OrderFlow Scripting Guide

OrderFlow Ltd.

Document Version: 4.2.4

Document Built: 2024-02-16

Overview

This document provides details of the scripting capabilities available on OrderFlow, and how to access and use them. It also provides a reference point for the different scripts that are configured on OrderFlow, with examples of how they are used, and what is available in the scripting context for each of the respective script types.

Audience

The intended audience of this document is:

- the OrderFlow support team, who use OrderFlow's scripting capability to implement solutions for customers.
- customer technical users, who can use OrderFlow's scripting to implement their own solutions.

OrderFlow Scripting API

Most of the scripts run on OrderFlow use the OrderFlow API. It is very useful to know how to access specific data items using Groovy scripting expressions.

Order

The following fields are the **top level** fields for the order.

Top Level Order Data

Groovy Expression	Database Column	Type	Description
order.externalReference	*externalReference*	string	Unique and human-readable reference for the order.
order.originatingExternalReference	*originatingExternalReference*	string	Typically used when the order is received via a third party system where it is identified using a different reference.
order.source	*source*	string	Used to denote the source of the order.
order.brand	*brand*	string	If multiple brands are sold on the same channel, this field can be used to identify the brand.
order.type	*type*	string	Used to distinguish between different types of sales orders.
order.paymentGatewayIdentifier	*paymentGatewayIdentifier*	string	Identifies the payment gateway on which the financial transaction for the order was made.
order.paymentTransactionInfo	*paymentTransactionInfo*	string	Further info used to distinguish the order. Only useful if further manual validation of the order may be required.
order.customerComment	*customerComment*	string	Holds a customer comment place on the order.
order.requiresApproval	*requiresApproval*	boolean	If set order requires approval before any shipments should be picked.

The following fields are the **system** fields for the order.

System Data

Groovy Expression	Database Column	Type	Description
order.multiShipment	*multiShipment*	string	
order.systemReserved	*systemReserved*	boolean	
order.state	*state*	string	
order.deleted	*deleted*	boolean	
order.commented	*commented*	boolean	

The following fields are the **channel** fields for the order.

Channel Data

Groovy Expression	Database Column	Type	Description
order.channel.externalReference	*externalReference*	string	Unique identifier for the sales channel on which the order was placed.
order.channel.name	*name*	string	Name of the sales channel.

The following fields are the **price** fields for the order.

Price Data

Groovy Expression	Database Column	Type	Description
order.currency	*currency*	string	"Three character SO currency code, for example 'GBP', 'USD'."
order.totalPrice.net	*totalPriceNet*	double	"The net total price value, before tax."
order.totalPrice.gross	*totalPriceGross*	double	"The gross total price, after tax."
order.totalPrice.tax	*totalTax*	double	The tax payable on the gross price.
order.totalPrice.taxCode	*totalTaxCode*	double	The tax code for the tax on on the total price.
order.shippingPrice.net	*shippingPriceNet*	double	"The net shipping price value, before tax."
order.shippingPrice.gross	*shippingPriceGross*	double	The gross price of the shipment.
order.shippingPrice.tax	*shippingTax*	double	The tax payable. Net price is derived as gross price less tax.
order.shippingPrice.taxCode	*shippingTaxCode*	double	The tax code for the tax payment.
order.goodsPrice.net	*goodsPriceNet*	double	"The net price value, before tax."
order.goodsPrice.gross	*goodsPriceGross*	double	The gross price of the goods.
order.goodsPrice.tax	*goodsTax*	double	The tax payable. Net price is derived as gross price less tax.
order.goodsPrice.taxCode	*goodsTaxCode*	double	The tax code for the tax payment.
order.promotionCode	*promotionCode*	string	Holds order level promotions/discounts - primarily usable for management reporting (e.g. winter_sale_20).
order.promotionDescription	*promotionDescription*	string	Holds human readable analogue of *promotionCode* (e.g. Winter Sale - 20% Discount).

The following fields are the **delivery address** fields for the order.

Delivery Address Data

Groovy Expression	Database Column	Type	Description
order.deliveryAddress.line1	*deliveryAddressLine1*	string	First line of the order delivery address.
order.deliveryAddress.line2	*deliveryAddressLine2*	string	Second line of the order delivery address.
order.deliveryAddress.line3	*deliveryAddressLine3*	string	Third line of the order delivery address.
order.deliveryAddress.line4	*deliveryAddressLine4*	string	Fourth line of the order delivery address.
order.deliveryAddress.line5	*deliveryAddressLine5*	string	Fifth line of the order delivery address.
order.deliveryAddress.line6	*deliveryAddressLine6*	string	Sixth line of the order delivery address.
order.deliveryAddress.countryCode	*countryCode*	string	The country code. Generally a two character code e.g GB.
order.deliveryAddress.postCode	*postCode*	string	Postal code.

The following fields are the **delivery contact** fields for the order.

Delivery Contact Data

Groovy Expression	Database Column	Type	Description
order.deliveryContact.contactName	*deliveryContactName*	string	Abbreviated contact name used as an alternative to the salutation/ firstName/lastName combination.
order.deliveryContact.salutation	*deliverySalutation*	string	Delivery contact salutation.
order.deliveryContact.firstName	*deliveryFirstName*	string	Delivery contact first name.
order.deliveryContact.lastName	*deliveryLastName*	string	Delivery contact last name.
order.deliveryContact.emailAddress	*deliveryEmailAddress*	string	Delivery contact email address.
order.deliveryContact.dayPhoneNumber	*deliveryDayPhoneNumber*	string	Delivery contact day phone number.
order.deliveryContact.eveningPhoneNumber	*deliveryEveningPhoneNumber*	string	Delivery contact evening phone number.
order.deliveryContact.mobilePhoneNumber	*deliveryMobilePhoneNumber*	string	Delivery contact mobile phone number.
order.deliveryContact.faxNumber	*deliveryFaxNumber*	string	Delivery contact fax number.
order.deliveryContact.companyName	*deliveryCompanyName*	string	Delivery contact company name.

The following fields are the **invoice address** fields for the order.

Invoice Address Data

Groovy Expression	Database Column	Type	Description
order.invoiceAddress.line1	*invoiceAddressLine1*	string	First line of the order invoice address.
order.invoiceAddress.line2	*invoiceAddressLine2*	string	Second line of the order invoice address.
order.invoiceAddress.line3	*invoiceAddressLine3*	string	Third line of the order invoice address.
order.invoiceAddress.line4	*invoiceAddressLine4*	string	Fourth line of the order invoice address.
order.invoiceAddress.line5	*invoiceAddressLine5*	string	Fifth line of the order invoice address.
order.invoiceAddress.line6	*invoiceAddressLine6*	string	Sixth line of the order invoice address.
order.invoiceAddress.countryCode	*invoiceAddressCountryCode*	string	The country code. Generally a two character code e.g GB.
order.invoiceAddress.postCode	*invoiceAddressPostCode*	string	Postal code.

