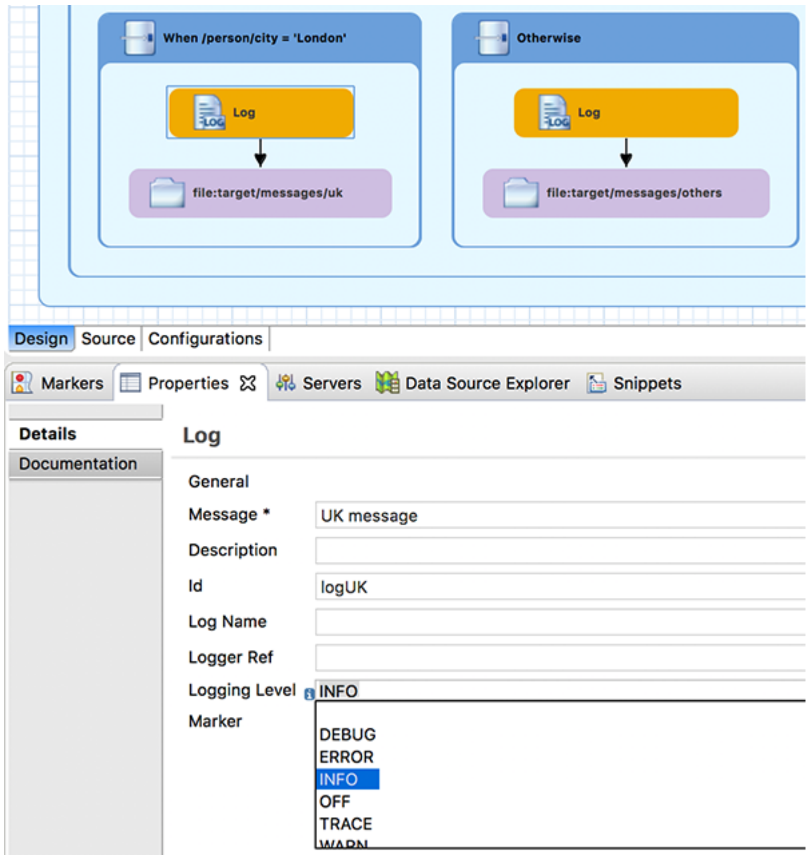


Figure 19.1 Graphical Camel editor showing the Camel route using EIP icons and depicting their connections. Clicking an EIP allows you to configure the EIP in the properties panel in a type-safe way.



Log

General

Message *	UK message
Description	
Id	logUK
Log Name	
Logger Ref	
Logging Level	INFO
Marker	

DEBUG
ERROR
INFO
OFF
TRACE
WARN

Figure 19.2 Type-safe editing the selected Log EIP from the route in the panel underneath. The drop-down panel for the logging level shows the possible values accepted.

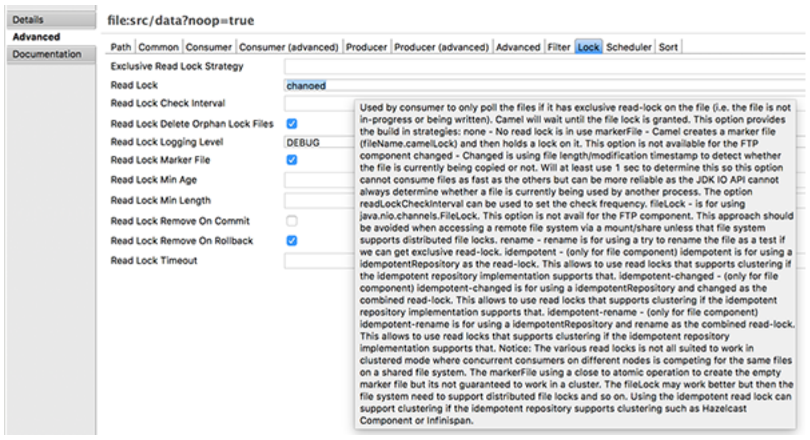
In figure 19.2, we've selected the Log EIP; the tiny blue box around its borders indicates the selection. In the properties panel underneath, the possible options from the Log EIP are shown with its current values. If you want to change the logging level, you can click the drop-down, which lists the possible values as shown in the screenshot.

Type-safe endpoint editor

The tooling also offers type-safe editing of all the Camel endpoints. The example from chapter19/eclipse-camel-editor is a Camel route that consumes files from the following endpoint:

```
<from uri="file:src/data?noop=true"/>
```

Now, suppose you want to configure a read lock on this endpoint. Using the tool, you have all the options available presented in the editor. By hovering the mouse pointer over an option's label, you can access the documentation as a tooltip. Figure 19.3 shows how you're changing the endpoint to use the changed read lock with a 5-second interval.



file:src/data?noop=true

Exclusive Read Lock Strategy	
Read Lock	chanoed
Read Lock Check Interval	
Read Lock Delete Orphan Lock Files	✓
Read Lock Logging Level	DEBUG
Read Lock Marker File	✓
Read Lock Min Age	
Read Lock Min Length	
Read Lock Remove On Commit	☐
Read Lock Remove On Rollback	✓
Read Lock Timeout	

Used by consumer to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the build in strategies: none - No read lock is in use markerFile - Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is not available for the FTP component changed - Changed is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec to determine this so this option cannot consume files as fast as the others but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option readLockCheckInterval can be used to set the check frequency. fileLock - is for using java.nio.channels.FileLock. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. rename - rename is for using a try to rename the file as a test if we can get exclusive read-lock. idempotent - (only for file component) idempotent is for using a idempotentRepository as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-changed - (only for file component) idempotent-changed is for using a idempotentRepository and changed as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-rename - (only for file component) idempotent-rename is for using a idempotentRepository and rename as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. Notice: The various read locks is not all suited to work in clustered mode where concurrent consumers on different nodes is competing for the same files on a shared file system. The markerFile using a close to atomic operation to create the empty marker file but its not guaranteed to work in a cluster. The fileLock may work better but then the file system need to support distributed file locks and so on. Using the idempotent read lock can support clustering if the idempotent repository supports clustering such as Hazelcast Component or Infinispan.

