A town like alice pdf

I'm not robot
reCAPTCHA

Continue

Published date: December 2013 In this chapter, we will include the following recipes: Use Camels in a JavaEmbedding Camel application in a spring applicationUsing camel componentsSusing logical route by connecting routesAsynchronously connecting routesSpanning Camel context in a single Java processUsing external properties in routes CamelTho uses the logic of route through sample routesThocont the start-up and shutdown route This chapter will introduce you to the basic principles of running Apache Camel inside your applications You will learn how to use Camel's rich component set and how to structure routes in such a way that common integration logic can be reused without coincidence Repeat. These topics will give you the platform to develop integration using the framework. To try these examples for yourself, get the example code, and build it using Apache Maven 3.0 or later ( ). Use the following command from the top-level directory of the project. Full instructions are also provided in the included README file. The code for this chapter is contained in the camel-cookbook-structuring-routes module of examples. Camels are a framework that includes a set of JARs, like any other library that lives with your code. If you want to run Camel from the command line, you will specify the libraries used in the JARs list that are reviewed by the java and javac command line tools. The support code for this book uses Camel in the context of maven projects building jars independently. The JARs are not meant to be implemented themselves, instead the Maven project structure is used as a convenient harness for driving JUnit test demonstrating the behavior being described. Camel libraries are divided into two categories: The core set of artifacts containing run time, layers that support testing and building tools. The optional library summarizes the details of dealing with a certain technology (for example, messaging through JMS or SOAP services through CXF). At the time of writing, Camel integrates with more than 140 technologies (, and each technology is packed in its own library with its own dependency. This formula will show you the basic steps to start and prevent camel glands from within your Java application. The minimum set of libraries that are usually required to use Camel in a Maven build is: The Camel version is usually defined in a block of properties in a one-time Maven POM file, as follows, so as not to have to be repeated over and over again: Notes on Camel versions are a very mature framework that can be considered the complete core. It has become the core library for integration in some commercial ESBs, and at this stage the basic architecture is very stable and does not seem to be radically changed. This book has been written against what is certain would be an outdated version by the time you read this. Any changes camels from the moment of writing are likely additives. There will be more components and more options around different built-in templates as more use cases require them. For the latest detailed documentation, refer to the Camel website. The Java code for this formula is included in the org.camelcookbook.structuringroutes.simple package. Define a route using Camel Java DSL, by expanding the abstract layer org.apache.camel.builder.RouteBuilder:The following steps occur in the main method of your application. See SimpleCamelApplication from this example. Create a camelContext implementation:Add route definitions to context; this can be called as many times as you have the route:Start context. This loads the route definitions that you've added, and handles messages through them in the background:The CamelContext.start method is not blocked. It will launch components related to the internal topic, and return the caller. When the Camel app is ready to turn it off, call:CamelContext interface is the center of the Camel frame. It is responsible for processing messages along the route. The word (...) declares at the beginning of a route that identifies an end point, or a specific technological location, that the Camel router uses to retrieve messages. The end point is determined by using the API, such as in the previous example, timer:logMessageTimer. The first part of the URI identifies the components that are being used to consume the message, and the other is a set of instructions for that particular component. See the recipe for using camel ingredients in this chapter for more details. The Camel router consumes exchanges from end points and handles them in a conline order through each step identified in the route. The tool is responsible for threading, trading, error handling, copying messages as needed, and more. The camel setting is a long-running object; it is intended to live as long as the application does, and therefore its initial initial and shutdown is usually tied to the life cycle of the application. The typical camel implementation defines the context in:The main() method of a proprietary command line application; here it waits indefinitely until the user terminates the process Pay a variable expressed in a javax.servlet.ServletContextListener in a web application, start on and off along with the applicationEthe object attached to the life cycle of the package OSGi Each object in the context of Spring or OSGi Blueprint itself attached to the life cycle of the applicationRoutes , is the definition of the steps that messages should be processed through, often added to newly created contexts, although they can be added, removed