The following fields are the **invoice contact** fields for the order.

Invoice Contact Data

Groovy Expression	Database Column	Type	Description
order.invoiceContact.contactName	*invoiceContactName*	string	Abbreviated contact name used as an alternative to the salutation/ firstName/lastName combination.
order.invoiceContact.salutation	*invoiceSalutation*	string	Invoice contact salutation.
order.invoiceContact.firstName	*invoiceFirstName*	string	Invoice contact first name.
order.invoiceContact.lastName	*invoiceLastName*	string	Invoice contact last name.
order.invoiceContact.emailAddress	*invoiceEmailAddress*	string	Invoice contact email address.
order.invoiceContact.dayPhoneNumber	*invoiceDayPhoneNumber*	string	Invoice contact day phone number.
order.invoiceContact.eveningPhoneNumber	*invoiceEveningPhoneNumber*	string	Invoice contact evening phone number.
order.invoiceContact.mobilePhoneNumber	*invoiceMobilePhoneNumber*	string	Invoice contact mobile phone number.
order.invoiceContact.faxNumber	*invoiceFaxNumber*	string	Invoice contact fax number.
order.invoiceContact.companyName	*invoiceCompanyName*	string	Invoice contact company name.

The following fields are the **user defined** fields for the order. Note that the usage of individual user defined fields is not predefined, and depends entirely on the environment.

User Defined Data

Groovy Expression	Database Column	Type
order.userDefined.userDefined1	*userDefined1*	string
order.userDefined.userDefined2	*userDefined2*	string
order.userDefined.userDefined3	*userDefined3*	string
order.userDefined.userDefined4	*userDefined4*	string
order.userDefined.userDefined5	*userDefined5*	string

The following fields are the **date** fields for the order.

Date Data

Groovy Expression	Database Column	Type	Description
order.created	*created*	string	The date/time that the order was created on the system.
order.lastUpdated	*lastUpdated*	string	The date/time that the order was last updated.
order.completed	*completed*	string	The date/time that the order was completed.
order.placed	*placed*	string	The date/time that the order was placed on the third party system.
order.authorised	*authorised*	string	The date/time that the order was approved by the relevant payment gateway (if available).
order.exported	*exported*	string	The date/time at which the order was exported from an external system.

The following fields are the **implicit** fields for the order. Implicit fields are fields for which the underlying data may differ, and typically involve default or fallback sources of the data if the primary data item is not present.

Implicit Data

Groovy Expression	Database Column	Type	Description
order.implicitInvoiceAddress	*n/a*	Address	Returns *InvoiceAddress* if this is set - otherwise returns *DeliveryAddress*.
order.implicitInvoiceContact	*n/a*	Contact	Returns *InvoiceContact* if this is set - otherwise returns *DeliveryContact*.
order.implicitPlaced	*n/a*	datetime	Returns the date placed if this is set - otherwise returns date created.
order.implicitAuthorised	*n/a*	datetime	Returns the date authorised if this is set - otherwise returns date created.

Order Attribute

A particular order attribute can be obtained from an order using the following Groovy.

```
def orderAttribute = order.getAttribute('order_attribute_1');
```

With the reference to the attribute, the `name`, `title` and `value` of the attribute can be obtained using the following code:

```
def orderAttribute = order.getAttribute('attribute_a');

println 'Attribute Name: ' + orderAttribute.name
println 'Attribute Title: ' + orderAttribute.title
println 'Attribute Value: ' + orderAttribute.value
```

For example, to obtain the value of a known attribute, you can use the following expression:

```
println order.getAttribute('pet_name')?.value;
```

Shipment

The following fields are the **top level** fields for the shipment.

Top Level Shipment Data

Groovy Expression	Database Column	Type	Description
shipment.externalReference	*externalReference*	string	Unique and Human-readable reference for the Shipment.
shipment.thirdPartReference	*thirdPartyReference*	string	Optional reference for shipment by which a third party system might identify it.
shipment.state	*state*	string	The current state of the shipment.
shipment.paidFor	*paidFor*	boolean	Persists whether this shipment has been paid for; that is a successful response to a payment request has been received.
shipment.weight	*weight*	string	The weight of the shipment.
shipment.weightUnits	*weightUnits*	string	The unit measurement.
shipment.pickingMode	*pickingMode*	string	The mode used for picking for this shipment. Used to indicate that the shipment is to be picked individually in a batch or via cross docking.
shipment.packageCount	*packageCount*	integer	The number of packages in this shipment.
shipment.packageState	*packageState*	string	Not null if multiple packages are being used to handle shipment despatching.

The following fields are the **system** fields for the shipment.

System Data

Groovy Expression	Database Column	Type	Description
shipment.sequence	*sequence*	string	???.
shipment.originatingShipmentId	*originatingShipmentId*	integer	The identity of the originating shipment from which this shipment was split off. Applies only for split shipments.
shipment.multiLine	*multiLine*	boolean	Holds persistently whether a shipment is multiline so that the order line collection does not need to be loaded to determine this. It is the responsibility of the application to ensure that this value is set correctly.
shipment.hasWarnings	*hasWarnings*	boolean	Flag to indicate that the shipment has warnings which may be pack warnings or otherwise. Once set this typically won't change although for a cloned shipment it should be set to false. Should always be true if shipment warnings is populated
shipment.progressIndication	*progressIndication*	integer	Holds the progress indicator for an order line.
shipment.systemReserved	*systemReserved*	string	A field reserved for system use as required. Not designed for holding customer data.
shipment.commented	*commented*	boolean	True if at least one comment has been recorded against this Shipment.
shipment.deleted	*deleted*	boolean	Used to mark this Shipment as deleted.

The following fields are the **manifest** fields for the shipment.

Manifest Data

Groovy Expression	Database Column	Type	Description
shipment.manifestState	*manifestState*	string	Used to hold manifest state of the shipment. Note that a null manifest means that the shipment has not been manifested. If the shipment courier requires a manifest the shipment cannot be marked as despatched until the manifest state has been set.
shipment.addedToDespatchManifest	*addedToDespatchManifest*	string	Optionally-populated date to hold when shipment was added to the despatch manifest.

The following fields are the **priority** fields for the shipment.

Priority Data

Groovy Expression	Database Column	Type	Description
shipment.priority	*priority*	integer	The priority of this shipment.
shipment.priorityName	*priorityName*	string	The name or title which goes with the priority.
shipment.priorityTitle	*priorityTitle*	string	Returns the formatted title for the courier preferably using *priorityName* but falling back to *priority* if necessary.
shipment.priorityValue	*priorityValue*	integer	Returns the priority value which will either be the wired in priority value *priority* or the default value of 1.

The following fields are the **delivery instruction** fields for the shipment.

