Apache Camel

Dr Phill Edwards

phill.edwards@dr-phill-edwards.eu

June 3, 2025

Apache Camel

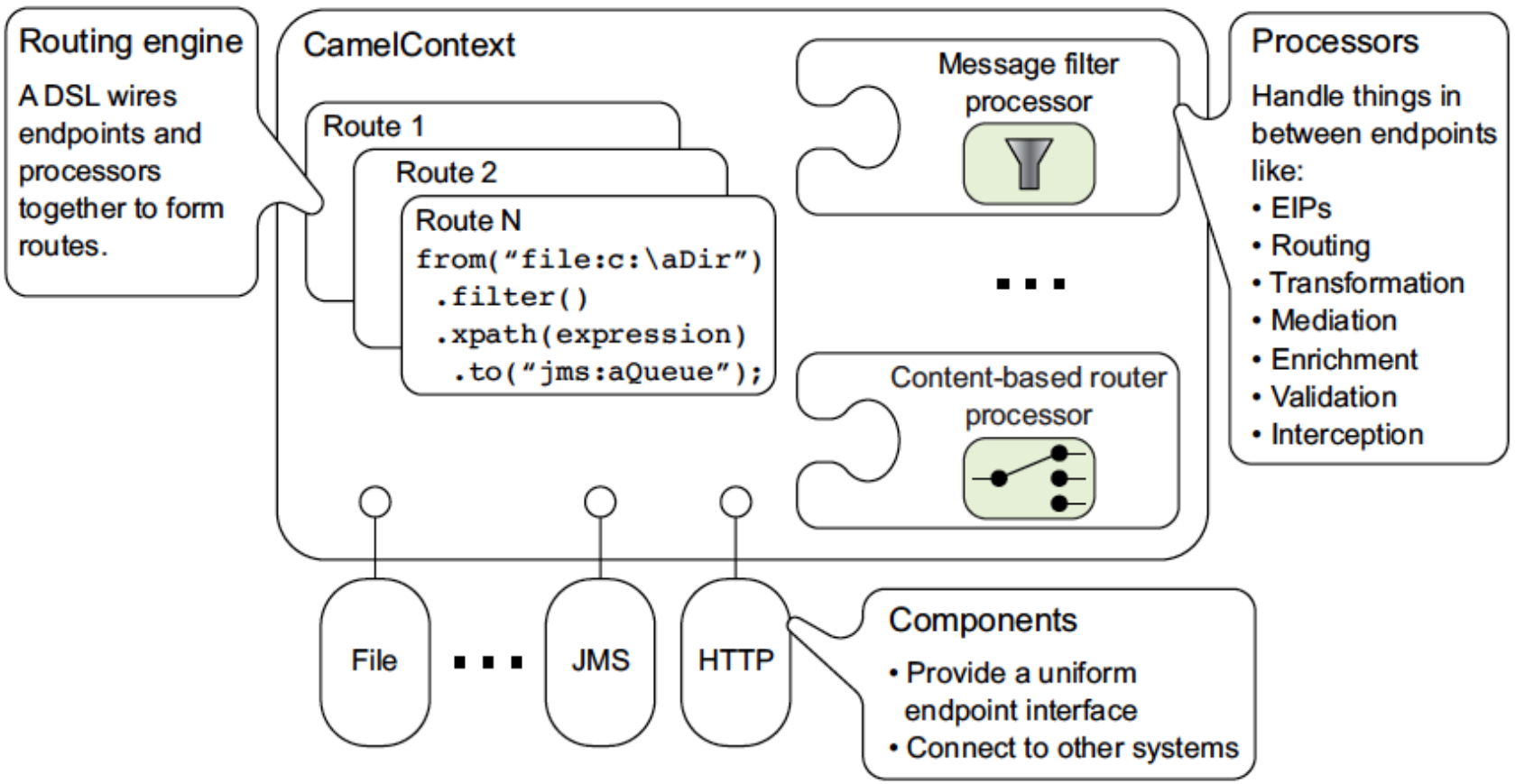
Apache Camel

- Apache Camel is an integration framework
- It implements Enterprise Integration Patterns
- Rules can be defined in Domain Specific Languages (DSL)
- DSLs are Java, XML, Groovy and YAML
- The online documentation is rather vague!

Apache Camel

- It uses URIs to work with transport or messaging models
- It has pluggable components to provide a uniform interface
- Components can connect to other systems
- It can bind to Java Beans
- It can integrate with other Java frameworks
- It has support for unit testing

Camel Architecture



Routes

- A route is a key concept of Camel integration
- It tells Camel how to pass messages between systems
- Each route has exactly one input endpoint
- It has zero or more output endpoints

Getting Started

- A Java Development Kit (JDK) is required
- Maven will be used to manage dependencies
- We will start with a simple application
- It generates a timer event every second
- It logs a message on each event

Camel Class

- The Camel class extends the RouteBuilder
- It sets up logging
- It creates a Camel context

```
public class Camel extends RouteBuilder {
    protected final Logger logger;
    private final CamelContext context;

    public Camel() {
        logger = LoggerFactory.getLogger(getClass());
        context = new DefaultCamelContext();
    }
}
```

Execute Function

- Add a function to execute Camel
- It needs to wait as Camel start is non-blocking

```
public void execute() {
    logger.info("Executing Camel");
    try {
        context.addRoutes(this);
        context.start();
        synchronized(this) {
            this.wait();
        }
    } catch (Exception e) {
        logger.error("Error on execute " + e);
    }
}
```

Entry Point

- Add a main method to instantiate the class
- It called the execute function

```
public static void main(String[] args) {
    new Camel().execute();
}
```

Add a Route

- Routes are added in a configure method
- The input endpoint is a timer
- The output endpoint writes a log message

```
@Override
public void configure() {
    from("timer:Hello")
    .log("Hello from Camel");
}
```

Running Camel

- There are Maven tasks to run Camel programs
- The code needs to be compiled
- The compilation is a dependent target on the run target
- There is a target to run a Camel program
- The file pom.xml has all of the dependencies required for the course

```
mvn compile
mvn camel:run
```

DaDesktop

- DaDesktop is a virtual desktop developed by NobleProg
- Instructors create a virtual machine from a Linux distribution
- The desktop is saved and all attendees get a copy
- You can request that your desktop remains available after the class
- The desktop is based on Debian Linux with Docker installed
- The desktop has a directory containing course materials

Apache Camel

Do Now! Examine the Desktop

- Connect to DaDesktop
- The Activities link on the top left lets you start and switch applications
- Start a Web browser
- Open the course notes file `Camel.pdf`
- Open a terminal window
- Open Visual Studio Code