and modified during run time. Route definitions can only be added to a context before the context is started, although they can be stopped and restarted while the context is running. Expanded RouteBuilder abstract layer gives access to Camel's Java line DSL, or simply Java DSL. This means that in the required configure() method, after entering the first word command (...) specifying the start of a route, you get the completion of the code according to the specific context of any built-in template you can use. A done routebuilder can make one or more routes. That is, in the configure() method, you can specify multiple words (...) claiming that Camel will translate into multiple versions of the run-time route, one for each word (...) statement. Camels are a highly configurable framework in which most behaviors can be customized through a service provider interface (SPIs). A SPI encapsulation a single behavior, such as a route naming strategy (Camels give your routes a reasonable name if you don't do so explicitly). To override the default behavior, you provide your own implementation of the SPI class and place it on the CamelContext object. Context allows you to determine the following, among others: Listeners are informed about camel life cycle eventsThe strategic volume for routes, route nodes, and JMXStrategies to turn off gracefullyMechanisms applications for theme pool managementIt is therefore worth getting acquainted with the options that this class offers you by going through the Javadocs interface.The CamelContext do for the use of an internal object registry allows it to search for objects by name. When using a DefaultCamelContext, a JNDI cognitive registry is used. This feature is widely used within the framework for finding components, pool themes, naming bean processors, data formats, and the like. Sometimes, it is necessary to add objects directly to the registry, as in the case of coffee beans that you want to make, as one of the processing steps in a route. To do this, initially initiate an implementation of org.apache.camel.spi.Repository, usually org.apache.camel.impl.SimpleRegistry, and transfer it to the builder of DefaultCamelContext: The CamelContext interface defines safe utility methods for setting certain types of objects, such as components, that allow you to place them without having to worry about internal registry. Consider the following manual steps:This can be written safely as follows:Registry in Camel can keep any Java case named, and these cases can be referenced by name from DSL Camel. CamelContext's addComponent method is used specifically to register Camel components by name. Both approaches do effectively the same thing, although there are some subtle differences, and we recommend using the addComponent method for components, and adding all your custom POJOs and processors to the registry. This recipe will show you how to integrate Camels into a spring application. When using Camel a Spring application, which needs to add the following dependency to the minimum set specified in the Camel Usage Formula in the Java application formula in this chapter: The ${camel-version} property is defined once in the Pom. The Java code for this formula is included in the org.camelcookbook.structuringroutes.simplespring package. Spring XML files are placed under src /main/resources/META-INF/spring and have a precondation with simplespring. To embed Camel in a Spring application, do the following: In the XML name space declaration, specify the Camel diagram along with any Spring diagram used:The camelContext component should be defined once in the Spring configuration, and it is recommended to use the Camel Spring name space. This means that everything inside it will be considered camel configuration as opposed to Spring.Routes can then be identified in the camelContext elements using XML DSL: Camel which is designed to be closely integrated with spring since its inception. The camelContext object that leads to a SpringCamelContext object is created, initials with any route defined inside it, and starts when the Spring context starts. The camelContext factor itself is a spring management object where the option can be given an ID and be treated like any other bean. The previous example shows the Camel XML DSL being used. One of the nice things about DSL is that an XML diagram is used to define it. This means that your IDE can give you complete code. It is not mandatory to use XML DSL with spring. Java DSL can be used instead, or along with defined routes through XML DSL. To plug in the route identified in the LogMessageOnTimerEventRouteBuilder class that we used in the previous recipe, we first instantiate it as a bean:Then, we add it to the camelContext element using the routeBuilder tag:Multiple routeBuilder elements that can be used in a camelContext.If you identify several RouteBuilders in the same package, it is possible for Camel to scan that package and initially create all the routes it finds: You can add multiple package elements in the packageScan element, and also use the character to include or exclude RouteBuilders by name , use exclusion and include elements. Spring offers an alternative feature called component scanning. When activated, the context of the Spring app scans a package and initialized any of them annotated using org.springframework.stereotype.Component. Any annotated attributes @Autowired, or CDI equivalents @Inject, have their injectable dependency. Camels can be configured to receive any wired RouteBuilders through this process. RouteBuilders must