Delivery Instruction Data

Groovy Expression	Database Column	Type	Description
shipment.deliveryInstruction	*deliveryInstruction*	string	???.
shipment.requestedDeliveryDate	*requestedDeliveryDate*	datetime	The date on which the customer has requested or opted for delivery of the shipment.
shipment.requestedDeliveryTimeSlot	*requestedDeliveryTimeSlot*	string	The requested time slot for the delivery. The granularity depends on the system configuration.
shipment.earliestShipDate	*earliestShipDate*	datetime	???.
shipment.despatchComment	*despatchComment*	string	???.
shipment.deliveryType	*deliveryType*	string	The delivery type for the shipment.
shipment.collectionPoint	*collectionPoint*	string	The identity of the collection point for the shipment. For example for collect from store shipments - indicates the store from which the shipment should be collected.
shipment.collectionPointName	*collectionPointName*	string	The name of the collection point name. For example would be the human readable name of the store.
shipment.deliverySuggestion.code	*deliverySuggestionCode*	integer	The code used for the delivery suggestion. Used by scripts.
shipment.deliverySuggestion.name	*deliverySuggestionName*	string	The name used for thd delivery suggestion. Provides human-readable representation of name.

The following fields are the **courier and carriage** fields for the shipment.

Courier and Carriage Data

Groovy Expression	Database Column	Type	Description
shipment.courierValidated	*courierValidated*	boolean	Whether this shipment has been passed through courier validation. Note that if the shipment changes it is up to the application to set this.
shipment.courierAccepted	*courierAccepted*	boolean	Whether this shipment has been passed through the courier prepare step. Note that if the shipment changes it is up to the application to set this. Typically if a shipment has been accepted it will need to be cancelled with the courier if changes are to be made to the shipment courier options.
shipment.courierErrors	*courierErrors*	boolean	Flag to indicate the presence of courier errors (that is entries in the *ShipmentCourierError* table for this shipment.
shipment.courierState	*courierState*	string	The courier workflow state for the shipment
shipment.deliveryMethod.carrierCode	*carrierCode*	string	The code of the ultimate carrier of the shipment i.e. the company doing the physical transportation of the goods. This may be different to or the same as the carrier name or it may be null.
shipment.deliveryMethod.carrierName	*carrierName*	string	The name of the ultimate carrier of the shipment i.e. the company doing the physical transportation of the goods. For example for the couriers royalmail_ppi and royalmail_tracked in both cases the carrierName is 'Royal Mail'. However there are two different courier implementations.

Courier and Carriage Data (cont.)

Groovy Expression	Database Column	Type	Description
shipment.deliveryMethod.serviceCode	*serviceCode*	string	The selected courier service.
shipment.deliveryMethod.serviceName	*serviceName*	string	The name of the courier service.
shipment.deliveryMethod.options	*courierOptions*	string	A comma separated list of options.
shipment.deliveryMethod.mailFormat	*mailFormat*	string	the mail format assigned to this shipment.
shipment.deliveryMethod.lineHaulSiteReference	n/a	string	Optional reference of a line-haul destination site.
shipment.presortValue	*presortValue*	string	The postal presort number which if set is used as a first stage of sorting prior to collection by the courier.
shipment.despatchReference	*despatchReference*	string	???.
shipment.courierIntermediateReference	*courierIntermediateReference*	string	The courier reference (in addition to the *despatchReference*) for the shipment. Primarily used for courier aggregator services.

The following fields are the **price** fields for the shipment.

Price Data

Groovy Expression	Database Column	Type	Description
shipment.actualShippingPrice.net	*actualShippingPriceNet*	double	"The net price value, before tax."
shipment.actualShippingPrice.gross	*actualShippingPriceGross*	double	The gross price of the item.
shipment.actualShippingPrice.tax	*actualShippingTax*	double	The tax payable. Net price is derived as gross price less tax.
shipment.actualShippingPrice.taxCode	*actualShippingTaxCode*	double	The tax code for the tax payment.

The following fields are the **address** fields for the shipment.

Address Data

Groovy Expression	Database Column	Type	Description
shipment.address.line1	*addressLine1*	string	First line of the address.
shipment.address.line2	*addressLine2*	string	Second line of the address.
shipment.address.line3	*addressLine3*	string	Third line of the address.
shipment.address.line4	*addressLine4*	string	Fourth line of the address.
shipment.address.line5	*addressLine5*	string	Fifth line of the address.
shipment.address.line6	*addressLine6*	string	Sixth line of the address.
shipment.address.countryCode	*countryCode*	string	"The ISO two character country code, for example, 'GB', 'DE'."
shipment.address.postCode	*postCode*	string	Postal code.

The following fields are the **contact** fields for the order.

Contact Data

Groovy Expression	Database Column	Type	Description
shipment.contact.contactName	*contactName*	string	Abbreviated contact name used as an alternative to the salutation/firstName/ lastName combination.
shipment.contact.salutation	*salutation*	string	Contact salutation.
shipment.contact.firstName	*firstName*	string	Contact first name.
shipment.contact.lastName	*lastName*	string	Contact last name.
shipment.contact.emailAddress	*emailAddress*	string	Contact email address.
shipment.contact.dayPhoneNumber	*dayPhoneNumber*	string	Contact day phone number.
shipment.contact.eveningPhoneNumber	*eveningPhoneNumber*	string	Contact evening phone number.
shipment.contact.mobilePhoneNumber	*mobilePhoneNumber*	string	Contact mobile phone number.
shipment.contact.faxNumber	*faxNumber*	string	Contact fax number.
shipment.contact.companyName	*companyName*	string	Contact company name.

The following fields are the **date** fields for the shipment.

Date Data

Groovy Expression	Database Column	Type	Description
shipment.created	*created*	string	The date/time that the shipment was created on the system.
shipment.lastUpdated	*lastUpdated*	string	The date/time that the shipment was last updated.
shipment.completed	*completed*	string	The date/time that the shipment was completed.

The following fields are the **user defined** fields for the shipment. Note that the usage of individual user defined fields is not predefined, and depends entirely on the environment.

User Defined Data

Groovy Expression	Database Column	Type	Description
shipment.userDefined.userDefined1	*userDefined1*	string	
shipment.userDefined.userDefined2	*userDefined2*	string	
shipment.userDefined.userDefined3	*userDefined3*	string	
shipment.userDefined.userDefined4	*userDefined4*	string	
shipment.userDefined.userDefined5	*userDefined5*	string	

The following fields are the **implicit** fields for the shipment. Implicit fields are fields for which the underlying data may differ, and typically involve default or fallback sources of the data if the primary data item is not present.

