13

Parallel processing

This chapter covers

- Camel's threading model
- Configuring thread pools and thread profiles
- Using concurrency with EIPs
- Handling scalability with Camel
- Writing asynchronous Camel components

Concurrency is another word for *multitasking*, and we multitask all the time in our daily lives. We put the coffee on, and while it brews, we grab the tablet and glance at the morning news while we eagerly wait for the coffee to be ready. Computers are also capable of doing multiple tasks—you may have multiple tabs open in your web browser while your mail application is fetching new email, for example.

Juggling multiple tasks is also common in enterprise systems, such as when you're processing incoming orders, handling invoices, and doing inventory management, and these demands only grow over time. With concurrency, you can achieve higher performance; by executing in parallel, you can get more work done in less time.

Camel processes multiple messages concurrently in Camel routes, and it uses the concurrency features from Java, so we'll first discuss how concurrency works in Java before moving on to how thread pools work and how you define and use them in Camel. The *thread pool* is the mechanism in Java that orchestrates multiple tasks. After we've discussed thread pools, we'll move on to using concurrency with the EIPs.

The last section focuses on achieving high scalability with Camel and using this in your custom components. We take extra care to tell you all the ins and outs of writing custom asynchronous components, as this is a bit complicated to understand at first and do the right way. But it's the price to pay for having Camel achieve high scalability (routing using nonblocking mode).

You can also build highly scalable applications using reactive streams, which is covered separately in chapter 20 (available online).

13.1 Introducing concurrency

As we've mentioned, you can achieve higher performance with concurrency. When performance is limited by the availability of a resource, we say it's *bound* by that resource—CPU bound, I/O bound, database bound, and so on. Integration applications are often I/O bound, waiting for replies to come back from remote servers or for files to load from a disk. This usually means you can achieve higher performance using resources more effectively, such as by keeping CPUs busy doing useful work.

Camel is often used to integrate disparate systems, where data is exchanged over the network. There's often a mix of resources, which are a combination of CPU bound or I/O bound. It's likely you can achieve higher performance using concurrency.

To help explain the world of concurrency, we'll look at an example. Rider Auto Parts has an inventory of all the parts its suppliers currently have in stock. It's vital for any business to have the most accurate and up-to-date information in its central ERP system. Having the information locally in the ERP system means the business can operate without depending on online integration with their suppliers. This system has been in place for many years and is therefore based on what would be considered a legacy system today. The data is exchanged between stakeholders using files that are transferred via FTP servers.

Figure 13.1 illustrates this business process. Figure 13.2 shows the route of the inventory-updating Camel application from figure 13.1. Listing 13.1 details the application.

Figure 13.1 Suppliers send inventory updates, which are picked up by a Camel application. The application synchronizes the updates to the ERP system.

Figure 13.2 A route picks up incoming files, which are split and transformed to be ready for updating the inventory in the ERP system. This application is responsible for loading the files and splitting the file content on a line-by-line basis using the Splitter EIP, which converts the line from CSV format to an internal object model. The model is then sent to another route that's responsible for updating the ERP system.

Implementing this in Camel is straightforward, as shown in the following listing.

Listing 13.1 Rider Auto Parts application for updating inventory

```
public void configure() throws Exception {
    from("file:rider/inventory")
        .log("Starting to process file: ${header.CamelFileName}")
        .split(body().tokenize("\n")).streaming()    1
```

1

Splits file line by line

```
            .bean(InventoryService.class, "csvToObject")
            .to("direct:update")
        .end()
        .log("Done processing file: ${header.CamelFileName}");

  from("direct:update")    2
```

2

Updates ERP system

```
        .bean(InventoryService.class, "updateInventory");
}
```

Listing 13.1 shows the `configure` method of the Camel `RouteBuilder` that contains the two routes for implementing the application. As you can see, the first route picks up the files and then splits the file content line by line ❶. This is done using the Splitter EIP in *streaming mode*. Streaming mode ensures that the entire file isn't loaded into memory; instead it's loaded piece by piece on demand, which ensures low memory usage.

To convert each line from CSV to an object, you use a bean—the `InventoryService` class. To update the ERP system, you use the `updateInventory` method of the `InventoryService`, as shown in the second route ❷.

Now suppose you're testing the application by letting it process a big file with 100,000 lines. If each line takes a tenth of a second to process, processing the file would take 10,000 seconds, which is roughly 167 minutes. That's a long time. In fact, you might end up in a situation where you can't process all the files within the given timeframe.

In a moment, we'll look at various techniques for speeding things up using concurrency. But first let's set up the example to run without concurrency to create a baseline to compare to the concurrent solutions.

13.1.1 Running the example without concurrency

The book's source code contains this example (both with and without concurrency) in the chapter13/bigfile directory.

First, you need a big file to be used for testing. To create a file with 1,000 lines, use the following Maven goal:

```
mvn compile exec:java -PCreateBigFile -Dlines=1000
```

A bigfile.csv file will be created in the target/inventory directory.

The next step is to start a test that processes bigfile.csv without concurrency. This is done using the following Maven goal:

```
mvn test -Dtest=BigFileTest
```

When the test runs, it'll output its progress to the console.

`BigFileTest` simulates updating the inventory by sleeping for a tenth of a second, which means it should complete processing the bigfile.csv in approximately 100 seconds (with a 1,000-line file). When the test completes, it should log the total time taken:

```
[ad 0 - file://target/inventory] INFO - Inventory 997 updated
[ad 0 - file://target/inventory] INFO - Inventory 998 updated
[ad 0 - file://target/inventory] INFO - Inventory 999 updated
[ad 0 - file://target/inventory] INFO - Done processing big file
Took 102 seconds
```

In the following section, you'll see three solutions to run this test more quickly using concurrency.

13.1.2 Using concurrency

The application can use concurrency by updating the inventory in parallel. Figure 13.3 shows this principle using the Concurrent Consumers EIP.

Figure 13.3 Using the Concurrent Consumers EIP to provide concurrency and process inventory updates in parallel

As you can see, the incoming files ❶ are being split ❷ into individual lines that are parallelized ❸. By doing this, you can parallelize steps ❹ and ❺ in the route. In this example, those two steps could process messages concurrently.

The last step ❺, which sends messages to the ERP system concurrently, is possible only if the system allows a client to send messages concurrently to it. In some situations, a system won't permit concurrency or may allow up to only a certain number of concurrent messages. Check the service-level agreement (SLA) for the system you integrate with. Another reason to disallow concurrency is if the messages have to be processed in the exact order they're split. Let's try three ways to run the application faster with concurrency:

- Using `parallelProcessing` options on the Splitter EIP
- Using a custom thread pool on the Splitter EIP
- Using staged event-driven architecture (SEDA)

The first two solutions are features that the Splitter EIP provides out of the box. The last solution is based on the SEDA principle, which uses queues between tasks.

Using parallelProcessing

The Splitter EIP offers an option to switch on parallel processing, as shown here:

```
.split(body().tokenize("\n")).streaming().parallelProcessing()
    .bean(InventoryService.class, "csvToObject")
    .to("direct:update")
.end()
```

Configuring this in Spring XML is simple as well:

```
<split streaming="true" parallelProcessing="true">
    <tokenize token="\n"/>
    <bean beanType="camelinaction.InventoryService"
           method="csvToObject"/>
    <to uri="direct:update"/>
</split>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileParallelTest
mvn test -Dtest=SpringBigFileParallelTest
```

As you'll see, the test is now much faster and completes in about a tenth of the previous time:

```
[Camel Thread 1 - Split] INFO - Inventory 995 updated
[Camel Thread 4 - Split] INFO - Inventory 996 updated
[Camel Thread 9 - Split] INFO - Inventory 997 updated
[Camel Thread 3 - Split] INFO - Inventory 998 updated
[Camel Thread 2 - Split] INFO - Inventory 999 updated
[e://target/inventory?]  INFO - Done processing big file
Took 11 seconds
```

When `parallelProcessing` is enabled, the Splitter EIP uses a thread pool to process the messages concurrently. The thread pool is, by default, configured to use 10 threads, which helps explain why it's about 10 times faster: the application is mostly I/O bound (reading files and remotely communicating with the ERP system involves a lot of I/O activity). The test would not be 10 times faster if it were solely CPU bound—for example, if all it did was “crunch numbers.”