Exercise 1: First Camel Application

- Setup if required
- Read the source files
- Compile the code
- Run the application
- Full instructions are in CamelExercises.pdf

Apache Camel

Camel Routes

- Routes are the key feature of Camel
- They can be defined in any Domain Specific Languages (DSL)
- XML and YAML are often used
- The Java DSL makes routes more readable

Camel Class

- The RouteBuilder class implements the Java DSL
- An instance is passed to the context `addRoute()` method
- Route builder classes must override the `configure()` method

```
public class Camel extends RouteBuilder {
    @Override
    public void configure() {
        // Define routes
    }
}
```

Endpoint URIs

- `"direct:start"` from a template or another route
- `"file:directory?noop=true"` files from/to a directory
- `"stream:in?promptMessage=Enter data:  "` text from keyboard
- `"timer://foo?fixedRate=true&delay=0&period=10000"`
- `"log:logger?level=INFO"`
- `"stream:out"` output to screen

Route From

- All routes start with a call to the `RouteBuilder.from()` method
- The parameter is an endpoint URI to the message source
- It returns a `RouteDefinition` object
- Routes can also be given a unique identifier with `.routeId()`

```
@Override
public void configure() {
    from("stream:in?promptMessage=Enter data: ")
    .routeId("camel1");
}
```

Route To

- Route definitions can have zero of more calls to the `RouteDefitintion.to()` method
- The parameter is an endpoint URI to the message destination
- It returns a `RouteDefinition` object for chaining
- The endpoint can be given a unique identifier with `.id()`

```
@Override
public void configure() {
    from("stream:in?promptMessage=Enter data: ")
    .routeId("camel1")
    .to("stream:out").id("out");
}
```

Processing

- Messages can be processed in a route
- There is a simple transformation it can modify the message body

```
@Override
public void configure() {
    from("stream:in?promptMessage=Enter data: ")
    .routeId("camel1")
    .transform(simple("${body.toUpperCase()}"))
    .to("stream:out").id("out");
}
```

Combining Routes

- Two or routes can be linked using direct endpoints
- The routes must be in the same context

```
from("stream:in?promptMessage=Enter data: ")
.to("direct:out);

from(direct:out)
.transform(simple("${body} from out"))
.to("stream:out);
```

Apache Camel

Unit Testing

- Camel provides extensions to JUnit tests
- A test context is created automatically
- A test template CamelTest.java is available
- Tests can change the source endpoint to inject test data
- Tests can change output endpoints to mock endpoints
- Mock endpoints can verify message count and content

Test Class

- There is a test class template CamelTest.java
- It defines an instance of the class under test
- It defines mock endpoints
- It defines a producer template to inject data

```
public class CamelTest extends CamelTestSupport {
    private Camel camel;
    private MockEndpoint result;
    private ProducerTemplate template;
}
```

Test Setup

- The setup is called before every test method
- It uses the class under test to create routes
- It creates a producer template to inject data
- It creates mock endpoints

```
@BeforeEach
void setup() throws Exception {
    camel = new Camel();
    context.addRoutes(camel);
    template = context.createProducerTemplate();
    result = context.getEndpoint("mock:out", MockEndpoint.class);
}
```

Modify Routes

- Route endpoints may not be available during tests
- Unit tests should be fast and not use external resources
- The Camel Advice allows endpoints to be modified
- Weave methods add, remove or change route

```
AdviceWith.adviceWith(context.getRouteDefinition("camel1"),
    context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() {
        weaveAddFirst().to("mock:first);
        replaceFromWith("direct:in");
        weaveById("out").replace().to("mock:out");
        weaveById("out").remove();
        weaveAddLast().to("mock:last);
    }
});
```

Test Method

- The test method sets expectations, injects data and verifies

```
@Test
void shouldChangeToUpperCase() throws Exception {
    AdviceWith.adviceWith(context.getRouteDefinition("camel1"),
        context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() {
            replaceFromWith("direct:in");
            weaveById("out").replace().to("mock:out");
        }
    });
    context.start();
    result.expectedBodiesReceived("HELLO WORLD");
    result.expectedMessageCount(1);
    template.sendBody("direct:in", "hello world");
    result.assertIsSatisfied();
}
```

Apache Camel

Exercise 2: Camel Unit Testing

- Copy exercise1 to exercise2
- Add a transformation route to Camel.java
- Add a test method to CamelTest.java
- Run the unit tests
- Run the application
- Full instructions are in CamelExercises.pdf

Apache Camel

File Endpoint

- The file endpoint takes a directory as a parameter
- A file consumer is used in `from()`
- Consumed files are moved to a `.camel` directory
- Files will not be moved if the `noop=true` parameter is set

```
@Override
public void configure() {
    from("file:../data/input?noop=true")
    .routeId("camel1")
    .to("file:../data/output");
}
```

Unit Testing

- The unit test will replace the file output endpoint with a mock

```
@Test
void filterOnFilenameTest() throws Exception  {
    AdviceWith.adviceWith(context.getRouteDefinition("camel1"),
        context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() {
            weaveById("out").replace().to("mock:out");
        }
    });
    context.start();
}
```

Unit Testing

- The message body contains the content of the file
- The test will just use the file name which is in a message header
- The test will only look at the first three messages

```
result.message(0).header("CamelFileName").isEqualTo("jabberwocky.txt");
result.message(1).header("CamelFileName").isEqualTo("jabberwocky1.txt");
result.message(2).header("CamelFileName").isEqualTo("jabberwocky2.txt");
result.expectedMessageCount(3);
result.assertIsSatisfied();
```

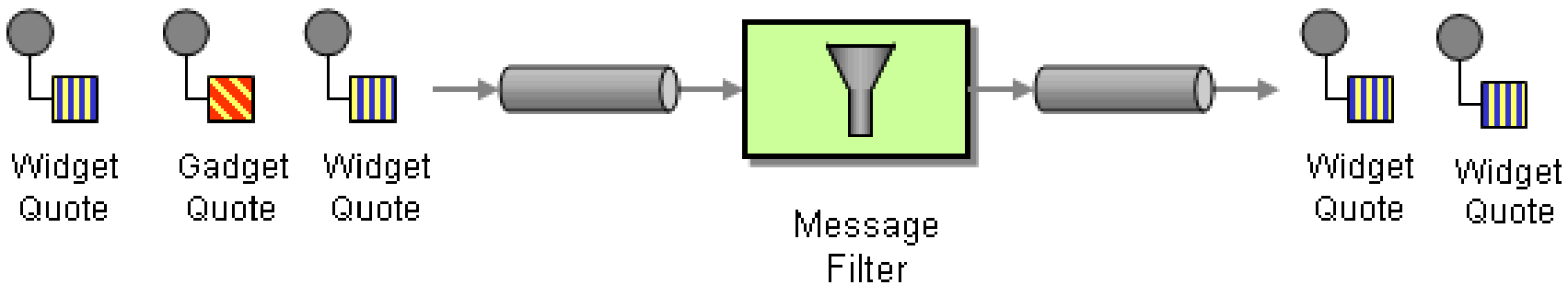
Test Failure

- The test will fail as the files will not be ordered
- The messages can be resequenced using the filename header
- The tests should now pass

```
@Override
public void configure() {
    from("file:../data/input?noop=true")
    .routeId("camel1")
    .resequence(header("CamelFileName"))
    .to("file:../data/output");
}
```