Implicit Data

Groovy Expression	Database Column	Type	Description
shipment.implicitWeightUnits	*n/a*	string	If weight units are set then uses this. Otherwise uses grams.
shipment.implicitCarrierCode	*n/a*	string	Returns the carrier code. Otherwise returns the couriers external reference.
shipment.implicitCarrierName	*n/a*	string	Returns the carrier name. Otherwise returns the carrier code. Otherwise returns the courier name.
shipment.implicitServiceName	*n/a*	string	Returns the service name. Otherwise returns the service code.
shipment.implicitPriority	*n/a*	string	Returns the priority name. Otherwise returns the priority value.
shipment.implicitShippingPrice	*n/a*	string	Returns the shipping price if one exists. Otherwise it returns a newly instantiated shipping price.
shipment.implicitAddress	*n/a*	Address	Returns shipment address if this is set - otherwise returns order's *DeliveryAddress*.

Implicit Data (cont.)

Groovy Expression	Database Column	Type	Description
shipment.implicitContact	*n/a*	Contact	Returns shipment contact if this is set - otherwise returns order's *DeliveryContact*.
shipment.implicitPhoneNumber	*n/a*	string	Returns implicit phone number. Useful for case where single phone number is required. Uses mobile phone number if available. Otherwise defaults to day phone number then evening phone number.
shipment.formattedImplicitCompanyAndAddress	*n/a*	string	Returns company name and address with '\n' delimiter string.
shipment.formattedImplicitCompanyAndAddressNoCountry	*n/a*	string	Returns company name and address with '\n' delimiter string and no country.

Shipment Attribute

A particular shipment attribute can be obtained from a shipment using the following Groovy.

```
def shipmentAttribute = shipment.getAttribute('shipment_attribute_1');
```

With the reference to the attribute, the `name`, `title` and `value` of the attribute can be obtained using the following code:

```
def shipmentAttribute = shipment.getAttribute('attribute_a');

println 'Attribute Name: ' + shipmentAttribute.name
println 'Attribute Title: ' + shipmentAttribute.title
println 'Attribute Value: ' + shipmentAttribute.value
```

For example, to obtain the value of a known attribute, you can use the following expression:

```
    println shipment.getAttribute('custom_reference')?.value;
```

Order Line

The following fields are the **top level** fields for the order line.

Order Line Data

Groovy Expression	Database Column	Type	Description
orderLine.quantity	*quantity*	integer	Quantity of a particular line.
orderLine.thirdPartyReference	*thirdPartyReference*	string	An optional reference supplied by a 3rd party system that may be played back to it.
orderLine.description	*description*	string	The description of the order line. If available uses the declared description (may be language-specific). Otherwise uses the description of the product.
orderLine.state	*state*	string	The current state of the Order Line.
orderLine.virtual	*virtual*	boolean	True if this is an order line which is inactive from a stock management point of view. The *virtual* flag will typically be set at the point at which an order line is identified as not having any concrete stock requirement. An order line which is associated with a virtual product is marked as virtual.
orderLine.packaging	*packaging*	boolean	True if this is an order line which is for a packaging product. If set the line may be excluded from certain workflow processes e.g. excluded from despatch notes. An order line which is associated with a packaging product is marked as packaging.
orderLine.productBatch	*productBatch*	string	Used to capture the product batch number for batch tracked products.

The following fields are the **system** fields for the order line.

System Data

Groovy Expression	Database Column	Type	Description
orderLine.version	*version*	integer	The current version of a specific row. This is used to prevent concurrent changes.
orderLine.deleted	*deleted*	boolean	Used to mark this Order Line as deleted.
orderLine.complete	*complete*	boolean	Flag which indicates that the order line has been complete. Flag is used to indicate that the order line is no longer outstanding. A cancelled order line is also marked as complete.
orderLine.progressIndication	*progressIndication*	integer	Holds the progress indicator for an order line.
orderLine.mergedFromOrderLines	*mergedFromOrderLines*	Collection	A collection of order line identifiers from which this order line was merged. Null / empty if this order line was never merged from other lines.
orderLine.inactive	*inactive*	boolean	True if the order line is inactive so should not be considered for any operations but should still be present for visibility. An order line which is cancelled becomes inactive at the point at which it is cancelled.

The following fields are the **price** fields for the order line.

Price Data

Groovy Expression	Database Column	Type	Description
orderLine.totalPrice.net	*totalPriceNet*	double	"The net price value, before tax."
orderLine.totalPrice.gross	*totalPriceGross*	double	The gross price of the item.
orderLine.totalPrice.tax	*totalTax*	double	The tax payable. Net price is derived as gross price less tax.
orderLine.totalPrice.taxCode	*totalTaxCode*	double	The tax code for the tax payment.
orderLine.unitPrice.net	*unitPriceNet*	double	"The net price value, before tax."
orderLine.unitPrice.gross	*unitPriceGross*	double	The gross price of the item.
orderLine.unitPrice.tax	*unitTax*	double	The tax payable. Net price is derived as gross price less tax.
orderLine.unitPrice.taxCode	*unitTaxCode*	double	The tax code for the tax payment.
orderLine.promotionCode	*promotionCode*	string	Holds product level promotion code used when order line was taken.
orderLine.promotionPriceDescription	*promotionPriceDescription*	string	Holds promotion price description field which can be used directly on reports (e.g '£16.99 (was £20.99)').

The following fields are the **user defined** fields for the order line. Note that the usage of individual user defined fields is not predefined, and depends entirely on the environment.

User Defined Data

Groovy Expression	Database Column	Type
orderLine.userDefined.userDefined1	*userDefined1*	string
orderLine.userDefined.userDefined2	*userDefined2*	string
orderLine.userDefined.userDefined3	*userDefined3*	string
orderLine.userDefined.userDefined4	*userDefined4*	string
orderLine.userDefined.userDefined5	*userDefined5*	string

The following fields are the **date** fields for the order line.

Data Data

Groovy Expression	Database Column	Type	Description
orderLine.created	*created*	string	The date/time that the order line was created on the OrderFlow system.

The following fields are the **implicit** fields for the order line. Implicit fields are fields for which the underlying data may differ, and typically involve default or fallback sources of the data if the primary data item is not present.

Implicit Data

Groovy Expression	Database Column	Type	Description
orderLine.implicitTotalPrice	*n/a*	price	Returns the implicit total price calculated if necessary from the unit price multiplied by the quantity. Note that all net gross and tax fields of the price will be populated.
orderLine.implicitUnitPrice	*n/a*	price	Returns implicit unit price for item. If declared unit price is present then uses this. Does not use product price.
orderLine.implicitDescription	*n/a*	string	Returns implicit description for order line. If declared description is present then uses this. Otherwise uses product description.
orderLine.orderLineAndProductDescription	*n/a*	string	Returns where possible both the Order Line and Product descriptions combined.
orderLine.implicitWeightFromProduct	*n/a*	double	Returns the implicit weight from the individual products. Returns null if any of the constituent products does not have a weight.
orderLine.implicitWeight	*n/a*	double	Returns the implicit weight for this order line based on the product weight.