NOTE In the console output, you’ll see that the thread name is displayed, containing a unique thread number, such as `Camel Thread 4 - Split`. This thread number is a sequential, unique number assigned to each thread as it’s created, in any thread pool. This means if you use a second Splitter EIP, the second splitter will most likely have numbers assigned from 11 upward.

You may have noticed from the previous console output that the lines were processed in order; it ended by updating 995, 996, 997, 998, and 999. This is a coincidence, because the 10 concurrent threads are independent and run at their own pace. The reason they appear in order here is that we simulated the update by delaying the message for a tenth of a second, which means they’ll all take approximately the same amount of time. But if you take a closer look in the console output, you’ll probably see some interleaved lines, such as with order lines 954 and 953:

```
[Camel Thread 5 - Split] INFO - Inventory 951 updated
[Camel Thread 7 - Split] INFO - Inventory 952 updated
[Camel Thread 8 - Split] INFO - Inventory 954 updated
[Camel Thread 9 - Split] INFO - Inventory 953 updated
```

You now know that `parallelProcessing` will use a default thread pool to achieve concurrency. What if you want to have more control over which thread pool is being used?

Using a custom thread pool

The Splitter EIP also allows you to use a custom thread pool for concurrency. You can create a thread pool using the `java.util.Executors` factory:

```
ExecutorService threadPool = Executors.newCachedThreadPool();
```

The `newCachedThreadPool` method creates a thread pool suitable for executing many small tasks. The pool automatically grows and shrinks on demand.

To use this pool with the Splitter EIP, you need to configure it as shown here:

```
.split(body().tokenize("\n")).streaming().executorService(threadPool)
    .bean(InventoryService.class, "csvToObject")
    .to("direct:update")
.end()
```

Creating the thread pool using Spring XML is done as follows:

```
<bean id="myPool" class="java.util.concurrent.Executors"
       factory-method="newCachedThreadPool"/>
```

The Splitter EIP uses the pool by referring to it, using the `executorSer-viceRef` attribute:

```
<split streaming="true" executorServiceRef="myPool">
    <tokenize token="\n"/>
    <bean beanType="camelinaction.InventoryService"
           method="csvToObject"/>
    <to uri="direct:update"/>
</split>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileCachedThreadPoolTest
mvn test -Dtest=SpringBigFileCachedThreadPoolTest
```

The test is now much faster and completes within a few seconds:

```
[pool-1-thread-442] INFO - Inventory 971 updated
[pool-1-thread-443] INFO - Inventory 972 updated
[pool-1-thread-449] INFO - Inventory 982 updated
[/target/inventory] INFO - Done processing big file
Took 2 seconds
```

You may wonder why it’s now so fast. The cached thread pool is designed to be aggressive and to spawn new threads on demand. It has no upper bounds and no internal work queue, which means that when a new task is being handed over, it'll create a new thread if there are no available threads in the thread pool.

You may also have noticed the thread name in the console output, which indicates that many threads were created; the output shows thread numbers 442, 443, and 449. Many threads have been created because the Splitter EIP splits the file lines more quickly than the tasks update the inventory. The thread pool receives new tasks at a faster pace than it can execute them; new threads are created to keep up.

This can cause unpredicted side effects in an enterprise system; a high number of newly created threads may impact applications in other areas. That's why it's often desirable to use thread pools with an upper limit for the number of threads.

For example, instead of using the cached thread pool, you could use a fixed thread pool. You can use the same `Executors` factory to create such a pool:

```
ExecutorService threadPool = Executors.newFixedThreadPool(20);
```

Creating a fixed thread pool in Spring XML is done as follows:

```
<bean id="myPool" class="java.util.concurrent.Executors"
       factory-method="newFixedThreadPool">
    <constructor-arg index="0" value="20"/>
</bean>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileFixedThreadPoolTest
mvn test -Dtest=SpringBigFileFixedThreadPoolTest
```

The test is now limited to use 20 threads at most:

```
[pool-1-thread-13] INFO - Inventory 997 updated
[pool-1-thread-19] INFO - Inventory 998 updated
[ pool-1-thread-5] INFO - Inventory 999 updated
```

```
[target/inventory] INFO - Done processing big file
Took 6 seconds
```

As you can see by running this test, you can process the 1,000 lines in about 6 seconds and using only 20 threads. The previous test was faster, as it completed in about 2 seconds, but it used nearly 500 threads (this number can vary on different systems). By increasing the fixed thread pool to a reasonable size, you should be able to reach the same timeframe as with the cached thread pool. For example, running with 50 threads completes in about 3 seconds. You can experiment with different pool sizes.

Now, on to the last concurrency solution: SEDA.

Using SEDA

Staged event-driven architecture, or SEDA, is an architecture design that breaks down a complex application into a set of stages connected by queues. In Camel lingo, that means using internal memory queues to hand over messages between routes.

NOTE The Direct component in Camel is the counterpart to SEDA. Direct is fully synchronized and works like a direct method call invocation.

Figure 13.4 shows how you can use SEDA to implement the example. The first route runs sequentially in a single thread. The second route uses concurrent consumers to process the messages that arrive on the SEDA endpoint, using multiple concurrent threads.

Figure 13.4 Messages pass from the first to the second route using SEDA. Concurrency is used in the second route.

The following listing shows how to implement this solution in Camel using the `seda` endpoints, shown in bold.

Listing 13.2 Rider Auto Parts inventory-update application using SEDA

```
public void configure() throws Exception {
    from("file:rider/inventory")
        .log("Starting to process file: ${header.CamelFileName}")
        .split(body().tokenize("\n")).streaming()
            .bean(InventoryService.class, "csvToObject")
            .to("seda:update")
         .end()
         .log("Done processing file: ${header.CamelFileName}");

    from("seda:update?concurrentConsumers=20")    (1)
```

(1) SEDA consumers using concurrency

```
        .bean(InventoryService.class, "updateInventory");
}
```

By default, a `seda` consumer will use only one thread. To use concurrency, you use the `concurrentConsumers` option to increase the number of threads—to 20 in this listing (1).

The equivalent example from listing 13.2 is as follows when using XML DSL:

```
<route>
  <from uri="file:target/inventory?noop=true"/>
  <log message="Starting to process big file: ${header.CamelFileName}"/>
  <split streaming="true">
    <tokenize token="\n"/>
    <bean beanType="camelinaction.InventoryService" method="csvToObject"
    <to uri="seda:update"/>
  </split>
  <log message="Done processing big file: ${header.CamelFileName}"/>
</route>

<route>
  <from uri="seda:update?concurrentConsumers=20"/>    (1)
```

1

SEDA consumers using concurrency

```
   <bean beanType="camelinaction.InventoryService"
          method="updateInventory"/>
</route>
```

To run this example, use the following Maven goals:

```
mvn test -Dtest=BigFileSedaTest
mvn test -Dtest=SpringBigFileSedaTest
```

The test is fast and completes in about 6 seconds:

```
[ead 20 - seda://update] INFO - Inventory 997 updated
[ead 18 - seda://update] INFO - Inventory 998 updated
[read 9 - seda://update] INFO - Inventory 999 updated
Took 6 seconds
```

As you can see from the console output, you're now using 20 concurrent threads to process the inventory update. For example, the last three thread numbers from the output are 20, 18, and 9.