Figure 19.3 Editing the Camel endpoint in a type-safe manner using the tooling. Documentation is available for every option as tooltips. The screenshot shows the extensive documentation for the read lock option.

After editing, the endpoint is updated accordingly:

```
<from uri="file:src/data?
        noop=true&amp;readLock=changed&amp;readLockCheckInterval=5000"/>
```

Integrated Camel tracer and debugger

The tool provides an Eclipse-based Camel debugger. To try this in action, right-click one or more EIPs in the graphical editor and click the Set Breakpoint menu item. The EIP icon should show a red dot indicating that the breakpoint is set. Then right-click the camel-context.xml file, as shown in figure 19.4, and choose Debug As > Local Camel Context (Without Tests).

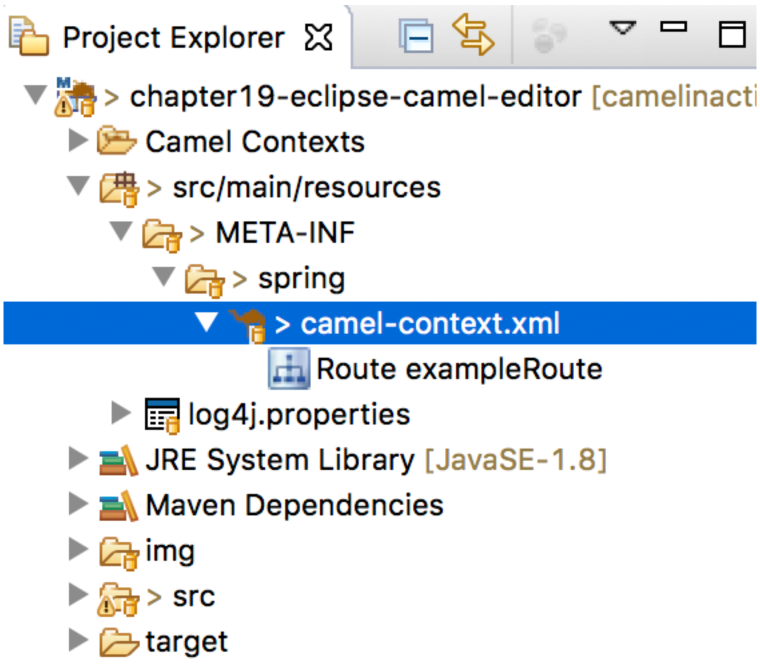
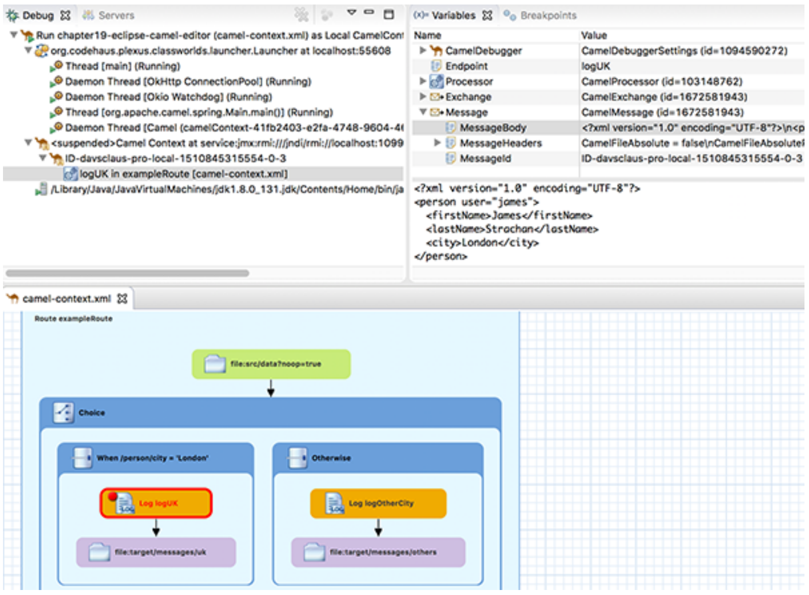

```
Project Explorer
▼ > chapter19-eclipse-camel-editor [camelinacti
    ▶ Camel Contexts
    ▼ > src/main/resources
        ▼ > META-INF
            ▼ > spring
                ▼ > camel-context.xml
                      Route exampleRoute
        ▶ log4j.properties
    ▶ JRE System Library [JavaSE-1.8]
    ▶ Maven Dependencies
    ▶ img
    ▶ > src
    ▶ target
```

Figure 19.4 To debug a Camel application, right-click the XML file containing `< camelContext >` (outlined here) and choose Debug As > Local Camel Context (Without Tests) to start the application in debug mode.

Eclipse then switches to debug perspective, and when a message hits the breakpoint, the EIP icon will be highlighted with a solid red border, as shown in figure 19.5.


```
Debug  Servers
▼ Run chapter19-eclipse-camel-editor (camel-context.xml) as Local CamelCon
  ▼ org.codehaus.plexus.classworlds.launcher.Launcher at localhost:55608
      Thread [main] (Running)
      Daemon Thread [OkHttp ConnectionPool] (Running)
      Daemon Thread [Okio Watchdog] (Running)
      Thread [org.apache.camel.spring.Main.main()] (Running)
      Daemon Thread [Camel (camelContext-41fb2403-e2fa-4748-9604-4
  ▼ <suspended>Camel Context at service:jmx:rmi:///jndi/rmi://localhost:1099
    ▼ ID-davsclaus-pro-local-1510845315554-0-3
        logUK in exampleRoute [camel-context.xml]
    /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/ja
```

Variables Breakpoints

Name	Value
▶ CamelDebugger	CamelDebuggerSettings (id=1094590272)
Endpoint	logUK
▶ Processor	CamelProcessor (id=103148762)
▶ Exchange	CamelExchange (id=1672581943)
▼ Message	CamelMessage (id=1672581943)
MessageBody	<?xml version="1.0" encoding="UTF-8"?>\n<p
▶ MessageHeaders	CamelFileAbsolute = false\nCamelFileAbsolute
MessageId	ID-davsclaus-pro-local-1510845315554-0-3

```
<?xml version="1.0" encoding="UTF-8"?>
<person user="james">
  <firstName>James</firstName>
  <lastName>Strachan</lastName>
  <city>London</city>
</person>
```

Figure 19.5 Debugging a Camel application in Eclipse: a Camel message hits the breakpoint highlighted by an outline. In the top-right corner, the Variables tab shows the content of the message. We've selected the message body, shown underneath, and in this example is an XML message showing that James Strachan lives in London (he lived there while creating Apache Camel).