Order line Attribute

A particular order line attribute can be obtained from an order line using the following Groovy.

```
def orderLineAttribute = orderLine.getAttribute('order_line_attribute_1');
```

With the reference to the attribute, the `name`, `title` and `value` of the attribute can be obtained using the following code:

```
def orderLineAttribute = orderLine.getAttribute('attribute_a');

println 'Attribute Name: ' + orderLineAttribute.name
println 'Attribute Title: ' + orderLineAttribute.title
println 'Attribute Value: ' + orderLineAttribute.value
```

For example, to obtain the value of a known attribute, you can use the following expression:

```
println orderLine.getAttribute('line_identifier')?.value;
```

Product

The following fields are the **top level** fields for the product.

Product Data

Groovy Expression	Database Column	Type	Description
product.externalReference	*externalReference*	string	Unique and Human-readable reference for the Order.
product.thirdPartyReference	*thirdPartyReference*	string	An optional reference supplied by a 3rd party system that may be played back to it.
product.state	*state*	string	The current state of the Product.
product.description	*description*	string	The description of the product.
product.warehouseDescription	*warehouseDescription*	string	The description of the product in the warehouse.
product.barcode	*barcode*	string	Barcode for product typically provided by supplier. Can be used as a simpler alternative to specifying supplier product codes. Useful if all products are coming from a single source and only a single alternative barcode is needed.
product.imageReference	*imageReference*	string	A reference which can be used to display the image on an external system.
product.customsDescription	*customsDescription*	string	The customs description. If not set can be inferred from product category.
product.productComposition	*productComposition*	string	The product composition.
product.countryOfOrigin	*countryOfOrigin*	string	The country of origin. Typically used in customs declarations. The normal expectation is that this field will be populated using the letter ISO country code for the product although it will also support full country name if required.
product.weight	*weight*	double	The weight for a single unit of the specified product. Only applies for countable products that is products that don't have a decimal quantity.
product.weightUnits	*weightUnits*	string	The weight units to be applied for this product if different from the default.

Product Data (cont.)

Groovy Expression	Database Column	Type	Description
product.quantityType	*quantityType*	string	Quantity type. Applies only for products that support decimal quantities.
product.displayUnits	*displayUnits*	string	The units to be used for display.
product.packagingDescription	*packagingDescription*	string	Text describing how item is packaged and quantity thereof. e.g boxes of 6 cartons of 12
product.physicalStorageTypes	*physicalStorageTypes*	string	If set then the location in which the product is stored must have a physical location type value corresponding with one of the storage physical types specified.
product.sellable	*sellable*	boolean	Is considered sellable if it is eligible to appear as a sellable product on one or more sales platforms.
product.dangerous	*dangerous*	boolean	"If true product is dangerous
product.fragile	*fragile*	boolean	If true product is fragile.
product.activated	*activated*	boolean	If true product is active.

The following fields are the **system** fields for the product.

System Data

Groovy Expression	Database Column	Type	Description
product.version	*version*	integer	The current version of a specific row. This is used to prevent concurrent changes.
product.deleted	*deleted*	boolean	Used to mark this Order Line as deleted.
product.hasMultipleBarcodes	*hasMultipleBarcodes*	boolean	If true this product has more than one barcode. In addition to the "primary" *barcode* field it has other "secondary" barcodes held in *Barcode* entities that reference this product.
product.hasDatasheet	*hasDatasheet*	boolean	If true this product has an associated *ProductDataSheet*.
product.inGroup	*inGroup*	boolean	Is true if product has associated grouped product records. Needs to be set against the product at the point at which it is identified that the product is associated with *GroupedProduct* entries.

The following fields are the **dimensions** fields for the product.

Dimensions Data

Groovy Expression	Database Column	Type	Description
product.length	*length*	double	The length of the longest horizontal dimension for movable items (e.g. products). For locations the length is actually the depth of the location (the horizontal distance from the access face to the 'back' of the location).
product.width	*width*	double	The length of the shortest horizontal dimension for movable items (e.g. products). For locations the length is actually the depth of the location (the horizontal distance from the access face to the 'height' of the location).
product.height	*height*	double	The height of the vertical dimension.
product.area	*area*	double	The area of the horizontal dimension. Can be explicitly specified but otherwise is determined by the product of length and width.
product.volume	*volume*	double	The volume. Can be explicitly specified but otherwise is determined by the product of length width and height.

The following fields are the **price** fields for the product.

Price Data

Groovy Expression	Database Column	Type	Description
product.listPrice.currency	*currency*	double	The currency of the sale price.
product.listPrice.currencyUnits	*currencyUnits*	double	The units of currency of the sale price.
product.listPrice.net	*priceNet*	double	"The net price value, before tax."
product.listPrice.gross	*priceGross*	double	The gross price of the item.
product.listPrice.tax	*tax*	double	The tax payable. Net price is derived as gross price less tax.
product.listPrice.taxCode	*taxCode*	double	The tax code for the tax payment.

The following fields are the **user defined** fields for the product. Note that the usage of individual user defined fields is not predefined, and depends entirely on the environment.

User Defined Data

Groovy Expression	Database Column	Type
product.userDefined.userDefined1	*userDefined1*	string
product.userDefined.userDefined2	*userDefined2*	string
product.userDefined.userDefined3	*userDefined3*	string
product.userDefined.userDefined4	*userDefined4*	string
product.userDefined.userDefined5	*userDefined5*	string

The following fields are the **implicit** fields for the product. Implicit fields are fields for which the underlying data may differ, and typically involve default or fallback sources of the data if the primary data item is not present.

Implicit Data

Groovy Expression	Database Column	Type	Description
product.implicitHarmonizedSystemCode	*n/a*	string	Returns harmonized system (HC) code from product if set otherwise uses the value for category if present.
product.implicitCustomsDescription	*n/a*	string	Returns customs description from product if set otherwise uses the value for category if present.
product.implicitDimensions	*n/a*	ProductDimensions	Returns dimensions for the location if possible. Prefers the dimensions instance currently attached to the location. If not present attempts to find one attached to the *storageClass* field if present.

Import Mapping

Import Mapping Pre-translation

Background

Imports to OrderFlow are received using a canonical 'flattened' format of property data such as the data shown below:

```
order.externalReference=TEST_minorder
order.channel=MYCHANNEL
order.deliveryAddressLine1=4 Quayside Place
order.deliveryContactName=Phil Zoio
order.deliveryEmailAddress=phil@orderflow-wms.co.uk

shipment.state=ready
shipment.method=metapack

orderLine.1.product.externalReference=DVD-BELOVED
orderLine.1.quantity=10

orderLine.2.product.externalReference=DVD-MATR
orderLine.2.quantity=20
```

The data received at this point can be manipulated as data prior to being mapped to the OrderFlow entity instances (orders, shipments, lines, etc.). For this reason, the mapping is described as the *pre-translation* mapping.

Pre-translation mappings map imported data field values to entity attributes, with the additional capability of writing to any other entity attributes or context values.