Filters

- A filter selects which messages will be processed
- Filters are often applied to message headers
- It is like an if statement
- Simple conditions compare message components to values



Filters

- The filter method has a comparison operation
- It is terminated by an end method
- Routes are only processed if the condition is true

```
.filter(simple("${headers.CamelFileName} regex '.*[235].txt'"))
    .to("stream:out").id("out")
.end();
```

Filter Expressions

- There are a number of fiter simple expressions

```
simple("${header.foo} == 'foo'") // Match
simple("${header.foo} =~ 'foo'") // Match ignore case
simple("${header.bar} == '100'") // Integers converted to strings
simple("${header.bar} == 100")   // Integers converted to strings
simple("${header.bar} > 100")    // Integers converted to strings
simple("${header.title} contains 'Camel'")
simple("${header.number} regex '\\d{4}'")
```

Apache Camel

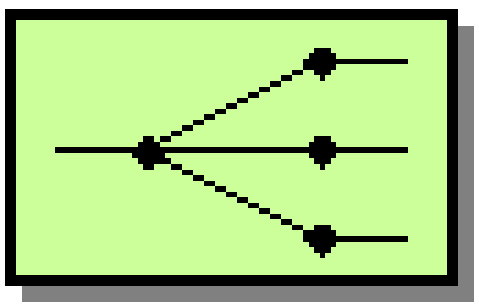
Exercise 3: Route Filters

- Copy exercise1 to exercise3
- Add a filter route to Camel.java
- Add a test method to CamelTest.java
- Run the unit tests
- Run the application
- Full instructions are in CamelExercises.pdf

Apache Camel

Multicast Routes

- Multicast routes send th same message to several endpoints
- Each endpoint can process the message differently
- Endpoints are processed sequentially
- They can be processed in parallel



Multicast

- The multicast sends the same message to several endpoints
- There can be another endpoint after the multicast

```
@Override
public void configure() {
    from("stream:in?promptMessage=Enter data: ")
    .routeId("camel1")
    .multicast()
        .to("direct:out1").id("out1")
        .to("direct:out2").id("out2")
        .to("direct:out3").id("out3")
    .end()
    .to("direct:out4").id("out4");
}
```

Multicast Processing

- Multicast uses a single thread by default
- Parallel processing uses a different thread for each endpoint
- It can also aggregate message bodies
- If an endpoint throws an exception all others are processed

```
.multicast().parallelProcessing()
.multicast(new MyAggregationStrategy()).parallelProcessing().timeout(500)
```

Multicast Testing

- A mock endpoint is required for each multicast endpoint
- Expectations need to be set on each mock endpoint
- Each mock end point needs to assert satisfied

Apache Camel

Exercise 4: Multicast Routes

- Copy exercise1 to exercise4
- Add a multicast route to Camel.java
- Add a test method to CamelTest.java
- Run the unit tests
- Run the application
- Full instructions are in CamelExercises.pdf

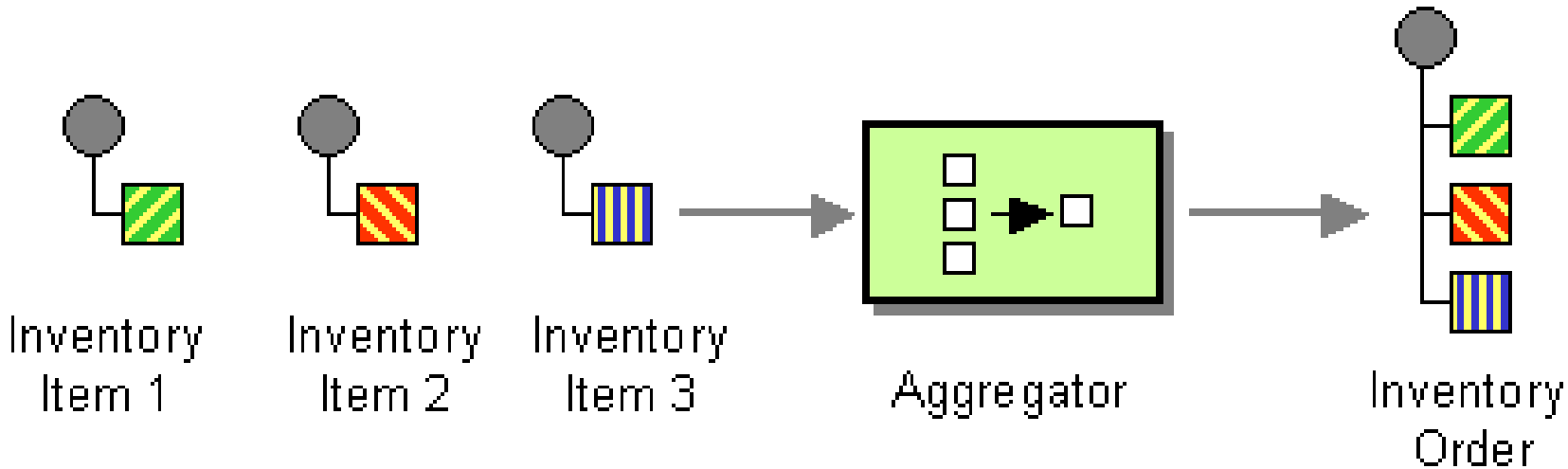
Message Exchanges

- Messages have a request message
- They can have a reply message
- They can have an exception message
- An Exchange is an interface used to exchange message components
- They support inbound only event messages and request and reply messages
- Endpoints have concrete implementations of Exchange
- EIP APIs use exchanges as abstractions

Apache Camel

Aggregator Routes

- An aggregator combines the bodies of one of more messages
- Headers control how messages are combined
- An aggregator class is required to implement it



Aggregator Class

- This aggregator combines message body strings
- It defines an aggregation strategy

```
package training;

import org.apache.camel.AggregationStrategy;
import org.apache.camel.Exchange;
import org.apache.camel.Message;

public class StringAggregator implements AggregationStrategy {
}
```