NOTE When using `concurrentConsumers` with SEDA endpoints, the thread pool uses a fixed size, which means that a fixed number of active threads are waiting at all times to process incoming messages. That's why it's best to use the concurrency features provided by the EIPs, such as the `parallelProcessing` on the Splitter EIP. It'll use a thread pool that can grow and shrink on demand, so it won't consume as many resources as a SEDA endpoint will.

We've now covered three solutions for applying concurrency to an existing application, and they all greatly improve performance. We were able to reduce the 102-second processing time down to 3 to 7 seconds, using a reasonable size for the thread pool.

TIP The camel-disruptor component works like the SEDA component but uses a different thread model with the LMAX Disruptor library instead of using thread pools from the JDK. You can find more information on the Camel website: http://camel.apache.org/disruptor.

In the next section, we'll review thread pools in more detail and learn about the threading model used in Camel. With this knowledge, you can go even further with concurrency.

13.2 Using thread pools

Using thread pools is common when using concurrency. In fact, thread pools were used in the example in the previous section. It was a thread pool that allowed the Splitter EIP to work in parallel and speed up the performance of the application.

In this section, we'll start from the top and briefly recap what a thread pool is and how it's represented in Java. Then we'll show you the default thread pool profile used by Camel and how to create custom thread pools using Java DSL and XML DSL.

13.2.1 Understanding thread pools in Java

A *thread pool* is a group of threads created to execute a number of tasks in a task queue. Figure 13.5 shows this principle.

Figure 13.5 Tasks from the task queue wait to be executed by a thread from the thread pool.

NOTE For more info on the Thread Pool pattern, see the Wikipedia article on the subject: http://en.wikipedia.org/wiki/Thread_pool.

Thread pools were introduced in Java 1.5 by the new concurrency API residing in the `java.util.concurrent` package. In the concurrency API, the `ExecutorService` interface is the client API that you use to submit tasks for execution. Clients of this API are both Camel end users and Camel itself, because Camel fully uses the concurrency API from Java.

NOTE Readers already familiar with Java's concurrency API may be in familiar waters as we go further in this chapter. If you want to learn in more depth about the Java concurrency API, we highly recommend *Java Concurrency in Practice* by Brian Goetz (Addison-Wesley Professional, 2006).

In Java, the `ThreadPoolExecutor` class is the implementation of the `ExecutorService` interface, and it provides a thread pool with the options listed in table 13.1.

Table 13.1 Options provided by thread pools from Java

Option	Type	Description
`corePoolSize`	`int`	Specifies the number of threads to keep in the pool, even if they're idle.
`maximumPoolSize`	`int`	Specifies the maximum number of threads to keep in the pool.
`keepAliveTime`	`long`	Sets the idle time for excess threads to wait before they're discarded.
`unit`	`TimeUnit`	Specifies the time unit used for the `keepAliveTime` option.
`allowCoreThreadTimeOut`	`boolean`	Determines whether the core threads may time out and terminate if no tasks arrive within

Option	Type	Description
		the keep alive time.
`rejected`	`RejectedExecution-Handler`	Identifies a handler to use when execution is blocked because the thread pool is exhausted.
`workQueue`	`BlockingQueue`	Identifies the task queue for holding waiting tasks before they're executed.
`threadFactory`	`ThreadFactory`	Specifies a factory to use when a new thread is created.

As you can see in table 13.1, you can use many options when creating thread pools in Java. To make it easier to create commonly used types of pools, Java provides `java.util.concurrent.Executors` as a factory, which you saw in section 13.1.2. In section 13.2.3, you'll see how Camel makes creating thread pools even easier.

When working with thread pools, you often must deal with additional tasks. For example, it's important to ensure that the thread pool is shut down when your application is being shut down; otherwise, it can lead to memory leaks. This is particularly important in server environments when running multiple applications in the same server container, such as a Java servlet, Java EE, or OSGi container.

When using Camel to create thread pools, the activities listed in table 13.2 are taken care of out of the box by Camel.

Table 13.2 Activities for managing thread pools taken care of by Camel

Activity	Description
Shutdown	Ensures the thread pool will be properly shut down, which happens when Camel shuts down.
Management	Registers the thread pool in JMX, which allows you to manage the thread pool at runtime. We'll look at management in chapter 16.
Unique thread names	Ensures the created threads will use unique and human-readable names.
Activity logging	Logs lifecycle activity of the pool.

Another good practice that's often neglected is to use human-understandable thread names, because those names are logged in production logs. By allowing Camel to use a common naming standard to name the threads, you can better understand what happens when looking at log files (particularly if your application is running together with other frameworks that create their own threads). For example, this log entry indicates it's a thread from the Camel file component:

```
[Camel (camel-1) thread #7 - file://riders/inbox] DEBUG - Total 3 files
```

If Camel didn't do this, the thread name would be generic and wouldn't give any hint that it's from Camel, nor that it's the file component:

```
[Thread 0] DEBUG - Total 3 files to consume
```

TIP Camel uses a customizable pattern for naming threads. The default pattern is `Camel Thread ##counter# - #name#`. A custom pattern can be configured using `ExecutorServiceManager`.

We'll cover the options listed in table 13.1 in more detail in the next section, when we review the default thread profile used by Camel.

13.2.2 Using Camel thread pool profiles

Camel thread pools aren't created and configured directly, but via the configuration of thread pool profiles. A *thread pool profile* dictates how a thread pool should be created, based on a selection of the options listed previously in table 13.1.

Thread pool profiles are organized in a simple two-layer hierarchy with custom and default profiles. There's always one default profile, and you can optionally have multiple custom profiles. The default profile is defined using the options listed in table 13.3.

Table 13.3 Settings for the default thread pool profile

Option	Default value	Description
`poolSize`	`10`	The thread pool will always contain at least 10 threads in the pool.
`maxPoolSize`	`20`	The thread pool can grow up to at most 20 threads.
`keepAliveTime`	`false`	Determines whether core thread is also allowed to terminate when there are no pending tasks.
`allowCoreThreadTimeOut`	`60`	Idle threads are kept alive for 60 seconds, after which they're terminated.
`maxQueueSize`	`1000`	The task queue can contain up to 1,000 tasks before the pool is exhausted.
`rejectedPolicy`	`CallerRuns`	If the pool is exhausted, the caller thread will execute the task.

As you can see from these default values, the default thread pool can use from 10 to 20 threads to execute tasks concurrently. The `rejectedPolicy` option corresponds to the `rejected` option from table 13.1, and it's an `enum` type allowing four values: `Abort`, `CallerRuns`, `DiscardOldest`, and `Discard`. The `CallerRuns` option uses the caller thread to execute the task itself. The other three options either abort by throwing an exception or discard an existing task from the task queue to make room for the new task.

If the `maxQueueSize` option is configured to `0`, there's no task queue in use. When a task is submitted to the thread pool, the task is processed only if there's an available thread in the pool. If there's no free thread, the rejection policy decides what happens (for example, to use the caller thread or fail with an exception). This is demonstrated later in section 13.3.1, where we process files concurrently without a task queue in use. No one-size-fits-all solution exists for every Camel application, so you may have to tweak the default profile values. But usually you're better off leaving the default values alone. Only by load testing your applications can you determine that tweaking the values will produce better results.