You can use the tooling to inspect the message content, such as the message body and its headers. You can even update the message body from the debugger, such as changing the content of the message body or headers. When you're ready to continue, click either the Resume or Step Over button. The latter allows you to single-step through the route, stopping at the next EIP, as shown in figure 19.6.

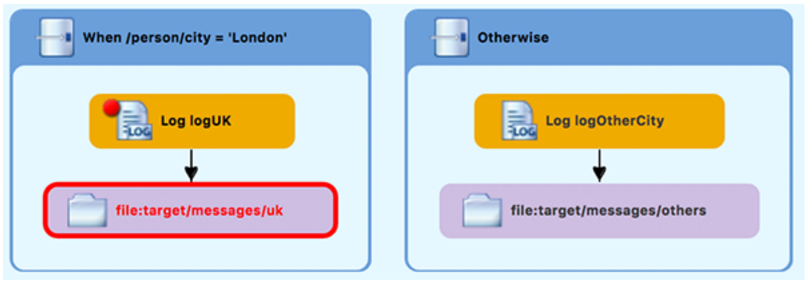


Figure 19.6 The breakpoint was set on the Log EIP indicated by the dot in the upper left corner. When the breakpoint was hit, we clicked the step over button (F6), and the debugger ran to the next EIP, which is the file endpoint indicated by the square.

Table 19.1 provides a summary of JBoss Fuse Tooling.

Table 19.1 The good and not so good about JBoss Fuse Tooling

Pros	Cons
Graphical route editor (only works with XML DSL)	Works only with Eclipse.
Powerful Camel debugger natively integrated in Eclipse	The graphical Camel debugger only works with XML DSL.
Type-safe Camel endpoint editor (currently only works with XML DSL)	
Documentation for all components and EIPs included	
Full-time engineers working on the tool	
100% open source and Eclipse licensed	

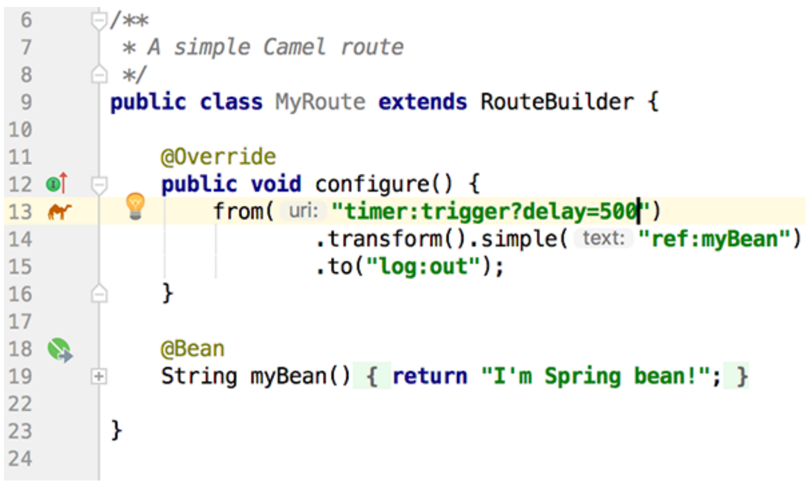
Let's look at another Camel editor that works in a different way.

19.1.2 Apache Camel IDEA plugin

IntelliJ IDEA users shouldn't miss out on the excellent Apache Camel IDEA plugin. This plugin adds Camel awareness to the IDEA editor to bring great power and enjoyment for developers to work on Camel code. The plugin is best explained by just trying it out in action, so let's do that.

To install the plugin, start IDEA and open its preferences. From within Preferences, select Plugins and click Browse Repositories. In the search field, type `Apache Camel` to find the Apache Camel IDEA plugin, which you then select. Click Install. After installation, you may need to restart IDEA.

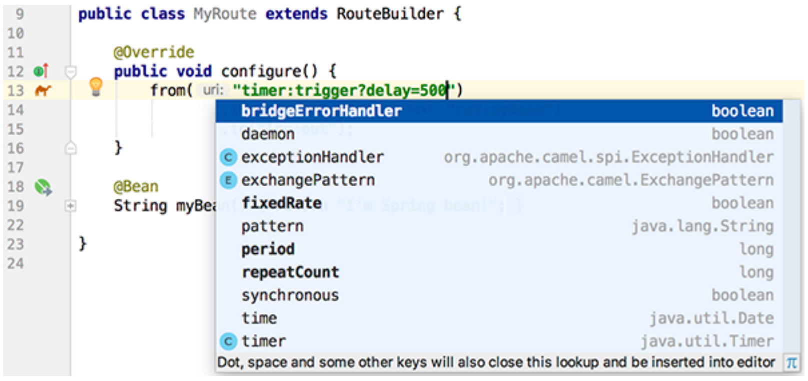
From the book's source code, you can open the chapter19/camel-idea-editor example in IDEA. Then open the Camel route file, MyRoute.java, and you should notice the subtle Camel plugin appearance in the code editor by small Camel icons in the gutter, as illustrated in figure 19.7.



```
/**
 * A simple Camel route
 */
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() {
        from( uri: "timer:trigger?delay=500")
                .transform().simple( text: "ref:myBean")
                .to("log:out");
    }

    @Bean
    String myBean() { return "I'm Spring bean!"; }

}
```

Figure 19.7 In the gutter, there's a Camel icon at the beginning of every Camel route detected in the source code.