Aggregate Method

- This combines pairs of message body strings
- The first exchange will be null for the first message
- Both message bodies can be null

```
@Override
public Exchange aggregate(Exchange existing, Exchange next) {
    Exchange result = next;
    if (existing != null) {
        result = next;
        String body = existing.getIn().getBody(String.class);
        Message nextIn = next.getIn();
        String nextBody = nextIn.getBody(String.class);
        nextIn.setBody(body + "," + nextBody);
    }
    return result;
}
```

Aggregator Test Class

- Create a unit test class for the aggregator
- It needs to test the aggregator class in isolation

Test class

```
package training;

import org.apache.camel.ProducerTemplate;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit5.CamelTestSupport;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class AggregatorTest extends CamelTestSupport {
    private MockEndpoint result;
    private ProducerTemplate template;
}
```

Route Test Class

- Create an inner class to define the route

```
class Aggregate extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from( "direct:input" )
        .aggregate(header( "words" ), new StringAggregator())
        .completionTimeout(3000)
        .to("mock:out");
    }
}
```

Setup

- Set up the test class

```
@BeforeEach
void setup() throws Exception {
    context.addRoutes(new Aggregate());
    template = context.createProducerTemplate();
    result = context.getEndpoint("mock:out", MockEndpoint.class);
}
```

Test Method

- Create the test method
- The template sends the body, header name, and header value
- Different header names and values are aggregated separately

```
@Test
void aggregateWordsById() throws Exception {
    context.start();
    result.expectedMessageCount(2);
    result.expectedBodiesReceived("the,brown", "quick,fox");

    template.sendBodyAndHeader("direct:input", "the", "words", 1);
    template.sendBodyAndHeader("direct:input", "quick", "words", 2);
    template.sendBodyAndHeader("direct:input", "brown", "words", 1);
    template.sendBodyAndHeader("direct:input", "fox", "words", 2);
    result.assertIsSatisfied();
}
```

Apache Camel

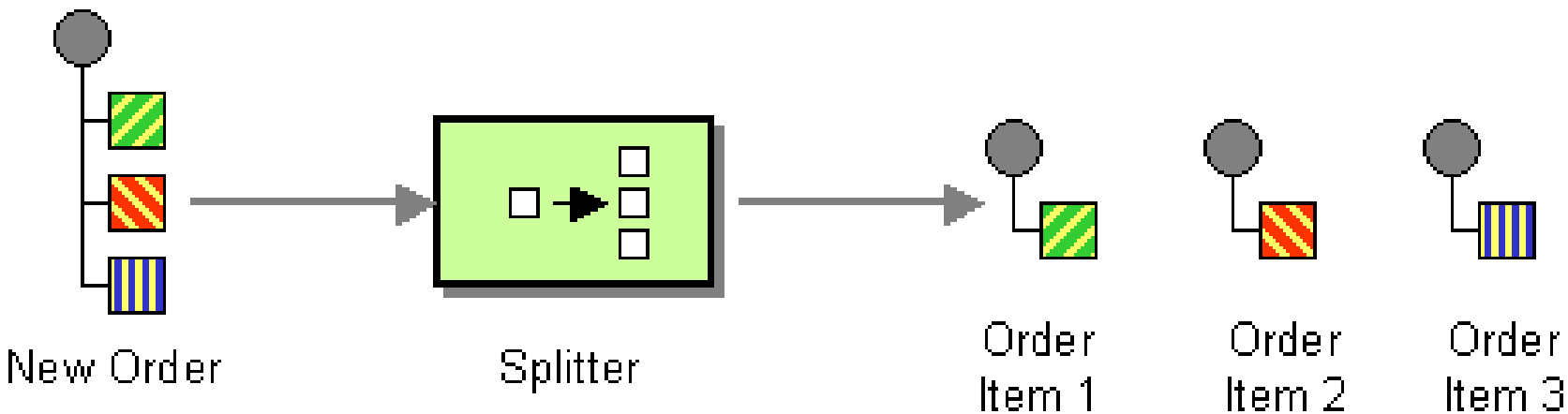
Exercise 5a: Aggregator

- Copy exercise1 to exercise5
- Create StringAggregator.java
- Create AggregatorTest.java
- Run the unit tests
- Full instructions are in CamelExercises.pdf

Apache Camel

Splitter Routes

- Splitter splits a message body into multiple messages
- It supports simple splitting of a message body
- It can also aggregate split components
- It can split strings by a delimiter character
- It can split lists into components



Split Body

- Splits arrays, lists, maps by entry
- Splits strings by commas

```
@Override
public void configure() {
    from("direct:input")
    .split(body())
    .to("direct:out");
}
```

Tokens

- Strings can be tokenised

```
@Override
public void configure() {
    from("direct:input")
    .split(body().tokenize("\n"))
    .to("direct:output");
}
```

Test Class

- We will use a test class for split
- It will be implemented differently to show options

```
package training;

import java.util.List;
import java.util.ArrayList;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit5.CamelTestSupport;
import org.junit.jupiter.api.Test;

public class SplitterTest extends CamelTestSupport {
}
```

Route Builder

- We will create a route builder to define routes

```
  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
  return new RouteBuilder() {
          @Override
          public void configure() throws Exception {
              from("direct:input")
              .split(body())
              .to("mock:out") ;
          }
      } ;
}
```

Test Method

- We create a test method to send a list as a body

```
@Test
void splitList() throws Exception {
    MockEndpoint result = getMockEndpoint("mock:out");
    result.expectedMessageCount(3);
    result.expectedBodiesReceived("quick", "brown", "fox");

    List<String> body = new ArrayList<>();
    body.add("quick");
    body.add("brown");
    body.add("fox");
    template.sendBody("direct:input", body);

    result.assertIsSatisfied();
}
```