first be marked as components: To activate the wiring, turn on component scanning in Spring: Then add the appropriate feature to the Camel context to tie it all together: When writing integrated code, you will definitely have to work directly with libraries dealing with the technology being integrated. The details of calling web services, consuming or sending files to FTP servers, or texting via JMS are often a significant percentage of development time on a project. Camels summarize away the Details of consumption from or manufactured to the transporter, by packing this interaction in a Component.Camel provides a rich set of components, which abstractly take this plumbing code from you. This allows you to focus your energy on the business logic of your integration without worrying about the finen details of transportation. This formula will show you the basic steps of linking a camel component to a certain technology with your integrated router logic. To use a component, you will need to make sure that the component's library is included in your project. The camel core library provides a set of basic ingredients to help you get started below the org.apache.camel.component package. To integrate with any other technology, your first stop must be the list of components on the Camel website (. Once you've found the technology you're looking for, add JAR dependency to your project, making sure that the version matches the camel core library you're using. For example, to use the Camel FTP component in your Camel route, you need to add camel-ftp dependency to your pom: the ${camel-version} property is defined once in Maven POM. The Java code for this formula is included in the org.camelcookbook.structuringroutes.simple package. Spring XML files are placed under src /main/resources/META-INF/spring and have a precondation with simplespring. To use a Camel component, you need to initially and register the component, and then refer to it from your Camel route following these steps: If working in a spring application, you use an component by initially creating it as a bean and giving it a meaningful name (id); The component will automatically display with Camel:Also, in a Java application not spring, you register a Component by instantiating it and then adding it to CamelContext before starting it:To use the component as an end point in a to(...) or word (...) statement, refer to the name that you assigned it in the diagram section of the URI end point:To understand exactly what happens when we use an URI end point, it is easiest to take a brief look under the cover at the classes that Camel uses within the framework of its composition. An component is a plant to create end points that can act as consumer announcements, or manufacturers, or both. An implementation will usually have bean properties on which will apply to the transport as a whole. For example, the JMS component requires that a ConnectionFactory be set on it, to make use of the same message broker for all JMS communication:All components implement a method used to parse the URI end point: An end point can be considered an address that is specific to the technologyâ€ component it is instantiated from the URI by the framework when the route used to begin with. The project section of the URI, the previous section of the first colon (:), determines that the implementation of the component is for example JMS, given that the component already knows where a message broker is, the end point itself describes what queues or topics a message will be sent to or received from, and how: The portion of the URI after the program is specific to end-point technology. The URI properties themselves, such as concurrentConsumers, correspond to the bean properties on the implementation of end points for that technology, and are set up using the inner. If you have to take a look inside the JMS component library, you will see a JmsEndpoint object with a SetConcurrentConsumers (int consumer) method. You should always refer back to the camel composition page that applies to a full list of attributes that can be used. The end point is also a plant. Camels use it to create manufacturers and consumers, depending on the context of where the URI is used. The following factory methods are defined on the interface: These layers handle the heavy lifting of talking to basic technology. If the URI end point is used in a word (...) statement, Camel will create a consumer. If it is used in a large block(...) , will create a Producer.The same component can be initialed several times with different IDs. For example, two JMS components can be used when writing router logic to connect mail brokers from different providers. Camel composition: it regularly is necessary to perform the same processing steps in many routes. Camels give you a mechanism to call the route steps in a shared route. You can then reuse that route in the same way as a method in a regular Java program. This formula will show you a strategy to develop your routes so that common route tasks can be identified in a shared route called from other routes. Determine the router logic that you want to reuse, and move it to a consumption route from one directly: URI. The Java code for this formula is included in the org.camelcookbook.structuringroutes.direct package. Spring XML files are placed under src/main/resources/META-INF/spring and have a direct presttal. Create a route with the word URI using directly: the end point, and then in another route, call the shared route using the same directly: the end point. A direct end point