Configuring the default thread pool profile

You can configure the default thread pool profile from either Java or XML DSL.

In Java, you access `ThreadPoolProfile` starting from `CamelContext`. The following code shows how to change the maximum pool size to 50:

```
ExecutorServiceManager manager = context.getExecutorServiceManager();
ThreadPoolProfile profile = manager.getDefaultThreadPoolProfile();
profile.setMaxPoolSize(50);
```

The default `ThreadPoolProfile` is accessible from `ExecutorServiceManager`, which is an abstraction in Camel allowing you to plug in different thread pool providers. We cover `ExecutorServiceManager` in more detail in section 13.2.4.

In XML DSL, you configure the default thread pool profile using the `<threadPoolProfile>` tag:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <threadPoolProfile id="myDefaultProfile"
                       defaultProfile="true"
```

```
                                             maxPoolSize="50"/>
        ...
    </camelContext>
```

It's important to set the `defaultProfile` attribute to `true` to tell Camel that this is the default profile. You can add options if you want to override any of the other options from table 13.3.

In some situations, one profile isn't sufficient. You can also define custom profiles.

Configuring custom thread pool profiles

Defining custom thread pool profiles is much like configuring the default profile.

In Java DSL, a custom profile is created using the `ThreadPoolProfileBuilder` class:

```
ThreadPoolProfile custom = new ThreadPoolProfileBuilder("bigPool")
                                     .maxPoolSize(200).build();
context.getExecutorServiceManager().registerThreadPoolProfile(custom);
```

This example increases the maximum pool size to 200. All other options will be inherited from the default profile, which means it will use the default values listed in table 13.3; for example, `keepAliveTime` will be 60 seconds. Notice that this custom profile is given the name `bigPool`; you can refer to the profile in the Camel routes using `executorServiceRef`:

```
.split(body().tokenize("\n")).streaming().executorServiceRef("bigPool")
    .bean(InventoryService.class, "csvToObject")
    .to("direct:update")
.end()
```

And in XML DSL:

```
<split streaming="true" executorServiceRef="bigPool">
    <tokenize token="\n"/>
    <bean beanType="camelinaction.InventoryService" method="csvToObject"
    <to uri="direct:update"/>
</split>
```

When Camel creates this route with the Splitter EIP, it refers to a thread pool with the name `bigPool`. Camel will now look in the registry for an `ExecutorService` type registered with the ID `bigPool`. If none is found, it will fall back and see whether there's a known thread pool profile with the ID `bigPool`. And because such a profile has been registered, Camel will use the profile to create a new thread pool to be used by the Splitter EIP. All of this means that `executorServiceRef` supports using thread pool profiles to create the desired thread pools.

When using XML DSL, it's simpler to define custom thread pool profiles. All you have to do is use the `<threadPoolProfile>` tag:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <threadPoolProfile id="bigPool" maxPoolSize="200"/>
</camelContext>
```

Besides using thread pool profiles, you can create thread pools in other ways. For example, you may need to create custom thread pools if you're using a third-party library that requires you to provide a thread pool. Or you may need to create one, as we did in section 13.1.2, to use concurrency with the Splitter EIP.

13.2.3 Creating custom thread pools

Creating thread pools with the Java API is a bit cumbersome, so Camel provides a nice way of doing this in both Java DSL and XML DSL.

Creating custom thread pools in Java DSL

In Java DSL, you use `org.apache.camel.builder.ThreadPoolBuilder` to create thread pools, as follows:

```
ThreadPoolBuilder builder = new ThreadPoolBuilder(context);
ExecutorService myPool = builder.poolSize(5).maxPoolSize(25)
                            .maxQueueSize(200).build("Cool");
```

`ThreadPoolBuilder` requires `CamelContext` in its constructor, because it'll use the default thread pool profile as the baseline when building custom thread pools. That means `myPool` will use the default value for `keepAliveTime`, which would be 60 seconds.

Creating custom thread pools in XML DSL

In XML DSL, creating a thread pool is done using the `<threadPool>` tag:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <threadPool id="myPool" threadName="Cool"
                   poolSize="5" maxPoolSize="25" maxQueueSize="200"/>
    <route>
        <from uri="direct:start"/>
        <to uri="log:start"/>
        <threads executorServiceRef="myPool">    ❶
```

❶

Using thread pool in the route

```
            <to uri="log:hello"/>
        </threads>
    </route>
</camelContext>
```

As you can see, `<threadPool>` is used inside a `<camelContext>` tag. That's because it needs access to the default thread profile, which is used as baseline (just as the `ThreadPoolBuilder` requires `CamelContext` in its constructor).

The preceding route uses a `<threads>` tag that references the custom thread pool ❶. If a message is sent to the `direct:start` endpoint, it should be routed to `<threads>`, which will continue routing the message using the custom thread pool. This can be seen in the console output that logs the thread names. The thread name in the log will use the name you configured in the `<threadPool>` tag, which in the example is `Cool`, as shown in bold:

```
Camel (camel-1) thread #0 Cool] INFO hello - Exchange[Body:Hello Camel]
```

NOTE When using `executorServiceRef` to look up a thread pool, Camel will first check for a custom thread pool. If none is found, Camel will fall back and see if a thread pool profile exists with the given name; if so, a new thread pool is created based on that profile.

All thread pool creation is done using `ExecutorServiceManager`, which defines a pluggable API for using thread pool providers.

13.2.4 Using ExecutorServiceManager

The `org.apache.camel.spi.ExecutorServiceManager` interface defines a pluggable API for thread pool providers. Camel will, by default, use the `DefaultExecutorServiceManager` class, which creates thread pools using the concurrency API in Java. When you need to use a different thread pool provider (for example, a provider from a Java EE server), you can create a custom `ExecutorServiceManager` to work with the provider.

In this section, we'll show you how to configure Camel to use a custom `ExecutorServiceManager`, leaving the implementation of the provider up to you.

Configuring Camel to use a custom ExecutorServiceManager

In Java, you configure Camel to use a custom `ExecutorServiceManager` via the `setExecutorServiceManager` method on `CamelContext`:

```
CamelContext context = ...
context.setExecutorServiceManager(myExecutorServiceManager);
```

In XML DSL, it's easy because all you have to do is define a bean. Camel will automatically detect and use it:

```
<bean id="myExecutorService"
       class="camelinaction.MyExecutorServiceManager"/>
```

So far in this chapter, we've mostly used thread pools in Camel routes, but they're also used in other areas, such as in some Camel components.

Using ExecutorServiceManager in a custom component

The `ExecutorServiceManager` defines methods for working with thread pools.

Suppose you're developing a custom Camel component and you need to run a scheduled background task. When running a background task, it's recommended that you use the `ScheduledExecutorService` as the thread pool, because it's capable of executing tasks in a scheduled manner.

Creating the thread pool is easy with the help of Camel's `ExecutorServiceManager`, as shown in the following listing.

Listing 13.3 Using `ExecutorServiceManager` to create a thread pool

```
public class MyComponent extends DefaultComponent implements Runnable {
    private static final Log LOG = LogFactory.getLog(MyComponent.class);
    private ScheduledExecutorService executor;

    public void run() {
        LOG.info("I run now");    (1)
```

(1)
Runs scheduled task

```
    }

    protected void doStart() throws Exception {
        super.doStart();
        executor = getCamelContext().getExecutorServiceManager()
            .newScheduledThreadPool(this,
                "MyBackgroundTask", 1);    (2)
```

(2)
Creates scheduled thread pool