Apache Camel

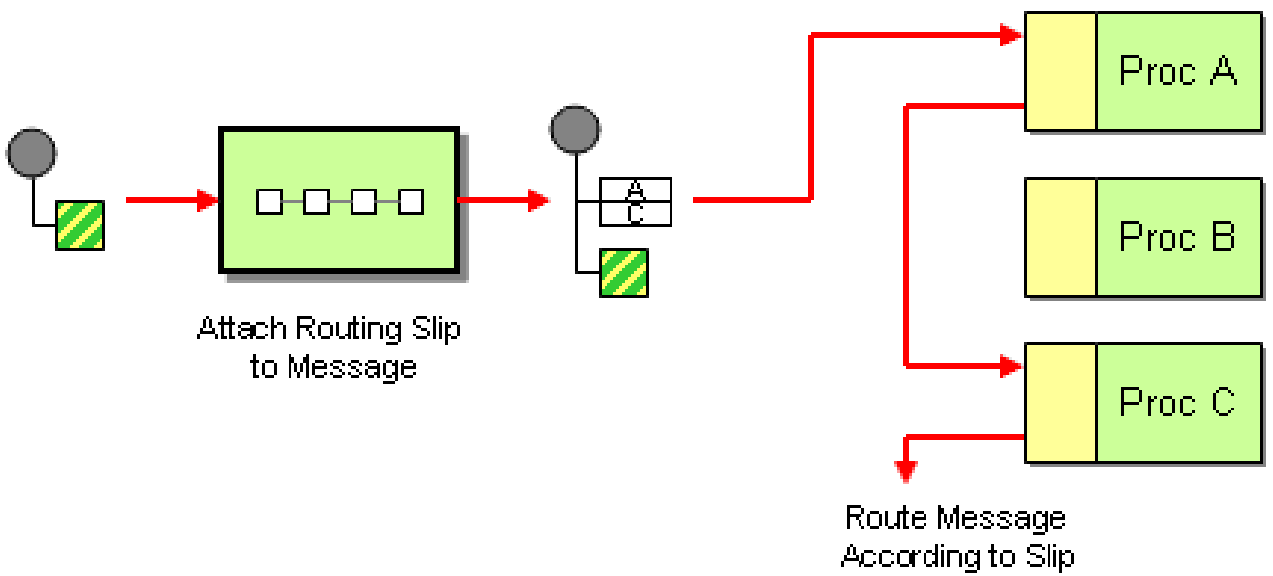
Exercise 5b: Splitter

- Create SplitterTest.java
- Run the unit tests
- Full instructions are in CamelExercises.pdf

Apache Camel

Routing Slip

- Messages get routed through a sequence of endpoints
- What if the sequence is not known in advance?
- What if the sequence is different for different messages?
- Routing slip gets the routing sequence from a header



Test Class

- We will use a test class for routing slip

```
package training;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit5.CamelTestSupport;
import org.junit.jupiter.api.Test;

public class RoutingSlipTest extends CamelTestSupport {
}
```

Route Builder

- We will create a route builder to define routes

```
  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
  return new RouteBuilder() {
          @Override
          public void configure() throws Exception {
              from("direct:input")
              .routingSlip("slip")
          }
      } ;
}
```

Test Method

- We will create a test class that defines three mock endpoints

```
@Test
void routingSlip() throws Exception {
    MockEndpoint mocka = getMockEndpoint("mock:a");
    mocka.expectedMessageCount(3);
    mocka.expectedBodiesReceived("alpha", "beta", "gamma");

    MockEndpoint mockb = getMockEndpoint("mock:b");
    mockb.expectedMessageCount(1);
    mockb.expectedBodiesReceived("beta");

    MockEndpoint mockc = getMockEndpoint("mock:c");
    mockc.expectedMessageCount(1);
    mockc.expectedBodiesReceived("gamma");
}
```

Test Method

- We will send messages with routes in the slip header
- The assert is applied to all mock endpoints in the context

```
template.sendBodyAndHeader("direct:input", "alpha", "slip", "mock:a");
template.sendBodyAndHeader("direct:input", "beta", "slip", "mock:a,mock:b");
template.sendBodyAndHeader("direct:input", "gamma", "slip", "mock:a,mock:c");

MockEndpoint.assertIsSatisfied(context);
```

Apache Camel

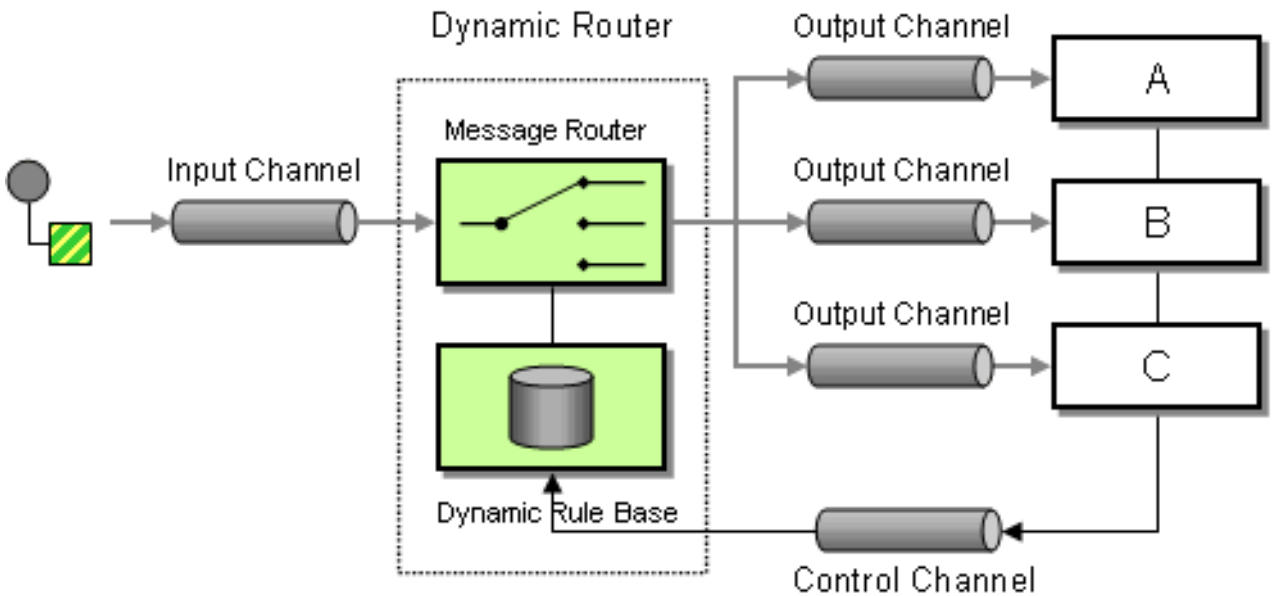
Exercise 5c: Routing Slip

- Create RoutingSlipTest.java
- Run the unit tests
- Full instructions are in CamelExercises.pdf

Apache Camel

Dynamic Router

- A dynamic router obtains route from a Java Bean
- It can provide several routes
- A null route terminates the router



Bean Class

- Create a routing bean
- The route number is stored in properties

```
package training;
import java.util.Map;
import org.apache.camel.ExchangeProperties;

public class RouterBean {
    public String slip(String body,
        @ExchangeProperties Map<String, Object> properties) {
        int invoked = (int) properties.getOrDefault("invoked", 0) + 1;
        properties.put("invoked", invoked);
    }
}
```