Some examples of what you may wish to do with pre-translation mappings:

- map an incoming value to a different field name
- change the string format of an incoming value
- map one set of predefined values to another set of predefined values
- introduce new field values, or even entire entity instances (e.g. order, shipment and product attributes).

An example

A complete example of an import mapping pre-transation is shown below:

```
<fieldmapper>
    <mappings useinput="true">
        <mapping qualifier="order" to="state">'validated'</mapping>
        <mapping qualifier="order" to ="deliveryPostCode">
        return value?.toUpperCase();
        </mapping>
        <mapping qualifier="order" from="paymentDetail" to ="paymentTransactionInfo"/>
        <mapping qualifier="shipment" to="method">
        <![CDATA[
        output['deliverySuggestionCode']=value;
        if (value == 'premium_method') {
        output['priorityName']='urgent';
        output['priority']=100;
        } else {
        output['priorityName']='normal';
        output['priority']=1;
```

```
        }
        ]]>
        </mapping>
        <mapping qualifier="order" remove="paymentDetail"/>
    </mappings>
</fieldmapper>
```

A few notes about the input mapping document follow below:

From and To Atributes

The *to* attribute of the *mapping* element defines the field in the target entity instance to which the data will be written. The value used can either be the literal value received from the incoming data, or the value returned from a script contained in the mapping element.

In the example above, the value `validated` is written to the order *state* field.

(Note that `'validated'` is actually a Groovy script which simply returns the literal string value `validated`).

The *from* attribute is not mandatory. If present, it will be used as the source field name for the input data.

For example, in the example

```
<mapping qualifier="order" from="paymentDetail" to ="paymentTransactionInfo"/>
```

the input value in the field *paymentDetail* will be mapped to the output field *paymentTransactionInfo*.

Note that if no *from* attribute is present, then the source or input field name will be the same as the output field name.

Use input attribute

The top-level mappings element has a *useinput* attribute, which can be set to `true` or `false`. If true, then an attempt will be made to write every import data attribute (that is not explicitly removed) to its target entity.

Depending on the configuration, any incorrectly-named data will raise an error in OrderFlow, but at the very least will log a warning.

If *useinput* is set false, the mapping operation will only set field values for fields for which there is an explicit mapping in the field document. This will result in no redundant fields, but will tend to require more configuration to set up the full set of mappings.

Remove Attribute

The *remove* attribute is only required when *useinput* is true; it is used to prevent input values from being mapped to output fields for which no equivalent OrderFlow field value exists.

In our example above, the *paymentDetail* field value is removed, as it is not a field that corresponds with the OrderFlow data model.

```
<mapping qualifier="order" remove="paymentDetail"/>
```

Qualifiers

Each contained mapping element must have a qualifier attribute, which (for order imports) can take the values order, shipment or orderLine. This defines which entity the mapped data applies to.

There is an important **restriction** that needs to be understood here. During the mapping process, each qualifier effectively gets its own set of input values. As a result, it is not possible to map directly, for example, from input values received using the qualifier `shipment` to output values with the qualifier `order`, and vice versa.

There is a technique for getting around this where necessary, to be discussed below.

Scripting Context

The contents of the mapping element can be a literal value (enclosed in single- or double-quotes), or a Groovy script. The script has the following values available to it, as described below:

value

This is automatically populated from the imported data referenced by the *from* attribute, if present, otherwise using the *to* attribute.

Note that all of the examples below are equivalent, all of which map to the output map with the field name of state.

```
<mapping qualifier="order" from="state" to="state"/>
```

The script below relies on the fact that the *from* field and the *to* field use refer to the same attribute.

```
<mapping qualifier="order" to="state"/>
```

The example below relies on the fact that the variable *value* refers to the value for the *to* attribute.

```
<mapping qualifier="order" to="state">
return value;
</mapping>
```

input

This is a map that contains all the supplied properties from the import data (which can be read from directly).

The *input* variable can be used to reference any value from the input map belonging to the same qualifier.

For example, another way of achieving the same result as the previous three mappings is shown below.

```
<mapping qualifier="order" to="state">
return input['state'];
</mapping>
```

In practice, *input* map is used when there is the need to reference multiple input values to obtain a mapped value.

For example,

```
<mapping qualifier="shipment" to="priority">
if (input['urgent'] == 'true' && input['shipping_option'] == 'expedited') {
    return '10';
}
return '1';
</mapping>
```

In the example above, the input fields *urgent* and *shipping_option* are together used to determine the shipment priority.

output

This is a map that contains all the resulting mapped properties (which can be written to explicitly).

The main use for the *output* map is to allow multiple field values to written to using as single mapping entry.

For example, in a modified version of the previous example:

```
<mapping qualifier="shipment" to="priority">
if (input['urgent'] == 'true' && input['shipping_option'] == 'expedited') {
    output['priorityName'] = 'Urgent';
    return '10';
}
output['priorityName'] = 'Normal';
return '1';
</mapping>
```

The priority name of 'Urgent' or 'Normal' can be set as a side effect of the previous mapping script. This avoids the need for the mapping document to contain repeated mapping entries with very similar logic.

context

This is a map that is available to write to and read from throughout the pre-translation processing.

The context can be used to allow data to be transferred across mapping inputs and outputs with different qualifiers.

Let's consider the import of an example of an order which has a single shipment and two order lines.

```
order.externalReference=TEST_minorder
order.channel=MYCHANNEL
order.deliveryAddressLine1=4 Quayside Place
order.deliveryContactName=Phil Zoio
order.deliveryEmailAddress=phil@orderflow-wms.co.uk
delivery_method=next_day

shipment.state=ready

orderLine.1.product.externalReference=DVD-BELOVED
orderLine.1.quantity=10

orderLine.2.product.externalReference=DVD-MATR
orderLine.2.quantity=20
```

The mapping input will be expanded into the following:

- 1 set of mapping entries with qualifier `order` for the order values.
- 1 set of mapping entries with qualifier `shipment` for the shipment values (beginning with the prefix `shipmet.`.
- 2 sets of mapping entries with qualifier `orderLine`, for the line values (beginning with the prefix `orderLine.1.` and `orderLine.2.`.

In the example above, notice the presence of

```
deliveryMethod=next_day
```

Because this entry has no *qualifier*, the qualifier is implicitly set to the top level entity, `order`. The problem occurs because there is no order field named `delivery_method`, although is a shipment field called `deliverySuggestionName` which would be a suitable target for this value.

However, it is **not possible** to map to this field using a mapping entry such as:

```
<mapping qualifier="order" to="delivery_method">
    output['deliverySuggestionName'] = value;
</mapping>
```

The way around this is to map the delivery method to a context attribute, then to map the context attribute to the target output field.

```
<mapping qualifier="order" to="delivery_method">
    context['deliverySuggestionName'] = value;
</mapping>
<mapping qualifier="shipment" to="deliverySuggestionName">
    return context['deliverySuggestionName'];
</mapping>
<mapping qualifier="order" remove="delivery_method"/>
```

For extra measure, we can remove the `delivery_method` mapping using the mapping with the *remove* attribute.

entries

The *entries* context variable gives us access to the complete set of input values that have been received across the entire input document.