```
            executor.scheduleWithFixedDelay(this, 1, 1, TimeUnit.SECONDS);
        }

        protected void doStop() throws Exception {
            getCamelContext().getExecutorServiceManager().shutdown(executor)
            super.doStop();
        }
    }
```

Listing 13.3 illustrates the principle of using a scheduled thread pool to repeatedly execute a background task. The custom component extends `DefaultComponent`, which allows you to override the `doStart` and `doStop` methods to create and shut down the thread pool. In the `doStart` method, you create the `ScheduledExecutorService` using `ExecutorServiceManager` ❷ and schedule it to run the task ❶ once every second using the `scheduleWithFixedDelay` method.

The book's source code contains this example in the chapter13/pools directory. You can try it using the following Maven goal:

```
mvn test -Dtest=MyComponentTest
```

When it runs, you'll see the following output in the console:

```
Waiting for 10 seconds before we shutdown
[Camel (camel-1) thread #0 - MyBackgroundTask] INFO  MyComponent - I run
[Camel (camel-1) thread #0 - MyBackgroundTask] INFO  MyComponent - I run
```

You now know that thread pools are how Java achieves concurrency; they're used as executors to execute tasks concurrently. You also know how to use this to process messages concurrently in Camel routes, and you saw several ways of creating and defining thread pools in Camel.

When modeling routes in Camel, you'll often use EIPs to build the routes to support your business cases. In section 13.1, you used the Splitter EIP and learned to improve performance using concurrency. In the next section, we'll look at other EIPs you can use with concurrency.

13.3 Parallel processing with EIPs

Some of the EIPs in Camel support parallel processing out of the box—they're listed in table 13.4. This section takes a look at them and the benefits they offer.

Table 13.4 EIPs in Camel that support parallel processing

EIP	Description
Aggregator	The Aggregator EIP allows concurrency when sending out completed and aggregated messages. We covered this pattern in chapter 5.
Delayer	The Delayer EIP allows you to delay messages during routing. The delay can either be synchronous (the current thread is blocked) or asynchronous (the delay uses a scheduled thread pool to continue routing in a future time). Only in the latter situation can the Delayer EIP process messages in parallel due to the usage of the scheduled thread pool.
Multicast	The Multicast EIP allows concurrency when sending a copy of the same message to multiple recipients. We discussed this pattern in chapter 2, and we'll use it in an example in section 13.3.2.
Recipient List	The Recipient List EIP allows concurrency when sending copies of a single message to a dynamic list of recipients. This works in the same way as the Multicast EIP, so what you learned there also applies for this pattern. We covered this pattern in chapter 2.
Splitter	The Splitter EIP allows concurrency when each split message is being processed. You saw how to do this in section 13.1. This pattern was also covered in chapter 5.
Threads	The Threads EIP always uses concurrency to hand over messages to a thread pool that will continue processing the message. You saw an example of this in section 13.2.3, and we'll cover it a bit more in section 13.3.1.

EIP	Description
Throttler	The Throttler EIP allows you to throttle messages during routing, where some messages may be held back when the throttling limit has been reached. The throttling can either be *synchronous* (the current thread is blocked) or *asynchronous* (the throttling uses a scheduled thread pool to continue routing in a future time). Only in the latter situation can the Throttler EIP process messages in parallel due to the usage of the scheduled thread pool. This is similar to how the Delayer EIP works with parallel processing.
Wire Tap	The Wire Tap EIP allows you to spawn a new message and let it be sent to an endpoint using a new thread while the calling thread can continue to process the original message. The Wire Tap EIP always uses a thread pool to execute the spawned message. This is covered in section 13.3.3. You encountered the Wire Tap pattern in chapter 2.

All the EIPs from table 13.4 can be configured to enable concurrency in the same way. You can turn on `parallelProcessing` to use thread pool profiles to apply a matching thread pool; this is likely what you'll want to use in most cases. Or you can refer to a specific thread pool using the `executorService` option. You've already seen this in action in section 13.1.2, where we used the Splitter EIP.

In the following three sections, we'll see how to use the Threads, Multicast, and Wire Tap EIPs in a concurrent way.

13.3.1 Using concurrency with the Threads EIP

The Threads EIP is the only EIP that has additional options in the DSL offering a fine-grained definition of the thread pool to be used. These additional options were listed previously in table 13.3.

For example, the thread pool from section 13.2.3 could be written as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <to uri="log:start"/>
        <threads threadName="Cool" poolSize="5" maxPoolSize="25"
                   maxQueueSize="200">
            <to uri="log:cool"/>
        </threads>
    </route>
</camelContext>
```

Figure 13.6 illustrates which threads are in use when a message is being routed using the Threads EIP.

Figure 13.6 Caller and pooled threads are in use when a message is routed. Two threads are active when a message is being routed. The caller thread hands over the message to the thread pool. The thread pool then finds an available thread in its pool to continue routing the message.

You can run this example from the chapter13/pools directory using the following Maven goal:

```
mvn test -Dtest=SpringInlinedThreadPoolTest
```

You'll see the following in the console:

```
[main]                                 INFO start - Exchange[Body:Hello
Camel (camel-1) thread #0 - Cool] INFO hello - Exchange[Body:Hello Camel
```

The first set of brackets contains the thread name. You see, as expected, two threads in play: `main` is the caller thread, and `Cool` is from the thread pool.

Parallel processing files

You can use the Threads EIP to achieve concurrency when using Camel components that don't offer concurrency. A good example is the Camel file component, which uses a single thread to scan and pick up files. Using the Threads EIP, you can allow the picked-up files to be processed concurrently.

The following listing shows an example of how to do that with XML DSL.

Listing 13.4 Processing files concurrently with the Thread EIP

```
<bean id="delayProcessor" class="camelinaction.DelayProcessor"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">

  <route id="myRoute" autoStartup="false">
  <from uri="file:target/inbox?delete=true"/>   1
```

1 Consuming files from the starting directory

```
      <log message="About to process ${file:name} thread #${threadName}"/>
  <threads poolSize="10" maxQueueSize="0">   2
```

2 Using Threads EIP to process the files in parallel

```
        <log message="Start ${file:name} thread #${threadName}"/>
    <process ref="delayProcessor"/>   3
```

3 Random delay processing the files to simulate CPU processing

```
        <log message="Done  ${file:name} thread #${threadName}"/>
      </threads>
  <to uri="log:done?groupSize=10"/>   4
```

4 Log average processing speed per 10 files

```
  </route>

</camelContext>
```

This route starts from a file directory ❶, where all incoming files are picked up by the Camel file component. This is done using a single thread. To archive parallel processing of the files, you use threads ❷ configured with a thread pool of 10 active threads and no task queue. By setting `maxQueueSize` to `0`, you don't use any in-memory task queue in the thread pool. The file consumer will pick up only new files, when there's a thread available from Threads EIP to process the file. By default, the thread pool would otherwise have a `maxQueueSize` of `1000` as outlined previously in table 13.3. A processor ❸ is used to delay processing each file—otherwise, the files are all processed too fast. A log endpoint ❹ is used to log the average processing speed per 10 files.

The book's source code contains this example in chapter13/pools, and you can run the example using the following Maven goals:

```
mvn test -Dtest=FileThreadsTest
mvn test -Dtest=SpringFileThreadsTest
```

We encourage you to try this example and experiment with the various settings of the Threads EIP. In addition, pay attention to the logging output to see which thread is doing what.

Let's look at how Rider Auto Parts improves performance using concurrency with the Multicast EIP.

13.3.2 Using concurrency with the Multicast EIP

Rider Auto Parts has a web portal where its employees can look up information, such as the current status of customer orders. When selecting a particular order, the portal needs to retrieve information from three systems to gather an overview of the order. Figure 13.7 illustrates this.

Figure 13.7 The web portal gathers information from three systems to compile the overview that's presented to the employee.