Bean Class Routing

- The route is determined by the invokation number

```
String route = null;
if (invoked == 1) {
    route = "mock:a";
} else if (invoked == 2) {
    route = "mock:a,mock:b";
} else if (invoked == 3) {
    route = "mock:b";
}
return route;
```

Test Class

- We will use a test class for dynamic router

```
package training;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit5.CamelTestSupport;
import org.junit.jupiter.api.Test;

public class DynamicRouterTest extends CamelTestSupport {
}
```

Route Builder

- We will create a route builder to define routes

```
  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
  return new RouteBuilder() {
          @Override
          public void configure() throws Exception {
              from("direct:input")
              .dynamicRouter(method(RouterBean.class, "slip"))
              .to("mock:c");
          }
      } ;
}
```

Test Method

- We will create a test class that defines three mock endpoints

```
@Test
void routingSlip() throws Exception {
    MockEndpoint mocka = getMockEndpoint("mock:a");
    mocka.expectedMessageCount(3);
    mocka.expectedBodiesReceived("message", "message");

    MockEndpoint mockb = getMockEndpoint("mock:b");
    mockb.expectedMessageCount(1);
    mockb.expectedBodiesReceived("message", "message");

    MockEndpoint mockc = getMockEndpoint("mock:c");
    mockc.expectedMessageCount(1);
    mockc.expectedBodiesReceived("message");
}
```

Test Method

- Send a single message to the route

```
template.sendBody("direct:input", "message");

MockEndpoint.assertIsSatisfied(context);
```

Apache Camel

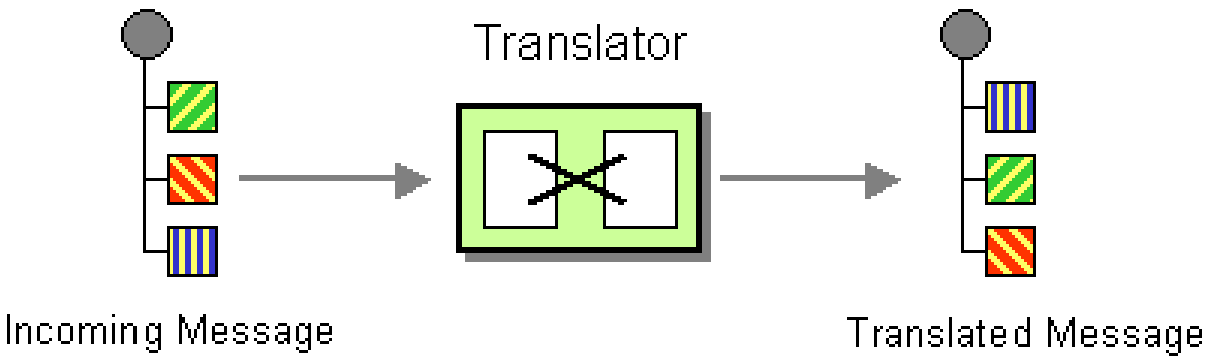
Exercise 5d: Dynamic Router

- Create RouterBean.java
- Create DynamicRouterTest.java
- Run the unit tests
- Full instructions are in CamelExercises.pdf

Apache Camel

Message Translator

- Transform change the message body
- There are several ways of doing this
- A null route terminates the router



Transforms

- Simple transformations can modify the message body

```
@Override
public void configure() {
    from("stream:in?promptMessage=Enter data: ")
    .routeId("camel1")
    .transform(simple("Transformed ${body.toUpperCase()}"))
    .to("stream:out").id("out");
}
```

Transforms

- Data format transformers are provided

```
@Override
public void configure() {
    transformer().name("base64")
    .withDataFormat(dataFormat().base64().end());
    DataType base64 = new DataType("base64");
    from("direct:input")
    .transform(base64)
    .to("mock:out");
}
```

Transforms

- Test method for base64 encoding

```
@Test
void transformation() throws Exception {
    MockEndpoint mocka = getMockEndpoint("mock:out");
    mocka.expectedMessageCount(1);
    mocka.expectedBodiesReceived("YWJj\r\n");

    template.sendBody("direct:input", "abc");
    MockEndpoint.assertIsSatisfied(context);
}
```

Bean Transformer

- A Java Bean can be a transformer class

```
package training;

import org.apache.camel.Message;
import org.apache.camel.spi.DataType;
import org.apache.camel.spi.DataTypeTransformer;
import org.apache.camel.spi.Transformer;

@DataTypeTransformer(name = "wordcase")
public class CaseTransformer extends Transformer{
}
```

Bean Transformer

- A transform method can rewrite a message body
- It capitalises the first letter of each word

```
@Override
public void transform(Message message, DataType from, DataType to)
    throws Exception {
    char[] chars = message.getBody(String.class).toCharArray();
    boolean inword = false;
    for (int i = 0; i < chars.length; i++) {
        if (!inword && Character.isLetter(chars[i])) {
            chars[i] = Character.toUpperCase(chars[i]);
            inword = true;
        } else if (Character.isWhitespace(chars[i])) {
            inword = false;
        }
    }
    message.setBody(String.valueOf(chars));
}
```

Test Route Builder

- The route defines the bean as a transformer

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            transformer().name("wordcase")
             .withJava(CaseTransformer.class);
            from("direct:input")
            .transform(new DataType("wordcase"))
            .to("mock:out");
        }
    };
}
```

Test Method

- The test method for the transformer bean

```
@Test
void transformation() throws Exception {
    MockEndpoint mocka = getMockEndpoint("mock:out");
    mocka.expectedMessageCount(1);
    mocka.expectedBodiesReceived("The Quick Brown Fox");

    template.sendBody("direct:input", "the quick brown fox");
    MockEndpoint.assertIsSatisfied(context);
}
```

Apache Camel

Exercise 5e: Bean Transformer

- Use a Java Bean Transformer to reverse the characters of the message body string
- Create ReverserBean.java
- Create ReverserBeanTest.java
- Run the unit tests
- Instructions are in CamelExercises.pdf

Apache Camel

CSV, XML and JSON

- Camel has support for different file types
- It can marshal Java objects into file formats
- It can unmarshal fime content into Java objects
- Annotations are used to identify fields to be translated
- Java classes can be annotaated for more than one format

Animal Class

- Start with a class Animal.java
- Each animal has a name, body mass, and brain mass

```
package training;

public class Animal {
    private String name;
    private double body;
    private double brain;