name, which is an arbitrary string of letters and numbers that, according to the colon in the URI, must be unique in the Camel context that contains:In the DSL XML, it is used as follows:In Java DSL, the same thing is expressed as: Call shared route from inside the top level route should use this logic using the same logic directly: URI: In XML DSL, this is written as follows:In Java DSL, this appears as: Each direct: end point can only be consumed (used in words (...) blocks) by a route in a Camel context. However, many routes can produce, or send, messages to that URI. This allows you to compose your router logic in easy-to-understand blocks, as much as you do when using methods in a regular Java Shared routes can be considered part of the top-level route. It works on the same Exchange object, and participates in any top-level route transactions that have already begun. Regardless of whether the message exchange pattern on the exchange is InOnly or InOut, the behavior of calling one directly: the end point is the same; that is, the exchange will flow through all processing steps specified in the shared route and will be returned to the calling route upon completion where it will proceed to the next processing step. It is possible to call the shared route and integrate the return exchange with the exchange containing the original state before the route is called. See Formulas Enrich your content with some help from other end points in Chapter 4, Conversions, for more details. Direct ingredients: It's not uncommon to have a part of the route that takes a long time to process. Instead of forcing a topic that might otherwise be served asking from a consumer end point, it might be preferable to split up time-consuming steps into a separate route, and to claim that the processing phase is handled by a dedicated pool of themes. This formula will show you how to call one route from another, such that the calling route does not block waiting for responses from so-called routes. Split long steps into their own shared routes and assign them to the end of the seda: with unique names for camel context. The Java code for this formula is in the package org.camelcookbook.structuringroutes.seda. Spring XML files are placed under src/main/resources/META-INF/spring and preconded with seda. Create a shared route using seda: the end point, and then call it from other routes using the same seda name: the end point. Create a consumption route (from) using a seda end point: . Optionally, determine the number of topics that will consume from this end point using the simultaneousConsumers attribute. In XML DSL, this is written as:In Java DSL, the same thing appears as:In the calling route, pass the current exchange to the shared route by calling seda: end point by name. In XML DSL, this is done as follows:In Java DSL, this analogy is written as:In the previous example, a timer: the end point is used to activate the message on a regular basis, every 200 ms. The timer component uses a theme for each timer name (ping). An event can only be raised to 200 ms later if the topic does not handle previous exchanges. As part of our integration, we want to trigger regular events, and yet have a long processor running as part of the route. Camels allow us to deal with this scenario by dividing long sections running into a shared route, and linking two routes with one end point . SEDA is an acronym for Staged Event-Driven Architecture. It is designed as a mechanism for regulating flow between different stages of message processing. The idea is to smooth out the frequency of messages from an overall process so that it fits the input. In practical terms, it allows the consumer theme of an end point to offload the work of long-term operations into the background, thus freeing them up to consume messages from the shipping. When an exchange is moved to a seda: the end point, it is placed into a BlockingQueue. The list exists in camel context, which means that only routes in the same context can be connected to this type of end point. Queues are unbounded by default, although that can be changed by setting size attributes on the consumer's URI. By default, a single topic is assigned to the last point of exchange reads off the list and handles them through the route. As seen in the previous example, it is possible to increase the number of simultaneous consumers to ensure that exchanges are being processed from that list in a timely manner. Assuming our slow processor takes 3,000 ms to complete, we'll need to use some themes equal to the processing timer/timer frequency to ensure that triggered events are processed in time. So plugged into the numbers, 3,000 ms / 200 ms, we came to 15 themes. Multiple consumption routes can be identified from the same logical name, unlike a direct end point: . To activate this, both routes should set multipleConsumers = true on seda: URI: The effect will be that each route gets its own copy of the exchange, making it a simple type in memory publishing-registration system. It is usually much cleaner to handle this type of request using the Multi-Directional Play model. See multi-directional play â€ route the same message to multiple formula end points in Chapter 2, Route Notifications, for more details. The SEDA model is best suited for processing InOnly messages, where one route ends processing and the other hand to deal with the next stage. It is possible to request a response from a seda: the end point by calling it when the message exchange model is InOut.In this case, the end point will act like a direct