Your boss has summoned you to help with this portal. The employees have started to complain about poor performance, and it doesn't take you more than an hour to find out why: the portal retrieves the data from the three sources in sequence. This is obviously a good use-case for using concurrency to improve performance.

You also look in the production logs and see that a single overview takes 4.0 seconds (1.4 + 1.1 + 1.5 seconds) to complete. You tell your boss that you can improve the performance by gathering the data in parallel.

Back at your desk, you build a portal prototype in Camel that resembles the current implementation. The prototype uses the Multicast EIP to retrieve data from the three external systems as follows:

```
<route>
    <from uri="direct:portal"/>
    <multicast strategyRef="aggregatedData">
        <to uri="direct:crm"/>
        <to uri="direct:erp"/>
        <to uri="direct:shipping"/>
    </multicast>
    <bean ref="combineData"/>
</route>
```

The Multicast EIP will send copies of a message to the three endpoints and aggregate their replies using the `aggregatedData` bean. When all data has been aggregated, the `combineData` bean is used to create the reply that will be displayed in the portal.

You decide to test this route by simulating the three systems using the same response times as from the production logs. Running your test yields the following performance metrics:

```
TIMER - [Message: 123] sent to: direct://crm took: 1404 ms.
TIMER - [Message: 123] sent to: direct://erp took: 1101 ms.
TIMER - [Message: 123] sent to: direct://shipping took: 1501 ms.
TIMER - [Message: 123] sent to: direct://portal took: 4139 ms.
```

As you can see, the total time is 4.1 seconds when running in sequence. Now you enable concurrency with the `parallelProcessing` options:

```
<route>
    <from uri="direct:portal"/>
    <multicast strategyRef="aggregatedData"
               parallelProcessing="true">
        <to uri="direct:crm"/>
        <to uri="direct:erp"/>
        <to uri="direct:shipping"/>
    </multicast>
    <bean ref="combineData"/>
</route>
```

This gives much better performance:

```
TIMER - [Message: 123] sent to: direct://erp took: 1105 ms.
TIMER - [Message: 123] sent to: direct://crm took: 1402 ms.
TIMER - [Message: 123] sent to: direct://shipping took: 1502 ms.
TIMER - [Message: 123] sent to: direct://portal took: 1623 ms.
```

The numbers show that response time went from 4.1 to 1.6 seconds, which is an improvement of roughly 250 percent. Note that the logged lines aren't in the same order as the sequential example. With concurrency enabled, the lines are logged in the order that the remote services' replies come in. Without concurrency, the sequential order is always fixed as defined by the Camel route.

The book's source code contains this example in the chapter13/eip directory. You can try the two scenarios using the following Maven goals:

```
mvn test -Dtest=MulticastTest
mvn test -Dtest=MulticastParallelTest
```

You've now seen how the Multicast EIP can be used concurrently to improve performance. The Aggregator, Recipient List, and Splitter EIPs can be configured with concurrency in the same way as the Multicast EIP.

The next pattern we'll look at using with concurrency is the Wire Tap EIP.

13.3.3 Using concurrency with the Wire Tap EIP

The Wire Tap EIP uses a thread pool to process the tapped messages concurrently. You can configure which thread pool it should use, and if no pool has been configured, it'll fall back and create a thread pool based on the default thread pool profile.

Suppose you want to use a custom thread pool when using the Wire Tap EIP. First, you must create the thread pool to be used, and then you pass that in as a reference to the wire tap in the route:

```
public void configure() throws Exception {
    ExecutorService lowPool = new ThreadPoolBuilder(context)
        .poolSize(1).maxPoolSize(5).build("LowPool");

    from("direct:start")
        .log("Incoming message ${body}")
        .wireTap("direct:tap", lowPool)
        .to("mock:result");

    from("direct:tap")
        .log("Tapped message ${body}")
        .to("mock:tap");
}
```

The equivalent route in XML DSL is as follows. Notice how `<wireTap>` uses the attribute named `executorServiceRef` to refer to the custom thread pool to be used:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">

    <threadPool id="lowPool"
                poolSize="1" maxPoolSize="5" threadName="LowPool"/>

    <route>
        <from uri="direct:start"/>
        <log message="Incoming message ${body}"/>
        <wireTap uri="direct:tap" executorServiceRef="lowPool"/>
        <to uri="mock:result"/>
    </route>
    <route>
        <from uri="direct:tap"/>
        <log message="Tapped message ${body}"/>
        <to uri="mock:tap"/>
    </route>
</camelContext>
```

The book's source code contains this example in the chapter13/eip directory. You can run the example using the following Maven goals:

```
mvn test -Dtest=WireTapTest
mvn test -Dtest=SpringWireTapTest
```

When you run the example, the console output should indicate that the tapped message is being processed by a thread from the `LowPool` thread pool:

```
[main]                                INFO route1 - Incoming message Hel
[Camel (camel-1) thread #0 - LowPool] INFO route2 - Tapped message Hello
```

WIRE TAP AND STREAM-BASED MESSAGES

The Wire Tap EIP creates a shallow copy of the message that's being tapped and routes the second message concurrently. If the message body is streaming based (for example, `java.io.InputStream` type), you should consider enabling stream caching that allows concurrent access to the stream. Chapter 15 provides more details about stream caching.

Wire Tap is useable not only for tapping messages during routing. The pattern can also be used when you want to return an early reply to the caller, while Camel is concurrently processing the message.

Returning an early reply to the caller

Consider an example in which a caller invokes a Camel service in a synchronous manner—the caller is blocked while waiting for a reply. In the Camel service, you want to send a reply back to the waiting caller as soon as possible; the reply is an acknowledgment that the input has been received, so `OK` is returned to the caller. In the meantime, Camel continues processing the received message in another thread.

Figure 13.8 illustrates this example in a sequence diagram.

Figure 13.8 A synchronous caller invokes a Camel service. The service lets the wire tap continue processing the message asynchronously while the service returns an early reply to the waiting caller.

The following listing implements this example as a Camel route with the Java DSL.

Listing 13.5 Using Wire Tap to return an early reply message to the caller using Java DSL

```
from("jetty:http://localhost:8080/early").routeId("input")
    .wireTap("direct:incoming") 1
```

1

Taps incoming message

```
    .transform().constant("OK"); 2
```

2

Returns early reply

```
from("direct:incoming").routeId("process") 3
```

3

Route that processes the message asynchronously

```
        .convertBodyTo(String.class)
        .log("Incoming ${body}")
        .delay(3000)
        .log("Processing done for ${body}")
        .to("mock:result");
}
```

You use the Wire Tap EIP ❶ to continue routing the incoming message in a separate thread, in the process route ❸. This gives room for the consumer to immediately reply ❷ to the waiting caller.

The next listing shows an equivalent example using XML DSL.

Listing 13.6 Using WireTap to return an early reply message to the caller using XML DSL

```
<camelContext xmlns="http://camel.apache.org/schema/spring">

    <route routeId="input">
        <from uri="jetty:http://localhost:8080/early"/>
    <wireTap uri="direct:incoming"/>    ❶
```

❶ Taps incoming message

```
        <transform>
            <constant>OK</constant>    ❷
```

❷ Returns early reply

```
        </transform>
    </route>

  <route routeId="process">    ❸
```

❸ Route that processes the message asynchronously

```
            <from uri="direct:incoming"/>
            <convertBodyTo type="String"/>
            <log message="Incoming ${body}"/>
            <delay>
                <constant>3000</constant>
            </delay>
            <log message="Processing done for ${body}"/>
            <to uri="mock:result"/>
        </route>