    // Define getter and setter methods

    public String toString() {
        return String.format("%s body %.2fkg brain %.2fg", name, body, brain);
    }
}
```

Camel Bindy

- Animal data is in data/csv/animals.csv
- Camel Bindy can marshal and unmarshal CSV
- Annotations are required on the java class

```
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",", skipFirstLine = true)
public class Animal {
    @DataField(pos = 1)
    private String name;
    @DataField(pos = 2)
    private double body;
    @DataField(pos = 3)
    private double brain;
}
```

Router

- Create a route in Camel.java

```
@Override
public void configure() throws Exception {
    DataFormat bindy = new BindyCsvDataFormat(Animal.class);

    from("file:../data/csv?fileName=animals.csv&noop=true")
    .unmarshal(bindy)
    .to("direct:toString");
}
```

Router

- Create a second route to display the animals

```
from("direct:toString")
.split(body())
    .transform(simple("${body.toString()}"))
    .to("stream:out")
.end();
```

Run Camel

- Run the program to see the list of animals
- Type control-C to terminate the wait

```
mvn camel:run
```

XML Files

- XML files can be processed using JAXB
- It used to be part of JEE but is now in Jakarta
- The package javax.xml became jakarta.xml
- It provides annotations
- It provides a Camel DataFormat

Wrapper Class

- The CSV file contains a list of animals
- It requires a wrapper class Animals.java for the XML root
- The annotation defones the XML root

```
package training;

import java.util.List;
import java.util.ArrayList;
import jakarta.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "animals")
public class Animals {
    private List<Animal> animals = new ArrayList<>();
}
```

Wrapper Class

- Define getter, setter and add methods

```
public List<Animal> getAnimals() {
    return animals;
}

public void addAnimal(Animal animal) {
    animals.add(animal);
}

public void setAnimals(List<Animal> animals) {
    this.animals.addAll(animals);
}
```

Annotations

- Add XML annotations to Animal.java

```
import jakarta.xml.bind.annotation.XmlAccessType;
import jakarta.xml.bind.annotation.XmlAccessorType;
import jakarta.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "animal")
@XmlAccessorType(XmlAccessType.FIELD)
@CsvRecord(separator = ",", skipFirstLine = true)
public class Animal {
}
```

Processors

- A Camel Processor is a Java class called from a route
- It must have a process() method
- It has an Exchange parameter which contains the message
- It replaces the message body

Animal List Processor

- Write the class AnimalListProcessor.java
- The input message is an array of Animal

```
package training;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class AnimalListProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        Animals animals = new Animals();
        for (Animal animal : exchange.getIn().getBody(Animal[].class)) {
            animals.addAnimal(animal);
        }
        exchange.getIn().setBody(animals);
    }
}
```

Add Route

- Add an XML data format
- Change the route target

```
import jakarta.xml.bind.JAXBContext;
import org.apache.camel.converter.jaxb.JaxbDataFormat;

    JaxbDataFormat xmlDataFormat = new JaxbDataFormat();
    JAXBContext con = JAXBContext.newInstance(Animals.class);
    xmlDataFormat.setContext(con);

    //.to("direct:toString");
    .to("direct:toXml");
```

Add Route

- Add a to XML route
- Run Camel

```
from("direct:toXml")
.process(new AnimalListProcessor())
.marshal(xmlDataFormat)
.to("stream:out");
```

JSON Files

- JSON files can be processed using Jackson
- It can use XML annotations
- It has a few quirks that can result in empty JSON
- It provides annotations
- It provides a Camel DataFormat

Annotations

- Add JSON annotations to Animal.java
- All getter methods need annotating

```
import com.fasterxml.jackson.annotation.JsonGetter;

    @JsonGetter
    public String getName() {
        return name;
    }
```

Global Options

- Enable Jackson conversion in Cale constructor

```
public Camel() {
    logger = LogManager.getLogger(getClass());
    context = new DefaultCamelContext();
    context.getGlobalOptions().put("CamelJacksonEnableTypeConverter", "true");
    context.getGlobalOptions().put("CamelJacksonTypeConverterToPojo", "true");
}
```

Add Route

- Change the route target
- Add JSON route and run Camel

```
import org.apache.camel.model.dataformat.JsonLibrary;

    //.to("direct:toString");
    //.to("direct:toXml");
    .to("direct:toJson");

    from("direct:toJson")
    .marshal().json(JsonLibrary.Jackson)
    .to("stream:out");
```

Apache Camel

Exercise 6: CSV, XML and JSON

- Copy exercise1 to exercise6
- Create Animal.java annotated for CSV
- Create a route in Camel.java
- Run the program
- Add the code to output the animal list as XML
- Run the program
- Add the code to output the animal list as JSON
- Run the program
- Instructions are in CamelExercises.pdf

Camel Components

- Camel components are endpoints to external services
- They have URIs for `from()` and `to()`
- We have already seen the file, stream, and timer endpoints
- We have used mock endpoints for testing
- Others are provided by adding dependencies to the POM
- Custom endpoints can be written

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

Apache Camel

Apache MINA

- Multipurpose Infrastructure for Network Applications
- Java network application framework
- APIs for TCP and UDP
- It also supports direct transport in the same JVM
- Provides low level APIs
- Has mocks for unit testing
- The online documentation is very vague!
- It requires the `camel-mina` dependency

String Processor

- Create a class StringProcessor.java

```
package training;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class StringProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getIn().setBody("Processed " + body);
    }
}
```

Test Class

- Create a test class MINATest.java
- Define an endpoint URI

```
package training;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.junit5.CamelTestSupport;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class MINATest extends CamelTestSupport {
    private static final String stringURI =
        "mina:tcp://localhost:6200?textline=true&sync=true";
    private MockEndpoint result;
}
```

Route Builder Class

- Create a MINIRoute inner class defining a route
- Add route to context and create mock endpoint

```
class MINARoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from(stringURI)
        .process(new StringProcessor())
        .to("mock:result");
    }
}

@BeforeEach
void setup() throws Exception {
    context.addRoutes(new MINARoute());
    result = context.getEndpoint("mock:result", MockEndpoint.class);
}
```

Test Method

- Create a MINASendMessage test class and run the tests

```
@Test
void MINASendMessage() throws Exception {
    result.expectedMessageCount(1);
    result.expectedBodiesReceived("Processed message");

    String response = (String)template.requestBody(stringURI, "message");
    Assertions.assertEquals("Processed message", response);
    result.assertIsSatisfied();
}
```

POJO

- Create a POJO Creature.java
- Add getters and setters and a `toString()` method

```
public class Creature implements Serializable{
    private String name;
    private int legs;
    private boolean flies;

    public Creature(String name, int legs, boolean flies) {
        this.name = name;
        this.legs = legs;
        this.flies = flies;
    }
}
```

Creature Processor

- Create a processor to look up a creature by name