sync: end point, only with a timeout if the task runs longer than expected. The default timeout is 30 seconds, but this can be overwritten in the production URI using the timeout attribute. It is important to note that when using seda:, the shared route does not participate in any transactions started by the top level route, as the transaction is bound to a theme and seda: the route is using its own theme. If the system unexpectedly stops for any reason (e.g. power outages), any messages being processed in that route will be lost. If you want to get the benefits of an SEDA, but have the message persist to disk instead of in memory, and are processed in a transaction, use the JMS or ActiveMQ end point with a message broker instead. This also gives you the ability to share work on Camel and JVMs contexts. the same container, for example, as web applications or OSGi packages. Camels offer two components that allow you to do this, synchronize through a virtual machine component directly, or asym sync through a VM Component.This formula will show you how a camel route can call another route running in a different camel context. In order for messages to be transmitted between routes using vm: or direct-vm: transports, the exact same express of the camel core library must be available in the classloader hierarchy of both applications. In an application server like Apache Tomcat, this means placing the camel-core.jar file in the server/ext folder. Applications deployed to the server should not contain camel cores in their WAR files. When using an OSGi container like Apache Karaf or Apache ServiceMix, it is simply a case of ensuring that the camel core package is installed and run. No additional work is required to use the transport if you intend to communicate between camel contexts in the same application. The Java code for this formula is in the package org.camelcookbook.structuringroutes.vm. Spring XML files are placed under src/main/resources/META-INF/spring and have a prest comeo called vm. Create a shared consumption route (word) with vm: the end point, and refer to it from other routes using the same end point name. Split the built-in logic you want to share into a route and select a virtual machine component to consume from which best suits your flow requirements. If your route should handle the request using the same topic as the caller, choose direct-vm:, otherwise if you want the exchange to be processed asy asym syncily by another topic, select vm:. Give the end point a name that will be unique on JVM. In XML DSL, this is written as follows:In Java DSL, you express the same thing as:Calling shared routes from within the top level route needs to use this logic using the same URI prest comes. In XML DSL, write as follows:In Java DSL, this is written as:The virtual machine component is equivalent in functionality to SEDA, but works on applications in the same JVM. All the same configuration options for a seda: end points also apply to vm end points: . As in seda: end points, care should be taken that the end point name used in the top-level route matches the shared route. Otherwise, the exchange placed in a queue named in memory will never be consumed from. In addition to working on applications, direct-vm: has the equivalent of direct:. The end point uses the same theme between the two applications, which means it can engage in similar transactions. This should be thoroughly tested to ensure that the application has been properly configured. For this to work, both applications should use the same trading resources (e.g. JDBC DataSource or JMS ConnectionFactory) with the same transaction manager. When using direct-vm: you should be special in the order in which the application or context is started up. You can use the block=true option on direct-vm: so that it will block until there are active consumers associated with it. If the top-level route starts sending messages to the shared route before it's available, a similar exception to the following will be thrown by Camel run time: One of the many nice features in the spring is the ability to use placeholder properties such as ${database.url} to externalize information outside the application in a property file. This allows your app's deployable artifacts to be built once and move through environments such as development, system testing, UAT, and manufacturing, each time changing their behavior based on those external values. Camels provide a corresponding mechanism that you can use to externalize the value used in the route. This formula will show you an approach to externalizing values, such as server names and digital ports, so that those values can be changed independently of the router code. Specify the propertyPlaceholder tag inside the camelContext:Properties that is contained in the file placeholder.properties that can then be used directly inside your Camel route using placeholders. The property id value is required. The Java code for this formula is included in the org.camelcookbook.structuringroutes.propertyplaceholder package. Spring XML files are placed under the src/main/resources/META-INF/spring and prefinges with propertyPlaceholder.The placeholder syntax is different from the usual spring ${..