We mentioned earlier that for the input document below that mapping input will support the creation of one order, one shipment and two order lines.

```
order.externalReference=TEST_minorder
...

shipment.state=ready
...

orderLine.1.product.externalReference=DVD-BELOVED
orderLine.1.quantity=10
...

orderLine.2.product.externalReference=DVD-MATR
orderLine.2.quantity=20
...
```

However, suppose we also receive additional values that we want to associate with the order, but we don't want to map to any existing order fields.

For this purpose, we need to add **attributes**.

The use of attributes tends to be domain specific. In the example of animal medicines, additional attributes might be:

- `animal_type` (dog, cat, etc.)
- `animal_age` (12 months)
- `animal_name` (e.g. Rover)

The attributes could be received natively (without modification) using the following input file:

```
order.externalReference=TEST_minorder
orderAttribute.1.name=animal_type
orderAttribute.1.value=dog
orderAttribute.2.name=animal_age
orderAttribute.2.value=12 months
orderAttribute.3.name=animal_name
orderAttribute.3.value=Rover
```

However, suppose the attributes are received in the following way, which is more economical for the producer of the document:

```
order.externalReference=TEST_minorder
animal_type=dog
animal_age=12 months
animal_name=Rover
```

The **entries** variable can be used in a script as below:

```
<mapping qualifier="order" to="animal_type">
    entries.addEntry('orderAttribute',1,'name','animal_type');
    entries.addEntry('orderAttribute',1,'value',value);
    entries.addEntry('orderAttribute',1,'orderItem','entity:order');
</mapping>
<mapping qualifier="order" to="animal_age">
    entries.addEntry('orderAttribute',2,'name','animal_age');
    entries.addEntry('orderAttribute',2,'value',value);
    entries.addEntry('orderAttribute',2,'orderItem','entity:order');
</mapping>
<mapping qualifier="order" to="animal_name">
    entries.addEntry('orderAttribute',3,'name','animal_name');
    entries.addEntry('orderAttribute',3,'value',value);
    entries.addEntry('orderAttribute',3,'orderItem','entity:order');
</mapping>
```

The `entries.addEntry('orderAttribute',1,'name','animal_type')` creates a mapping entry for the first input map for the `orderAttribute` qualifier, with the value `animal_name` for the attribute name.

By adding entries as above, additional entity instances can be populated using scripts.

Import Mapping Post-translation

Order import mappings allow imported data to be manipulated, irrespective of *how* they have been imported.
This manipulation is broadly split into what is known as *pre*-translation mappings and *post*-translation transformations.

This document details how to apply order import *post*-translation mappings.
Post-translation transformations apply changes to the imported entities *after instantiation* (but before persistence), so can also write to any related entity attributes, or context values.

An example of a post-transformation mapping is shown below:

```
<transformations>
    <transformation qualifier="order">
        <![CDATA[
        if (input.deliveryContact.mobilePhoneNumber == null) {
        input.deliveryContact.mobilePhoneNumber = input.deliveryContact.dayPhoneNumber;
        }
        ]]>
    </transformation>
</transformations>
```

The top-level *transformations* element contains *transformation* elements, each of which must have a *qualifier* attribute.
For order imports this can take the values `order`, `shipment` or `orderLine` (or indeed `orderAttribute`, `shipmentAttribute` or `orderLineAttribute`). This defines the type of entity to which the transformation will be applied.

Note that the transformation will be applied to *all* entities of that type created during the import process, so if there is more than one `orderLine` entity created, a transformation with the qualifier `orderLine` will be applied to each of these order lines.

There are two main strategies for applying individual translations:

- **scripted** transformations, which involves the running of a Groovy script defined within the *transformation* element.
- **built-in** transformations, involving the running of some predefined functionality.

Built-in transformations are discussed later in this document. The example above contains a single scripted transformation mapping, which will be discussed next.

Scripted Post-translators

The contents of the `transformation` element can be a literal value (enclosed in single- or double-quotes), or a Groovy script.

Unlike pre-transformations, the *return value* for post transformations is not important, as the scripts operate directly on OrderFlow entity instances after they have been fully populated using existing and received incoming data, but before these updated/created entities have been persisted. In other words, there is no need for `return` statement in a post-translation script; any value returned from the script will simply be ignored.

The following variables are available in the **scripting context** for an import post-transformation.

input

This is an instance of the entity referenced by the 'qualifier' attribute.

It is worth noting that the a post-translation script has much broader and direct access to the OrderFlow data model than the pre-translation mapping script.

While the pre-translation only has access to the data that feeds the import operation, it does not have access to the OrderFlow entity instances.

By contrast, the post-translation script can access OrderFlow orders, shipments, order lines and other entities directly using the OrderFlow Scripting API. This is evident in the example script below.

```
<transformation qualifier="order">
    <![CDATA[
    if (input.deliveryContact.mobilePhoneNumber == null) {
        input.deliveryContact.mobilePhoneNumber = input.deliveryContact.dayPhoneNumber;
    }
    ]]>
</transformation>
```

Note how the *input* variable is used to access the order entity instance directly, and to manipulate its field values.

context

Contains a map to which contextual data can be written. For example, this provides a mechanism by which data can be shared between different post-translation mappings.

Note that the context can be populated *declaratively* using the *context* and *parameter* elements, as shown in the example below:

```
<transformations>
    <context>
    <parameter name="postalSortFileProperty">royalmail.postal.sort.file</parameter>
    </context>
    <transformation .../>
</transformations>
```

context.importStateHolder

One of the values that will automatically be available in the **context** is stored under the key `importStateHolder`. This is an instance of the Java class `rtd.imports.spi.ImportStateHolder`, and contains the context of the entire import operation.

Use of this is pretty rare, but it does allow for contextual data that is used in the import process but not the OrderFlow domain model to be accessed (and, if appropriate, modified).

```
<transformation qualifier="productAttribute">
<![CDATA[
if (input.isPersisted()) {
    def context = context.importStateHolder.context;
    context.get('suppressPersistence').add(input);
}
]]>
</transformation>
```

Built-in Import Handlers

In addition to scripted post-translation handers, there are also **built-in** transformation handlers which can be used to manipulate an incoming order, shipment or OrderLIne.

```
<transformations>
<transformation qualifier="shipment" handler="weightCalculator">
</transformation>
</transformations>
```

As with the scripted transformation, the built-in transformation uses a *qualifier* to determine the scope of the transformation.