```
public class CreatureProcessor implements Processor {
    private static Map<String, Creature> creatures;

    static {
        creatures = new HashMap<>();
        creatures.put("bat", new Creature("Bat", 2, true));
        creatures.put("penguin", new Creature("Penguin", 2, false));
    }

    @Override
    public void process(Exchange exchange) throws Exception {
        String name = exchange.getIn().getBody(String.class);
        exchange.getIn().setBody(creatures.get(name));
    }
}
```

Add Route

- Define a URI for the route
- Warning, the POJO class name pattern is required here!
- Add a route

```
private static final String objectURI =
    "mina:tcp://localhost:6201?sync=true&objectCodecPattern=*";

        from(objectURI)
        .process(new CreatureProcessor());
```

Add test Cases

- Add test cases for the POJO
- Run the tests

```
Creature creature = (Creature)template.requestBody(objectURI, "bat");
Assertions.assertEquals("Bat 2 legs flies", creature.toString());

creature = (Creature)template.requestBody(objectURI, "penguin");
Assertions.assertEquals("Penguin 2 legs", creature.toString());
```

Apache Camel

Exercise 7a: Components

- Copy exercise1 to exercise7
- Create StringProcessor.java
- Create MINATest.java
- Run the tests
- Add a POJO of your choosing
- Create a processor to return a POJO object by name
- Add tests for the POJO
- Run the tests
- Instructions are in CamelExercises.pdf

Apache Camel

Camel JDBC

- Camel JDBC and Camel SQL provide database component endpoints
- This requires more plumbing code to work
- It requires a database data source
- SQL statements are route parameters
- The online documentation is very very vague!
- It requires the `camel-mina` `camel-sql` and database driver dependencies

Postgres Database

- A Postgres database can be run in a Docker container
- The `initdb` script sets up the database
- The `rundb` script starts or restarts the database
- The Posgress CLI has been installed
- A table needs to be created

```
docker ps
./rundb

psql -h localhost -U camel cameldb
cameldb# \i animal.sql
cameldb# \d animal
cameldb# \q
```

Test Class

- Create JdbcTest.java
- It needs a database URL and data source

```
import org.apache.commons.dbcp2.BasicDataSource;

public class JdbcTest extends CamelTestSupport {
    private static final String dburi =
      "jdbc:postgresql://localhost:5432/cameldb";
    private static BasicDataSource dataSource;
    private ProducerTemplate template;
    private MockEndpoint result;
}
```

Initial Setup

- Create a data source to connect to the database

```
@BeforeAll
static void initialSetup() throws Exception {
    dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUsername("camel");
    dataSource.setPassword("camel");
    dataSource.setUrl(dburi);
}
```

Pretest Setup

- Create a pre-test setup that registers the data source

```
@BeforeEach
void setup() throws Exception {
    context.getRegistry().bind("PostgresDataSource", dataSource);
    context.addRoutes(new JdbcRoute());
    template = context.createProducerTemplate();
    result = getMockEndpoint("mock:result");
    context.start();
}
```

Insert Processor

- Create a processor CreatureInsertProcessor.java to insert an object

```
public class CreatureInsertProcessor implements Processor {
    private String format;

    CreatureInsertProcessor() {
        format = "insert into animal (name, legs, flies) " +
                 "values ('%s', %d, %b)";
    }

    @Override
    public void process(Exchange exchange) throws Exception {
        Creature creature = exchange.getIn().getBody(Creature.class);
        exchange.getIn().setBody(String.format(format,
            creature.getName(), creature.getLegs(), creature.isFlies()));
    }
}
```

Route Class

- Create an inner class to define a route
- The end point is a data source that expects SQL

```
class JdbcRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:insert")
        .process(new CreatureInsertProcessor())
        .to("jdbc:PostgresDataSource")
        .log("${body}");
    }
}
```

Test Method

- Create test method
- Exception handing is required as exceptions can be cryptic!

```
@Test
void animalDatabase() throws Exception {
    Creature bat = new Creature("Bat", 2, true);
    Creature penguin = new Creature("Penguin", 2, false);
    try {
        template.sendBody("direct:insert", bat);
        template.sendBody("direct:insert", penguin);
    } catch (Exception e) {
        System.err.println(e);
        e.printStackTrace(System.err);
    }
}
```

Test Method

- Run the tests
- If an exception occurs look for the caused by exception trace
- Verify that the objects have been inserted

```
psql -h localhost -U camel cameldb
select * from animal;
\q
```

Delete Route

- The tests will fail if run again due to database constraints
- Add a route to delete the database records

```
from("direct:delete")
.to("jdbc:PostgresDataSource");
```

Test Delete

- Add a template send to the delete route and run tests

```
template.sendBody("direct:delete", "delete from animal");
template.sendBody("direct:insert", bat);
template.sendBody("direct:insert", penguin);
```

Select Processor

- Create CreatureSelectProcessor.java
- Java generics were very badly designed!

```
public class CreatureSelectProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        @SuppressWarnings("unchecked")
        Map<String,Object> creatureMap =
          (Map<String,Object>)exchange.getIn().getBody();
        Creature creature = new Creature((String)creatureMap.get("name"),
                    (Integer)creatureMap.get("legs"),
                    (Boolean)creatureMap.get("flies"));
        exchange.getIn().setBody(creature.toString());
    }
}
```

Select Route

- Add a route to select from the database

```
from("direct:select")
.to("jdbc:PostgresDataSource")
.split(body())
    .process(new CreatureSelectProcessor())
    .log("${body}")
    .to("mock:result")
.end();
```

Test Route

- Add expectations and send for select and run test

```
result.expectedBodiesReceived("Bat 2 legs flies","Penguin 2 legs");
result.expectedMessageCount(2);
// At end of try
template.sendBody("direct:select", "select name,legs,flies from animal");
// At end of method
esult.assertIsSatisfied();
```

Apache Camel

Exercise 7b: Database

- Create StringProcessor.java
- Create MINATest.java
- Run the tests
- Add a POJO of your choosing
- Create a processor to return a POJO object by name
- Add tests for the POJO
- Run the tests
- Instructions are in CamelExercises.pdf

Apache Camel

Project

- This is the end of the course material
- Any questions?
- We will now do a project either individually or pairs
- There is some data on over 300 penguins used for training AI models
- Penguin species can be identified by body mass, flipper length and beak dimensions!
- Use Camel features to process the CSV file, store in a database, and query penguin data
- Instructions are in CamelExercises.pdf

Thank you for attending
Any Questions?