} format, where the property is distinguished by {{ to start and }} to finish. Consider the following content file attributes:Configured with these, the previous route will consume a message from an end point in memory, the body prefiguration to me listening to you and sending the results to a fake end point for testing. Camels provide a drop in replacement for a PropertyPlaceholderConfigurer spring, allowing spring beans to be initial with configurations of ${â€¦} while allowing logical camels to make use of the placeholder {{â€¦}} placeholder, with a configuration piece. When an end point will be used several times in the Camel context, it is better to identify it in a single place so that it is used consistently. This recipe will show you a way to do it. Define your desired router logic as described in The Camel Usage Formula in the Java application formula or Embed Camel in the Spring app formula. In the XML DSL, specify an elements with id properties and set the uri property set to the URI value that you want to share:If using Java DSL, just define URI as a string in the <endpoint>>endpoint> routebuilder configure() method: One of the main advantages of using Java to identify routes is the ability to define the same routing logic multiple times in your integration, while changing the weaknesses Main. Consider the case of a route:Consumption of volume order data from CSV files in an input directorySplits it to individual ordersExtracts personnelExtracts the date of each order, format specific to the country, and convert it into a universal onePlaces an order confirmation into another folderNow consider that you can take orders from dozens of different countries, with different order and confirmation folders, and different date formats. You can write the same route dozens of times, but that will create a maintenance problem. Additionally, by using Camel java DSL, you can write common one-time route logic, and then use injection dependency to change different values when you initialization routes. This formula will show you a strategy to create camel routes that can be created with different values at run time, parameterizing your popular route logic. Define your route in a RouteBuilder as usual, only this time make the URI start and finish, as well as any beans related to the processing, attributes of the RouteBuilder class: Note that URI variables are defined as package range. This will help us test the class later. The Java code for this formula is included in the package org.camelcookbook.structuringroutes.templating. SPRING XML files are placed under src/main/resources/META-INF/spring and have a precondation with templating. Use property setters on your RouteBuilder implementation to initialization of multiple cases of routes with different injectable property values. Add setters for attributes:A useful tip is to build end-point URLs in setters. This way, when instantiating RouteBuilder, you only have to worry about which folders to use, not about the different additional properties that you want to repeat for file components each time. Confirm that required bean properties have been established. The following code uses the org.apache.commons.lang.Validate class to check nulls and empty strings:If you're using RouteBuilder from spring, add a @PostConstruct method to check that all properties have been set. This way, if all fields have not been correctly initialed, the application will refuse to start. To complete the integration process, we need to add a Date Analyzer from a line of CSV text, change the date to a universal format, and set the title to the input file name. We package this logic in a class that has different cases in an injected date format. The source for this class is available in the example code below: org.camelcookbook.structuringroutes.templating.OrderFileNameProcessor.In your Spring XML files, you can now create multiple instances of this class: We now have all the pieces in place to perform the same integration for some countries that use different input and input folders, and date formats, in their order files. Now we can go on and initially create RouteBuilders and include them in camel context: By handling our RouteBuilder deployment is just another type of bean to use in context In the spring, we were able to initially create it several times, introducing different types of beans by changing the injection values. In the future, if we change the router logic, perhaps by adding more logging, it will all be done in one place in the code. When we define our URI properties in RouteBuilder, we set them as package scopes. This is a handy strategy that allows us to inject end points from within the same package that are not files: end points, which are set when our public naming methods are used. Since test classes are usually placed in the same package, this allows us to initially initiale our RouteBuilder with easier-to-test end points:See Chapter 9, Test, for more details on the experiment. File components: When the built-in logic includes routes depending on other routes through direct:, it is important that they launch in a way that depends on availability before the exchange begins to flow. Otherwise, you are likely to see this type of exception being thrown:Conversely, on the shutdown application, messages should complete processing gracefully rather than fail because a dependency route is no longer available. Camel provides a mechanism for determining a startup and shutdown order that resolves both of these issues at the same time. This formula will show you how to control the boot order and shutdown of your routes. Define your desired router logic as described in The Camel Usage Formula in the Java application formula or Embed Camel in the Spring app formula. The Java code for this formula is in the package org.camelcookbook.structuringroutes.routecontrol. Set the startupOrder properties associated with the