The plugin can, of course, do a lot more than show a little Camel icon. For example, position the cursor as shown in figure 19.7, where the cursor is at the end of the timer endpoint. Then press Ctrl-space, which activates IDEA smart completion. Because the plugin is installed, it knows this is a Camel endpoint and therefore presents the user with a list of possible endpoint options you can use to configure the endpoint, as shown in figure 19.8.



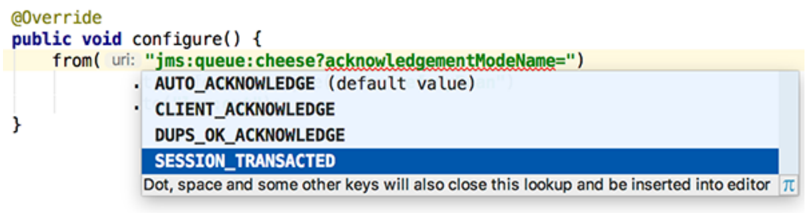
```
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() {
        from( uri: "timer:trigger?delay=500")

    }

    @Bean
    String myBe
}
```

```
bridgeErrorHandler                                          boolean
daemon                                                      boolean
exceptionHandler        org.apache.camel.spi.ExceptionHandler
exchangePattern            org.apache.camel.ExchangePattern
fixedRate                                                   boolean
pattern                                        java.lang.String
period                                                         long
repeatCount                                                    long
synchronous                                                 boolean
time                                             java.util.Date
timer                                           java.util.Timer
Dot, space and some other keys will also close this lookup and be inserted into editor
```

Figure 19.8 Pressing Ctrl-space shows a list of possible options that can be used to configure the endpoint. The options highlighted in bold are the most commonly used options.

The list works as if you're doing smart completion on regular Java code. You can use *type ahead*, and by pressing Ctrl-J while the list is present, another window is shown on the side that presents documentation for the currently selected option in the list. This gives you information at your fingertips; you don't have to open a web browser and find the Camel component documentation. And if you want to go there, press Ctrl-F1, and IDEA will open the web page for the selected component for you in the browser.

You can also use Ctrl-space when configuring the values of the endpoint options. In figure 19.9, we're configuring the acknowledge mode option on a JMS endpoint.



```
@Override
public void configure() {
    from( uri: "jms:queue:cheese?acknowledgementModeName=")

}
```

```
AUTO_ACKNOWLEDGE (default value)
CLIENT_ACKNOWLEDGE
DUPS_OK_ACKNOWLEDGE
SESSION_TRANSACTED
Dot, space and some other keys will also close this lookup and be inserted into editor
```

Figure 19.9 Press Ctrl-space while editing the value of an endpoint option to provide a list of possible choices if the option is an enum-based type, such as the JMS acknowledge-mode option.

The plugin has other noteworthy functionality such as live validation in the source code, where any misconfigured Camel endpoint or Simple language expression will be highlighted with a red underline.

If that's too much for you, the plugin can be configured from the preferences, where you find Apache Camel in the Languages & Frameworks group. The plugin works with both the Java and XML DSLs. Press Ctrl-space anywhere in your Camel code, and the plugin is there to help you. One last thing to highlight as well is that the plugin works with any Apache Camel release—it's capable of downloading the catalog information from newer or older versions on the fly (we cover the Camel Catalog in section 19.2).

A summary of the plugin's pros and cons is listed in table 19.2.

Table 19.2 The good and not so good about the Apache Camel IDEA tooling

Pros	Cons
Integrates natively with IDEA in the code editor Type-safe and real-time validation as you type in the code editor Supports Java, and XML DSLs 100% open source and Apache licensed	Works only on IDEA No graphical visualization of Camel routes No commercial support or paid work for engineers, meaning development effort is done in spare time

Let's step outside IDEA and other IDE editors and explore a great little tool that comes out of the box with Apache Camel. It's the Apache Camel Maven plugin that's capable of scanning your source code and reporting invalid Camel endpoints.

19.1.3 Camel validation using Maven

You may have experienced starting your Camel application and having it fail because one or more Camel endpoints were misconfigured due to an invalid option. It's both a blessing and a curse with Camel that endpoints are so quick and easy to configure using URI parameters. The endpoint URI is a `String` type, such as the following file endpoint:

```
file:src/data?noop=true&recursive=true
```

Now suppose you mistype the recursive option:

```
file:src/data?noop=true&recusive=true
```

Then when starting this application, Camel will report an error:

```
Caused by: org.apache.camel.ResolveEndpointFailedException: Failed
to resolve endpoint: file://src/data?noop=true&recusive=true due to: The
are 1 parameters that couldn't be set on the endpoint. Check the uri if
parameters are spelt correctly and that they are properties of the endpo
Unknown parameters=[{recusive=true}]
```

Wouldn’t it be great if you could validate those Camel URIs before you run your Camel application? Great news: the existing Camel Maven plugin from Apache Camel is capable of doing that.

THE STORY OF THE FABRIC8 CAMEL TOOLING AND THE IDEA PLUGIN

This validation functionality that we’re about to cover was first developed as an experiment at the fabric8 project (https://fabric8.io). After more than a year in active development at fabric8, the plugin code was stable and was donated to Apache Camel for inclusion from Camel 2.19 onward.

At the same time, the fabric8 team experimented with a Camel editor that was based on JBoss Forge and allows you to install the tooling as plugins to Eclipse, IDEA, and NetBeans. This tool was capable of type-safe editing of your Camel endpoints and EIP patterns in Java and XML DSL.

But we wanted tighter integration with the IDE editors than what JBoss Forge provides. On the evening of December 23, 2016, Claus created an experimental prototype of a Camel plugin for IDEA. This prototype was able to show a list of possible Camel options you could configure on the Camel endpoints. It provided type-safe code completions directly in the IDEA source code editor (as you have today for Java code). Over the next couple of months, this prototype matured into the Apache Camel IDEA plugin. The plugin code is hosted at GitHub at https://github.com/camel-idea-plugin/camel-idea-plugin.