</camelContext>
```

The book's source code contains this example in the chapter13/eip directory. You can run the example using the following Maven goals:

```
mvn test -Dtest=EarlyReplyTest
mvn test -Dtest=SpringEarlyReplyTest
```

When you run the example, you should see the console output showing how the message is processed:

```
11:18:15 [main] INFO - Caller calling Camel with message: Hello Camel
11:18:15 [Camel (camel-1) thread #0 - WireTap] INFO - Incoming Hello Cam
11:18:15 [main] INFO - Caller finished calling Camel and received reply:
11:18:18 [Camel (camel-1) thread #0 - WireTap] INFO - Processing done fo
```

Notice in the console output that the caller immediately receives a reply within the same second the caller sent the request. The last log line shows that the Wire Tap EIP finished processing the message 3 seconds after the caller received the reply.

NOTE In the preceding example, (the route with ID `process`) you need to convert the body to a `String` type to ensure that you can read the message multiple times. This is necessary because Jetty is stream based; consequently, Camel reads the message only once. Or, instead of converting the body, you could enable stream caching (covered in chapter 15).

So far in this chapter, you've seen concurrency used in Camel routes by the various EIPs that support them. The remainder of the chapter focuses on how scalability works with Camel and how you can build custom components that would support scaling.

13.4 Using the asynchronous routing engine

Camel uses its routing engine to route messages either synchronously or asynchronously. This section focuses on scalability. You'll learn that higher scalability can be achieved with the help of the asynchronous routing engine.

For a system, *scalability* is the desirable property of being capable of handling a growing amount of work gracefully. In section 13.1, we covered the Rider Auto Parts inventory application, and you saw that you could increase throughput using concurrent processing. In that sense, the application was scalable, because it could handle a growing amount of work in a graceful manner. That application could scale because it had a mix of CPU-bound and I/O-bound processes, and because it could use thread pools to distribute work.

In the next section, we'll look at scalability from a different angle. We'll see what happens when messages are processed asynchronously.

13.4.1 Hitting the scalability limit

Rider Auto Parts uses a Camel application to service its web store, as illustrated in figure 13.9. As you can see, Jetty consumer handles all requests from the customers. There are a variety of requests to handle, such as updating shopping carts, performing searches, gathering production information, and so on—the usual functions you expect from a web store. But one function involves calculating pricing information for customers. The pricing model is complex and individual to each customer; only the ERP system can calculate the pricing. As a result, the Camel application communicates with the ERP system to gather the prices. While the prices are calculated by the ERP system, the web store has to wait until the reply comes back before it returns its response to the customer.

Figure 13.9 The Rider Auto Parts web store communicates with the ERP system to gather pricing information.

The business is doing well for the company, and an increasing number of customers are using the web store, which puts more load on the system. Lately, problems have been occurring during peak hours, with customers reporting that they can't access the web store or that it's generally responding slowly.

The root cause has been identified: the communication with the ERP system is fully synchronous, and the ERP system takes an average of 5 seconds to compute the pricing. This puts a burden on the Jetty thread pool, as there are fewer free threads to service new requests.

Figure 13.10 illustrates this problem. You can see that the thread is blocked (the white boxes) while waiting for the ERP system to return a reply.

Figure 13.10 A scalability problem illustrated by the thread being blocked (represented as white boxes) while waiting for the ERP system to return the reply

Figure 13.10 reveals that the Jetty consumer is using one thread per request. This leads to a situation where you run out of threads as traffic increases. You've hit a scalability limit. Let's look into why and check out what Camel has under the hood to help mitigate such problems.

13.4.2 Scalability in Camel

It would be much better if the Jetty consumer could somehow *borrow* the thread while it waits for the ERP system to return the reply, and use the thread in the meantime to service new requests. This can be done using an asynchronous processing model. Figure 13.11 shows the principle.

Figure 13.11 The scalability problem is greatly improved. Threads are much less blocked (represented by white boxes) when you use asynchronous communication between the systems.

If you compare figures 13.10 and 13.11, you can see that the threads are much less blocked in the latter (the white boxes are smaller). No threads are blocked while the ERP system is processing the request. This is a huge scalability improvement because the system is much less affected by the processing speed of the ERP system. If it takes 1, 2, 5, or 30 seconds to reply, it doesn't affect the web store's resource use as much as it would otherwise. The threads in the web store are much less I/O bound and are put to better use doing actual work.

Figure 13.12 shows a situation in which two customer requests are served by the same thread without impacting response times. In this situation, customer 1 sends a request that requires a price calculation, so the ERP system is invoked asynchronously. A short while after, customer 2 sends a request that can be serviced directly by the web shop service, so it doesn't use the asynchronous processing model (it's synchronous). The response is sent directly back to customer 2. Later, the ERP system returns the reply, which is sent back to the waiting customer 1.

Figure 13.12 The same thread services multiple customers without blocking (white and gray boxes) and without impacting response times, resulting in much higher scalability.

In this example, you can successfully process two customers without increasing their response times. You've achieved higher scalability.

The next section peeks under the hood to see how this is possible in Camel when using the asynchronous processing model.

13.4.3 Components supporting asynchronous processing

The routing engine in Camel is capable of routing messages either synchronously or asynchronously. The latter requires the Camel component to support asynchronous processing, which in turn depends on the underlying transport supporting asynchronous communication. To achieve high scalability in the Rider Auto Parts web store, you need to use asynchronous routing at two points. The communication with the ERP system and with the Jetty consumer must both happen asynchronously. The Jetty component already supports this.

Communication with the ERP system must happen asynchronously too. To understand how this is possible with Camel, we'll take a closer look at figure 13.12. The figure reveals that after the request has been submitted to the ERP system, the thread won't block but will return to the Jetty consumer. It's then up to the ERP transport to notify Camel when the reply is ready. When Camel is notified, it'll be able to continue routing and let the Jetty consumer return the HTTP response to the waiting customer.

To enable all this to work together, Camel provides an asynchronous API that the components must use. The next section walks through this API.

13.4.4 Asynchronous API

Camel supports an asynchronous processing model, which we refer to as the *asynchronous routing engine*. Using asynchronous processing has advantages and disadvantages compared to using the standard synchronous processing model. They're listed in table 13.5.

Table 13.5 Advantages and disadvantages of using the asynchronous processing model

Advantage	Disadvantage
Processing messages asynchronously doesn't use up threads, forcing them to wait for processors to complete on blocking calls. It increases the scalability of the system by reducing the number of threads needed to manage the same workload.	Implementing asynchronous processing is much more complex.

The asynchronous processing model is manifested by an API that must be implemented to use asynchronous processing. You've already seen a glimpse of this API in figure 13.12; the arrow between the Jetty consumer and the web store service has the labels *Return false* and *Done*. Let's see the connection that those labels have with the asynchronous API.

AsyncProcessor

The `AsyncProcessor` is an extension of the synchronous `Processor` API:

```
public interface AsyncProcessor extends Processor {
   boolean process(Exchange exchange, AsyncCallback callback);
}
```

`AsyncProcessor` defines a single `process` method that's similar to its synchronous `Processor.process` sibling.

Here are the rules that apply when using `AsyncProcessor`:

- A non-null `AsyncCallback` must be supplied; it'll be notified when the exchange processing is completed.
- The `process` method must not throw any exceptions that occur while processing the exchange. Any such exceptions must be stored on the exchange via the `setException` method.
- The `process` method must know whether it'll complete the processing synchronously or asynchronously. The method will return `true` if it completes synchronously; otherwise, it returns `false`.
- When the processor has completed processing the exchange, it must call the `callback.done(boolean doneSync)` method. The `doneSync` parameter must match the value returned by the `process` method.

The preceding rules may seem a bit confusing at first. Don't worry—the asynchronous API isn't targeted at Camel end users but at Camel component writers. The next section covers an example of implementing a custom component that acts asynchronously. You'll be able to use this example as a reference if you need to implement a custom component.

NOTE You can read more about the asynchronous processing model at the Camel website: http://camel.apache.org/asynchronous-processing.html.

The `AsyncCallback` API is a simple interface with one method:

```
public interface AsyncCallback {
    void done(boolean doneSync);
}
```

It's this callback that's invoked when the ERP system returns the reply. This notifies the asynchronous routing engine in Camel that the exchange is ready to be continued, and the engine can then continue routing it.

Let's see how this all fits together by digging into the example and looking at some source code.

13.4.5 Writing a custom asynchronous component

The book's source code contains the web store example in the chapter13/scalability directory. This example contains a custom ERP component that simulates asynchronous communication with an ERP system. The following listing shows how the `ErpProducer` is implemented.

Listing 13.7 `ErpProducer` using the asynchronous processing model

```
public class ErpProducer extends DefaultAsyncProducer { 1
```

1 Extends DefaultAsyncProducer

```
    private ExecutorService executor;

    public ErpProducer(Endpoint endpoint) {
        super(endpoint);
    }

    protected void doStart() throws Exception {
        super.doStart();
        this.executor = getEndpoint().getCamelContext() 2
```

2 Thread pool used to simulate asynchronous tasks

```
            .getExecutorServiceManager().newFixedThreadPool(this, "ERP", 1
    }

    protected void doStop() throws Exception {
        super.doStop();
        getEndpoint().getCamelContext()
            .getExecutorServiceManager().shutdown(executor);
    }

    public boolean process(final Exchange exchange, 3
```

3

Implements asynchronous process method

```
                                  final AsyncCallback callback) {
        executor.submit(new ERPTask(exchange, callback));
        log.info("Returning false");
        return false;    4
```

4

Returns false to use asynchronous processing

```
    }

    private class ERPTask implements Runnable {
        private final Exchange exchange;
        private final AsyncCallback callback;

        private ERPTask(Exchange exchange, AsyncCallback callback) {
            this.exchange = exchange;
            this.callback = callback;
        }

        public void run() {
            log.info("Calling ERP");
            try {
                Thread.sleep(5000);
                log.info("ERP reply received");
                String in = exchange.getIn().getBody(String.class);
                exchange.getOut().setBody(in + ";516");    5
```

5

Sets reply on exchange

```
            } catch (Throwable e) {
                exchange.setException(e);    6
```

6

Catches all exceptions on the exchange

```
                } finally {
                    log.info("Continue routing");
                callback.done(false);    7
```

7

Notifies callback reply is ready

```
                }
            }
        }
    }
```

When implementing a custom asynchronous component, it's most often the `Producer` that uses asynchronous communication, and a good starting point is to extend the `DefaultAsyncProducer` ❶.

To simulate asynchronous communication, you use a thread pool to execute tasks asynchronously ❷; you need to create a thread pool in the `doStart` method of the producer. To support the asynchronous processing model, `ErpProducer` must also implement the asynchronous `process` method ❸.

To simulate the communication, which takes 5 seconds to reply, you submit `ERPTask` to the thread pool. When the 5 seconds are up, the reply is ready, and it's set on the exchange ❺. Notice how we use a `try ... catch ... finally` block to ensure that any errors are caught and set on the exchange ❻, and the `finally` block ensures that the callback is called ❼. This is a recommended design to use when building custom asynchronous Camel components.

According to the rules, when you're using `AsyncProcessor`, the `callback` must be notified when you're done with a matching synchronous parameter ❼. In this example, `false` is used as the synchronous parameter because the `process` method returned `false` ❹. By returning `false`, you instruct the Camel routing engine to use asynchronous routing from this point forward for the given exchange.

You can try this example by running the following Maven goal from the chapter13/scalability directory:

```
mvn test -Dtest=ScalabilityTest
```

This runs two test methods: one request is processed fully synchronously (not using the ERP component), and the other is processed asynchronously (by invoking the ERP component).

When running the test, pay attention to the console output. The synchronous test will log input and output as follows:

```
2017-07-17 11:41:42 [     qtp1444378545-11]             INFO  input
 - Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2017-07-17 11:41:42 [     qtp1444378545-11]             INFO  output
 - Exchange[ExchangePattern:InOut, Body:Some other action here]
```

Notice that both the input and output are being processed by the same thread.

The asynchronous example is different, as the console output reveals:

```
2017-07-17 11:49:48 [     qtp515060127-11]              INFO  input
 - Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper]
2017-07-17 11:49:48 [     qtp515060127-11]              INFO  ErpProduc
 - Returning false (processing will continue asynchronously)
2017-07-17 11:49:48 [Camel (camel-1) thread #0 - ERP] INFO  ErpProducer
 - Calling ERP
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO  ErpProducer
 - ERP reply received
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO  ErpProducer
- Continue routing
2017-07-17 11:49:53 [Camel (camel-1) thread #0 - ERP] INFO  output
- Exchange[ExchangePattern:InOut, Body:1234;4;1719;bumper;516]
```

This time two threads are used during the routing. The first is the thread from Jetty, which received the HTTP request. As you can see, this thread was used to route the message to `ErpProducer`. The other thread takes over communication with the ERP system. When the reply is received from the ERP system, the callback is notified, which lets Camel hijack the thread and use it to continue routing the exchange. You can see this from the last line, which shows the exchange routed to the log component.

Now, this was the happy path. When writing or using an asynchronous component, there's a price to pay, which is what we'll discuss next.

13.4.6 Potential issues when using an asynchronous component

You may have seen from implementing a custom asynchronous component (as shown in listing 13.7) that implementing this correctly is much more complicated. In particular, you need to understand the importance of making sure to call the `done` method on the `AsyncCallback` parameter. Calling the `done` method is what triggers the asynchronous routing engine in Camel to continue routing the exchange. Several potential issues can cause the `done` method to never be called, such as these:

- A bug in the code causes the `done` method to never be called.
- The remote system never returns with a reply, causing the task to not finish.
- The `done` method is called but with the wrong value (should use `false`).

This concludes our coverage of scalability and the ups and downs of implementing custom asynchronous components with Camel.

13.5 Summary and best practices

In this chapter, we looked at thread pools, the foundation for concurrency in Java and Camel. You saw how concurrency greatly improves performance, and we considered all the possible ways to create, define, and use thread pools in Camel. You saw how easy it is to use concurrency with the numerous EIPs in Camel. You witnessed how Camel can scale up using asynchronous (nonblocking) routing. This comes with a cost of complexity, which we covered in depth so you could learn about the pitfalls and best practices for implementing your custom Camel components.

Here are some best practices related to concurrency and scalability:

- *Use concurrency if possible*—Concurrency can greatly speed up your applications. Note that using concurrency requires business logic that can be invoked in a concurrent manner.
- *Tweak thread pools judiciously*—Tweak thread pools only when you have a means of measuring the changes. It's often better to rely on the default settings.
- *Use asynchronous processing for high scalability*—If you require high scalability, try using the Camel components that support the asynchronous processing model.
- *Take care when implementing your own asynchronous component*—You're required to structure your component code according to numerous rules. This ensures that Camel can route the messages when data has been received or a time-out has occurred, and avoids any potential issues with threads getting stuck.
- *Reactive systems*—Reactive streams and frameworks are gaining in popularity. You can find details on using these with Camel in chapter 20 (available online).

Now, prepare for something completely different, as you're about to embark on a journey focusing on securing your applications with Camel.