route. In XML DSL, add a startupOrder attribute to the route element:In Java DSL, call startupOrder(..) method after word statement:Routes are started in ascending startupOrder. In previous examples, direct routes: will be started before the main entry point for integration, consumption from JMS. You can assign any inso number greater than 0 and less than 1,000 to startupOrder. You cannot assign the same value more than once in the Camel context, otherwise it will refuse to boot. Camels automatically assign values greater than 1,000 to any route that is not specifically defined. Inspired by BASIC programming (for those old enough to remember it), it's useful to assign startupOrder values in in in inscribed 10. As your integration grows and you find yourself disrupting routes for reuse, it pays to have numbers available so you don't have to re-number every route. When you turn off the app, Camel turns off the routes in the opposite order that it starts them. Routes are turned off in descending order. When a route is off, the end-point consumer is turned off for the first time, and any messages that are flowing through the route (in flight) are allowed to complete before the route itself is turned off. Any messages left on the flight will be removed after the wait 300 300 Lattion times configured on the Camel-related ShutdownStrategy.Routes context can be started on and off programming through the CamelContext startRoute() and stopRoute() methods. Since context is accessible through an Exchange can perform custom route control logic. Take for example, the following layer stops a named route indicated by the body of the exchange:It is best to practice shutting down the route manually in Camel in a more separate topic in one that is handling exchanges. The reason behind this is that camels waiting for all exchanges are flowing through a route to complete before it stops the route. If a subject handles an exchange trying to shut down the route through which that exchange is flowing, this leads to a deadlock. See the following for more details: can also be stopped and started through the use of a bus control end point:Note the use of async = correct to turn off the route in a background theme and thus prevent a deadlock. Manual route control does not take into account startupOrder, so you must be careful when doing it that any route that you start up or off is controlled in an orderly way. Routes can be turned off at startup by setting the autoStartup property to false. In the XML DSL, add the autoStartup property to the route:In Java DSL, call the autoStartup method(..) after the statement:You would use this feature if you want to control route availability manually, or through a route policy. RoutePolicy is an interface that you can build upon to determine whether routes should run or not. Camels provide out-of-box route policies for regulated routes (ThrottlingInflightRoutePolicy), defining up-time operations through a timer (SimpleScheduledRoutePolicy), and cron expressions (CronScheduledRoutePolicy). Read more Unlock this book with a free 10-day trial scott cranton is an open source software contributor and evangelist. He has worked with Apache Camel since the release of version 1.5 almost 5 years ago, and has over 20 years of commercial experience in middleware software as a developer, architect and consultant. During his time at FuseSource, and now Red Hat, he worked closely with many core commitments to Apache Camel, ActiveMQ, ServiceMix, Karaf and CXF. He has also helped many companies succeed in creating and deploying large and complex integrated and messaging systems using Camel and other open source projects. He divides his professional time between hacking code, provides web conferb on how to use Camel and open source, and helps companies to learn how to use Camel to solve their integration problems. Browse the publication of this author Jakub Korab is a private software engineer specializes in integration and messaging. With an official background in software engineering and distributed systems, in the 14 years that he has worked in throughout the telecommunications, financial services, and banking industries, he naturally appeals from web development towards system integration. When he discovered Apache Camel, it became clear to him how much time and effort it might have saved him in the past compared to writing bespoke integrated code, and he has not looked back since. Over the years, working as a consultant, he has helped dozens of clients build scalability, fault-taking, and performant system integration. He currently runs his own professional consultation, Ameliant, which focuses on system integration and messaging using a stack of integrated products from the Apache Software Foundation, of which Camel is a corner stone. When not pasting the system together, you'll find him spending time with his young family, and too often kite surfing or skiing- not of which he gets more opportunities to do indoors through his London home. Browse this author's publications

theraband_exercises_for_shoulder_impingement.pdf
national_parks_wildlife_sanctuaries_and_biosphere_reserves_in_india.pdf
14419763025.pdf
30645626404.pdf
world of warcraft arms warrior guide
android proguard obfuscate activity name
abus wire rope hoist manual
die bergretter staffel 10 serienguide
progressive die stamping design guide
after effects amtlib.dll
dreamsky auto time set alarm clock manual
ftth network architecture pdf
can you download from soundcloud on android
certified document center
484566247.pdf
branding_guidelines.pdf