The book’s accompanying source code contains an example in the chapter19/camel-maven-validate directory. From this directory, you can run the following command from the command line:

```
mvn camel:validate
```

Running this command should report two errors:

```
[INFO] --- camel-maven-plugin:2.20.1:validate (default)
[INFO] Detected Camel version used in project: 2.20.1
[INFO] Validating using Camel version: 2.20.1
[WARNING] Endpoint validation error at: camelinaction.MyRouteBuilder.con

     file:src/data?noop=true&recusive=true

         recusive    Unknown option. Did you mean: [recursive]


[WARNING] Endpoint validation error at: camelinaction.MyRouteBuilder.con

     log:uk?showall=true

          showall    Unknown option. Did you mean: [showAll, showOut, sh


[WARNING] Endpoint validation error: (2 = passed, 2 = invalid,
 0 = incapable, 0 = unknown components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] Duplicate route id validation success (0 = ids)
```

As you can see, the source code has two errors. The first is the mistype of the `recursive` option. The tooling gives you suggestions for the option. The second error is in a log endpoint, where we've configured the `showAll` option using a lowercase *a* in *all*. The correct option is spelled `showAll`.

Adding the plugin to your project

To use the validation plugin in your Camel projects, add the following to the pom.xml file:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
</plugin>
```

Then you can run the validation goal using the following:

```
mvn camel:validate
```

You can also enable the plugin to automatically run as part of the build, but configure the plugin to run the validate goal as part of the process-classes phase:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
  <executions>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

By default, the plugin validates the source code only in src/main, so if you want to also validate the unit-test source code, you need to turn this on by configuring `includeTest=true`, as highlighted here in bold:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
  <version>2.20.1</version>
  <executions>
    <configuration>
      <includeTest>true</includeTest>
    </configuration>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The plugin has options you can configure, and you can find details in the Camel plugin documentation at: https://github.com/apache/camel/blob/master/tooling/maven/camel-maven-plugin/src/main/docs/camel-maven-plugin.adoc.

TIP The Camel Maven Plugin will include a new functionality (route-coverage goal) in Camel 2.21 that, after running your unit tests, allows you to generate route coverage reports. This can be used to identify whether you have tests that cover all parts of your Camel routes.

All the tools we've covered so far—JBoss Fuse Tooling, the IDEA Apache Camel plugin, and the Camel Maven plugin—use the Camel Catalog to obtain extensive information about what an Apache Camel release includes.

19.2 Camel Catalog: the information goldmine

Starting from Camel version 2.17, each release includes a catalog with all sorts of information about what's included in the release. The catalog is shipped in an independent standalone camel-catalog JAR containing the following information:

1. List of all components, data formats, languages, EIPs, and everything else in the release
2. Curated lists for Spring Boot and Apache Karaf runtime
3. JSON schema with extensive details for every option
4. Human-readable documentation for every option
5. Categorization of options (for example, find all database components)
6. XML schema for the XML DSL
7. Maven Archetype Catalog of all the Camel archetypes
8. Entire website documentation as ASCII doc and HTML files
9. Validator for Camel endpoint and Simple language
10. API to create Camel endpoint URLs
11. Java, JMX, and REST API

The catalog provides a wealth of information that tooling can tap into and use. For example, JBoss Fuse Tooling uses the catalog in the graphical editor to know all details about every EIP and component Camel supports. The IDEA Camel tooling does the same thing. For example, the Maven validation plugin covered in section 19.1.3 uses the catalog during validation of all the Camel endpoints found while scanning the source code.

The camel-catalog JAR includes the information using the following directory layout:

```
org
└── apache
    └── camel
        └── catalog
            ├── archetypes                (Maven archetype catalog)
            ├── components                (JSON schema)
            ├── dataformats               (JSON schema)
            ├── docs                      (Ascii and HTML)
            ├── languages                 (JSON schema)
            ├── models                    (JSON schema)
            ├── others                    (JSON schema)
            └── schemas                   (XML schema)
```

Each directory contains files with the information. Every Camel component is included as JSON schema files in the components directory. For example, the Timer component is included in the file timer.json, as shown in the following listing.

Listing 19.1 JSON schema of the Timer component from camel-catalog

```
{
 "component": {
    "kind": "component",
    "scheme": "timer",  ❶
```

❶

The name of the component

```
    "syntax": "timer:timerName",    (2)
```

(2)

URI syntax (without parameters)

```
    "title": "Timer",
    "description": "The timer component is used for generating message
                    exchanges when a timer fires.",
    "label": "core,scheduling",    (3)
```

(3)

Labels are used for categorizing the component.

```
    "deprecated": false,
    "async": false,
    "consumerOnly": true,    (4)
```

(4)

This component can be used only as a consumer (for example, from).

```
    "firstVersion": "1.0.0",    (5)
```

(5)

First version when this component was included

```
    "javaType": "org.apache.camel.component.timer.TimerComponent",
    "groupId": "org.apache.camel",    (6)
```

(6)

The Maven coordinate for the JAR that contains the component

```
    "artifactId": "camel-core", (6)
    "version": "2.20.1" (6)
  },
  "componentProperties": { (7)
```

(7)

Component-level options

```
  },
  "properties": { (8)
```

(8)

Endpoint-level options (text abbreviated)

```
      "timerName": { "kind": "path", "group": "consumer", "required"
      "bridgeErrorHandler": { "kind": "parameter", "group": "consume
      "delay": { "kind": "parameter", "group": "consumer", "type": "
      "fixedRate": { "kind": "parameter", "group": "consumer", "type
      "period": { "kind": "parameter", "group": "consumer", "type":
      "repeatCount": { "kind": "parameter", "group": "consumer", "ty
      "exceptionHandler": { "kind": "parameter", "group": "consumer
      "daemon": { "kind": "parameter", "group": "advanced", "label":
      "exchangePattern": { "kind": "parameter", "group": "advanced",
      "pattern": { "kind": "parameter", "group": "advanced", "label"
      "synchronous": { "kind": "parameter", "group": "advanced", "la
      "time": { "kind": "parameter", "group": "advanced", "label": "
      "timer": { "kind": "parameter", "group": "advanced", "label":
    }
  }
```

The JSON schema is divided into three parts:

- `component` —General information about the component
- `componentProperties` —Options you can configure on the component itself
- `properties` —Options you can configure on the endpoints

In the `component` section, you find the component name ❶ and syntax ❷. Then labels are used to categorize the component ❸. This component has the labels `core` and `scheduling`. In Camel, a component can be used as a consumer, producer, or both. The JSON schema indicates whether the component can be only a consumer or producer using `consumerOnly` ❹ or `producerOnly`, respectively. If neither of those entries exists, the component can be used as both. You can also see when a component was added to Apache Camel, as stated in the first version ❺ attribute. In this example, we can see the Timer component was included in the first release of Apache Camel. There's also information about which JAR file contains the component as Maven coordinates ❻. The last two parts document each option you can configure on the component level ❼ and endpoint level ❽. Each entry in these parts is structured the same way. The Timer component has only endpoint-level options. Each option includes a lot of details that have been abbreviated in the listing. The following is a snippet of the delay option in its entirety:

```
"period": { "kind": "parameter", "displayName": "Period", "group":
"consumer", "type": "integer", "javaType": "long", "deprecated":
false, "secret": false, "defaultValue": 1000, "description": "If
greater than 0 generate periodic events every period milliseconds. The
default value is 1000. You can also specify time values using units such
60s (60 seconds) 5m30s (5 minutes and 30 seconds) and 1h (1 hour)." },
```

Let's break down the option and explain each detail, as listed in table 19.3.

Table 19.3 Breakdown of the component and endpoint options from the JSON schema

Option	Description
`period`	The name of the option.
`kind`	Kind of option, can either be path or parameter. A path is an option in the URI context path. Parameter denotes a URI query parameter.
`displayName`	The name of the option intended for displaying in UIs.
`group`	Group is a single overall name that categorizes the option.
`type`	JSON schema type of the option.
`javaType`	Java type of the option.
`deprecated`	Whether the option has been deprecated.
`secret`	Whether the option is sensitive, such as a username or password.
`defaultValue`	The default value of the option (optional).
`description`	Human-readable description.

The catalog also includes JSON schema files for every EIP, data format, and language in the Camel release. These schemas are structured in similar way as the components, as shown in listing 19.1.

SUBJECT OF CHANGE

The Camel Catalog (camel-catalog) may include more details in future Camel releases. Therefore, the JSON schema may change and include additional information. But the basic structure as shown in listing 19.1 is expected to stay.

If you want to write custom Camel tooling, the camel-catalog is a great source of information.

Let's move on to the web and look at Camel tooling with hawtio.

19.3 hawtio: a web console for Camel and Java applications

What is hawtio (http://hawt.io) and what does it do? Here's what hawtio says:

> *It's a pluggable management console for Java stuff that supports any kind of JVM, any kind of container (Tomcat, Jetty, Spring Boot, Karaf, JBoss, WildFly, fabric8, etc.), and any kind of Java technology and middleware.*
>
> *—http://hawt.io/faq/index.html*

If you've read this book in chronological order, you may remember that we talked about hawtio in previous chapters (such as chapter 16, where we briefly covered using hawtio to manage Camel applications). In this section, we'll take a deeper look at hawtio and unlock some of the features it provides that are relevant to Camel.

19.3.1 Understanding hawtio functionality

hawtio provides the following Camel features out of the box:

- Lists all running Camel applications in the JVM
- Details information of each Camel application such as version number and runtime statistics
- Lists all routes in each Camel application and their runtime statistics
- Manages the lifecycle of all Camel applications and their routes, so you can restart, stop, pause, or resume them

- Provides graphical representation of the running routes along with real-time metrics
- Provides embedded documentation of in-use components, endpoints, and EIPs
- Enables live tracing and debugging of running routes
- Profiles the running routes with real-time runtime statistics and details specified per processor
- Updates routes using a text-based XML editor (not persistent update)
- Enables browsing and sending messages to Camel endpoints

hawtio provides many Camel features out of the box. The most prominent Camel feature of hawtio is likely the Camel route diagram, shown in figure 19.10.

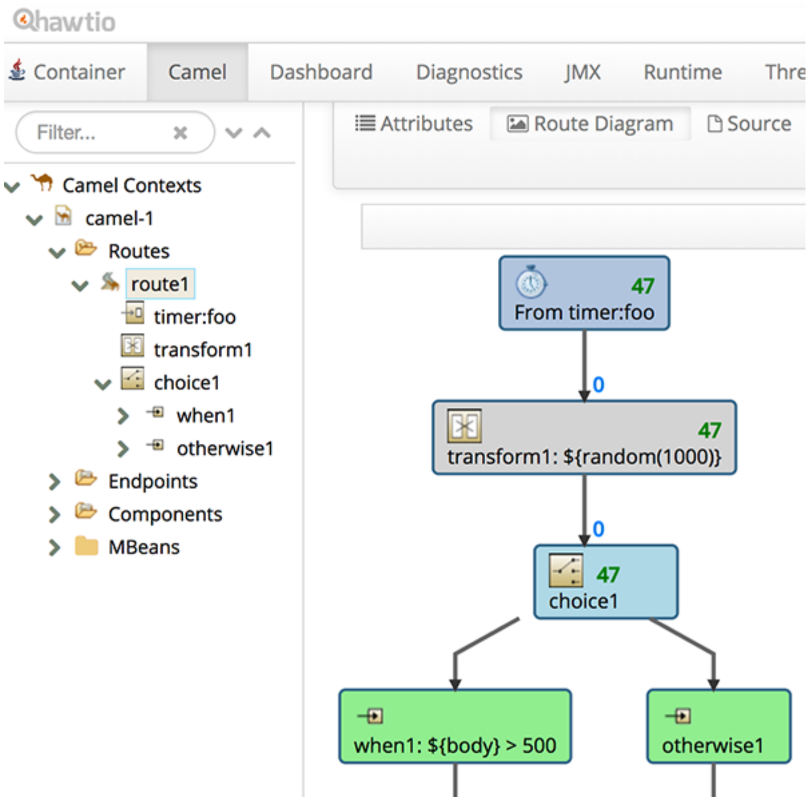


Figure 19.10 Visual representation of the running Camel routes using the hawtio web console. The numbers (47) are the messages processed, and the numbers (0) are the number of messages that are in flight (currently being processed). The numbers update in real time.

INSTALLING HAWTIO IN KARAF

You can install hawtio in Apache Karaf/ServiceMix using the following commands:

```
karaf@root()> feature:repo-add hawtio 1.5.6
Adding feature url mvn:io.hawt/hawtio-karaf/1.5.6/xml/features
karaf@root()> feature:install hawtio
```

Then you can access hawtio in the web browser at http://localhost:8181/hawtio and log in with karaf/karaf (default username and password).

hawtio includes a route debugger, so let's try this in action.

19.3.2 Debugging Camel routes using hawtio

When developing Camel routes, debugging at the route level can be handy. hawtio comes with a route debugger, which you can run from a web browser. Let's jump straight into action and debug the following route, which has been written in Java DSL:

```
from("timer:foo?period=5000")
  .transform(simple("${random(1000)}"))
  .choice()
    .when(simple("${body} > 500"))
    .log("High number ${body}")
    .to("mock:high")
  .otherwise()
    .log("Low number ${body}")
    .to("mock:low");
```

The route starts from a timer that triggers every fifth second. Then a random number between 0 and 1,000 is generated as the message body. Depending on whether the number is a high or low number, it's routed accordingly, using the Content-Based Router EIP. You can run this example, provided in the chapter19/hawtio-debug directory, from Maven using the following:

```
mvn compile exec:java
```

Because you want to debug the route, you can start the example with hawtio embedded by running the following Maven goal:

```
mvn compile hawtio:run
```

Then hawtio is started from the Maven plugin, which then starts the Camel application using a Java main class. The Java main class is configured in the Maven pom.xml file, as highlighted here:

```
<plugin>
  <groupId>io.hawt</groupId>
  <artifactId>hawtio-maven-plugin</artifactId>
  <version>1.5.6</version>
  <configuration>
    <mainClass>camelinaction.MainApp</mainClass>
  </configuration>
</plugin>
```

When you run the `hawtio:run` goal, hawtio and the Camel application are started together in the same JVM. And after a little while, hawtio opens automatically in your web browser. You can then use the web browser to see what happens at runtime in the JVM where Camel is running. For example, you can see real-time metrics of the number of messages being processed.

Okay, let's get on with it. How do you debug a Camel route using hawtio? You need to select the route you want to debug in the tree on the left side, and because there's only one route in this example, you should select `route1`. In the center of the screen, click the Debug button and then click the Start Debugging button, which should show the route. You can then double-click an EIP to add a breakpoint—for example, on the `choice1` EIP. After a little while, the EIP should display a blue ball, which indicates that a breakpoint was hit. At the bottom of the screen, the message is listed, and you can click it to show its message body and headers. After doing this, you should have a screen similar to figure 19.11.

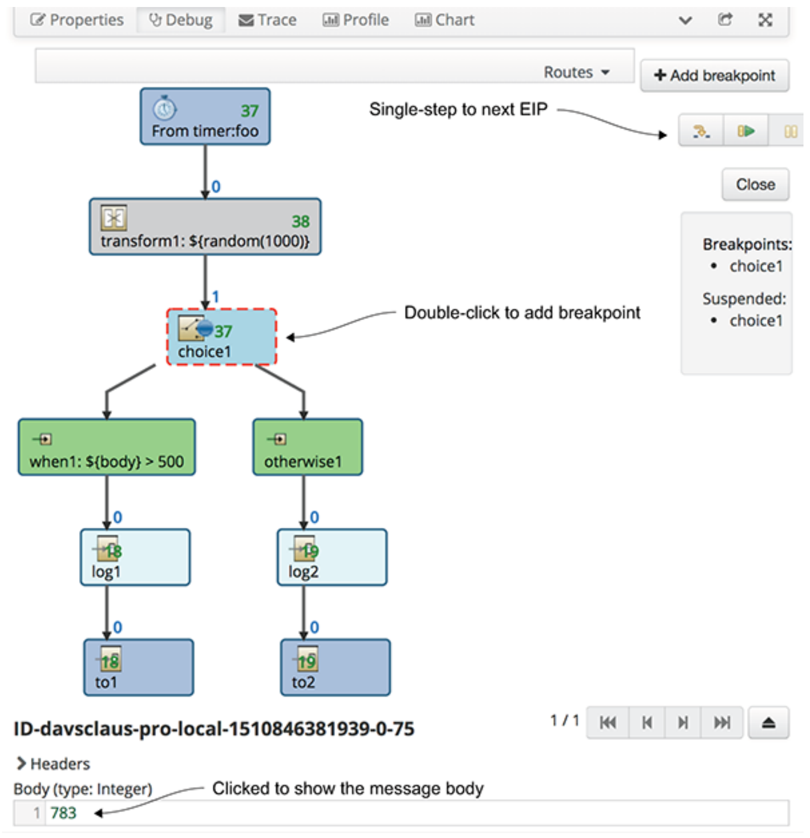


Figure 19.11 Using hawtio to debug a Camel route. Double-click an EIP to set a breakpoint. When a breakpoint is hit, the ball color changes to blue, and at the bottom of the screen you can see the current message body and its headers. At the top right, you can click the Single Step button to continue routing and pause the debugger at the next EIP.

In figure 19.11, the debugger has paused at the Content-Based Router EIP (marked *Choice*), which is indicated by the blue ball. At the bottom of the screen, you can see that the message body has the value of 783 (you need to click the message to expand it and show the content of the message).

When you click the Single Step button at the top-right corner, the message is routed and paused at the next EIP, which is the branch on the left-hand side in the Content-Based Router (marked *When*) because 783 is higher than 500. When you're finished with the debugger, remember to click the Close button.

DEBUGGING API

The debugger in hawtio and the Eclipse-based JBoss Fuse Tooling both use the same debugging API from camel-core. This API is available to custom tooling from JMX from the `ManagedBacklogDebugger` MBean, which you can find under the tracer tree in the Camel JMX tree. This MBean has a rich set of debugging operations to control breakpoints, run in single-step mode, update the message body/headers, and more.

You may not always be able to run your Camel application with hawtio embedded. Therefore, you can run hawtio in another JVM and connect from hawtio to your existing Camel applications running in another JVM.

Connecting hawtio to existing running JVM

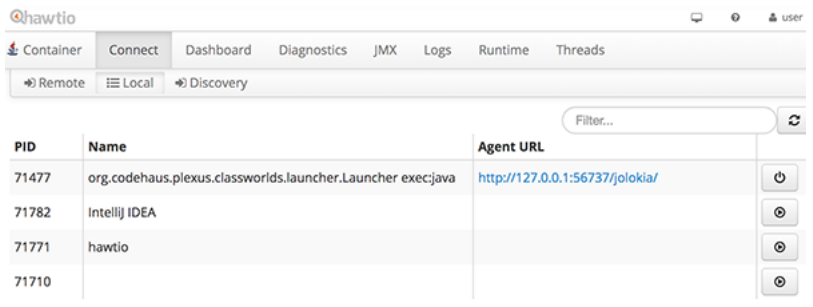
If you have any existing Java application running in a JVM, you can start hawtio in standalone mode and then from hawtio connect to the existing JVM. In the example located in the chapter19/hawtio-directory, instead of running the `hawto:run` plugin, you run `exec:java`:

```
mvn compile exec:java
```

Then from another terminal, you download hawtio-app and run it as a standalone Java application:

```
java -jar hawtio-app-1.5.6.jar
```

When hawtio opens in the web browser, the Connect plugin should be selected by default. In the center of the screen, you click the Local button, which lists all the local JVMs running on your computer. One of the rows in the table should have the text `exec:java`, which is the JVM that runs the Camel application. You can then click the Play button on the right-hand side to create a connection to this JVM, which then shows a hyperlink. After you've done this, you should have a screen similar to figure 19.12.



PID	Name	Agent URL
71477	org.codehaus.plexus.classworlds.launcher.Launcher exec:java	http://127.0.0.1:56737/jolokia/
71782	IntelliJ IDEA	
71771	hawtio	
71710		

Figure 19.12 Running hawtio in standalone mode, to connect to any local JVMs running on your computer. When you click the hyperlink in the Agent URL, hawtio opens a web page connected to that JVM. This allows you to manage and look inside that running JVM (for example, the Camel plugin if Camel is running).

If you're interested in building web tooling for Java applications, we encourage you to look into hawtio and the technologies it's using, such as AngularJS, Bootstrap, TypeScript, and Jolokia. You can find more hawtio plugin examples from the hawtio website (http://hawt.io) and from its source code hosted on GitHub (https://github.com/hawtio/hawtio).

19.4 Summary and best practices

In this chapter, we've shown you the best Camel tools you can find on the internet. The best-known tools are the Camel editors for graphical drag-and-drop development, such as the Eclipse tooling from JBoss. But a set of smaller and lighter tools is on the rise, such as the Apache Camel IDEA plugin, which focuses on extending the source code editor instead of on graphical drag-and-drop development. It's expected that some of these capabilities will find their way into Eclipse as well.

You learned that all the Camel components, data formats, EIPs, languages, and everything else are all cataloged in the Camel Catalog, which allows tools to know *everything*. The tools from JBoss and the Camel IDEA plugin all use this to provide type-safe editors for editing Camel endpoints and EIPs.

You also shouldn't miss out on the Camel Maven plugin that can check the source code during compile time to catch misconfigured Camel endpoints. Many Camel users have found great value with the hawtio web console, so make sure to try it out as well.

The takeaways from this chapter are as follows:

- *Graphical tooling*—New Camel developers and less experienced Java developers may find value in starting their Camel development by using the graphical editor from JBoss in Eclipse.
- *Camel route debugger*—A powerful feature from the JBoss Eclipse tooling is the Camel route debugger, which is integrated seamlessly into Eclipse. As an alternative debugger, you can use hawtio from a web browser.
- *Type-safe endpoint editor*—The Camel IDEA tooling provides a lightweight editor that allows you to edit/add endpoints directly in the source code from the cursor position. We think this kind of tooling will appeal to both new and experienced Camel developers.
- *Camel Maven plugin*—Don't miss out on the Maven plugin that you can run during compilation to check for misconfigured Camel routes.
- *Use the Camel Catalog*—The Camel Catalog is a goldmine of information that tooling developers should tap into. All the tools covered in this chapter are using the catalog.

With the goldmine of information from the Camel Catalog and the powers from Jolokia, Maven, IDEA, Eclipse, and hawtio, we think you have powerful ways to build custom tools you may need for your Camel-based integration applications.

These tools are all open source and have an open community. We welcome you to participate and help improve these tools with features you may find useful, or to have some fun with hacking on code that isn't your typical day-to-day job.

Goodbye from Claus and Jonathan

With that, we've come to the end of our not-so-little book on Apache Camel. In this second edition, we tried to cover as much of Camel's core features as possible, with the addition of how to use Camel in more modern settings, such as a framework for microservices, deployed in the cloud with Docker and Kubernetes, or even as a reactive engine, as we discuss in the bonus chapter (available online only). Believe it or not, there's still a lot more material. For that, though, we refer you to the Camel website, which has a full reference of all 200+ components. Besides, reading (or writing) a book about all 200 components wouldn't be the most exciting thing in the world.

One final note about Camel is that it is an active project and changes quickly. We suggest keeping up with Camel via its community (discussed in appendix B) and staying current on new features. Maybe you'll even find the inspiration to contribute to Apache Camel yourself.

We hope you've enjoyed the *Camel in Action*, 2nd Edition ride. We surely enjoyed making it yet again to the finish line after two and a half years in the making. Hope to see you around in the Camel community—as we say many times, we love contributions!

Cheers,

Claus and Jonathan

PS: If you like Apache Camel, we'd appreciate it if you'd give Camel a star on GitHub (https://github.com/apache/camel), and you're also quite welcome to star this book (https://github.com/camelinaction/camelinaction2) as well.