Among the main built-in post translators include:

Name	Qualifier	Usage
postalSorter	`order`	Applies a postal sort value to a *shipment*.
courierSetterTransformer	`shipment`	Sets the courier and/or service for a shipment.
batchSetterTransformer	`shipment`	Sets the batch type of a *shipment* after invoking the batch selection script.
batchTypeTransformer	`shipment`	Sets the batch type of a *shipment* from a context property value.
weightCalculator	`shipment`	Calculates the shipment weight from the weight of its constituent products.
goodsPriceCalculator	`shipment`	Calculates the shipment goods price from the order line or product data.
countryCodeSetter	`shipment`	Validates a supplied two character country code

Note that the transformation needs to be set correctly for the built-in tranformation implementation used.

A bit more detail on each of the above is provided next.

postalSorter

The postal sorter allows for the *shipment.presortValue* to be set during an order import. For some couriers, doing a presort at the warehouse can result in a cheaper rate for shipments.

The *postalSorter* transformation uses a spreasheet file which contains a mapping of post code prefixes to sort values, as in the example below

```
sortcode postcode
1  AB
2  AL
3  B
4  BA
5  BB
6  BD
7  BH
8  BL
```

```
9  BN
10 BR
11 BS
...
```

If the *postalSorter* built-in transformation is to be used, it needs to be accompanied by the name of the property which stores the postal sort file, this can be done as in the example below:

```
<transformations>
    <context>
    <parameter name="postalSortFileProperty">royalmail.postal.sort.file</parameter>
    </context>
    <transformation qualifier="order" handler="postalSorter">
    </transformation>
</transformations>
```

courierSetterTransformer

The *courierSetterTransformer* allows the selection of the courier to be determined at import item using business rules defined in the Courier Selection Script.

The declaration of the *courierSetterTransformer* transformer is simple, as shown below.

```
   <transformations>
      <transformation qualifier="shipment" handler="courierSetterTransformer">
      </transformation>
   </transformations>
```

Details on the Courier Selection Script are provided later in this document.

batchSetterTransformer

The *batchSetterTransformer* allows the selection of the batch types of shipments that are to undergo batch picking to be determined at import item using business rules defined in the Batch Selection Script.

As with the *courierSetterTransformer*, the declaration of the *batchSetterTransformer* transformer is simple, as shown below.

```
<transformations>
    <transformation qualifier="shipment" handler="batchSetterTransformer">
    </transformation>
</transformations>
```

Details on the Batch Selection Script are provided later in this document.

One word of recommendation is that if both *courierSetterTransformer* and the *batchSetterTransformer* are both present in the same mapping document, it is advisable to put the courier transformation first, as this allows the batch selection script to use an already populated courier selection, if required.

batchTypeTransformer

The *batchTypeTransformer* performs a similar function to the *batchSetterTransformer*, except that it uses the output of a script within the *transformation* element to determine the batch type, rather than an exteranlly defined Batch Selection Script. An example is shown

```
<transformation qualifier="shipment" handler="batchTypeTransformer" scriptFirst="true">
    <![CDATA[
    if (input.orderLineCount > 1) {
    context.batchType = 'multiline'
```

```
    } else {
    context.batchType = 'singleline'
    }
    ]]>
</transformation>
```

Notice how batch type is stored in the *context*. Notice also the use of the *scriptFirst* attribute. This ensures that when both a built-in handler and a script are present within the same transformation, the script is evaluated before the handler functionality is invoked.

The *batchTypeTransformer* is rarely used, as it is less flexible than the Batch Selection Script, which allows the batch type to be set at other points in the order processing workflow, and not just at import time.

weightCalculator

The *weightCalculator* is a useful transformation to apply to ensure that the weight value for shipments is set, even if not received as part of the shipment import data.

The usage of the weight calculator is also very simple, as shown below:

```
<transformations>
    <transformation qualifier="shipment" handler="weightCalculator">
    </transformation>
</transformations>
```

There are a few points to note about the use of the weight calculator as it is currently implemented:

- the *weightCalculator* will only set the weight of shipments if all products for order lines in the shipment have a weight value set. If the weight value is missing, then no weight value will be set for the shipment.
- if the *weightCalculator* is used, then it will override any value that is present in the data received from the eCommerce system.

If a weight value is required for the shipment, then it is possible to reject the import using a subsequent transformation, or even using a script within the same transformation.

goodsPriceCalculator

The goods price calculator, as the name suggests, allows for the goods price value to be calculated from order line or product data.

The declaration of the *goodsPriceCalculator* is shown below:

```
<transformations>
    <transformation qualifier="shipment" handler="goodsPriceCalculator">
    </transformation>
</transformations>
```

A couple of noteworthy points about the behaviour of the goods price calculator:

- if the order goods price is already set in the received order data, then no price calculation will be applied.
- the calculation will attempt to use the order line price values. If no values are set at the order line level, the handler will attempt to set the goods price using the product price values.
- if the price is not available for one of the order lines (or products), then no goods price will be set.

A More Complete Example

A complete example of an import post-translation mapping, which uses all of the elements described above, is shown below.

```
<transformations>
    <context>
    <parameter name="postalSortFileProperty">royalmail.postal.sort.file</parameter>
    </context>
    <transformation qualifier="order">
    <![CDATA[
    if (input.deliveryContact.mobilePhoneNumber == null) {
        input.deliveryContact.mobilePhoneNumber = input.deliveryContact.dayPhoneNumber;
    }
    ]]>
    </transformation>
    <transformation qualifier="order" handler="postalSorter">
    </transformation>
    <transformation qualifier="shipment" handler="courierSetterTransformer">
    </transformation>
    <transformation qualifier="shipment" handler="batchTypeTransformer" scriptFirst="true">
    <![CDATA[
    if (input.orderLineCount > 1) {
    context.batchType = 'multiline'
    } else {
    context.batchType = 'singleline'
    }
    ]]>
    </transformation>
</transformations>
```

Input Handlers

The import mapping pre-translation and post-translations are assumed to work on data which is essentially in the OrderFlow native or canonical format. The intention of these translations is to make minor changes as well as to apply business rules to the imported data.

There are situations where additional transformations are necessary just to bring the data into the OrderFlow canonical format.

Example of where this may be necessary include case where data received in non-standard CSV, XLS or JSON formats, or in completely custom formats.

For handling these kinds of situations, OrderFlow supports an array of input **handlers**. In many cases, there is scripting capability built around these.

CSV and XSLT Input Handlers

Name	Entities	Usage
import_asn_delimited	Advanced Shipping Note	Allows for a custom CSV input format for Advanced Shipping Notes.
import_delivery_delimited	Delivery	Allows for a custom CSV input format for Deliveries.
import_purchaseorder_delimited	Delivery	Allows for a custom CSV input format for Purchase Orders.
import_product_delimited	Product	Allows for a custom CSV input format for Products.
import_order_delimited	Order	Allows for a custom CSV input format for Orders.
import_xslt	Any	Allows for XSLT transformation to be applied to non-standard XML input for Orders.

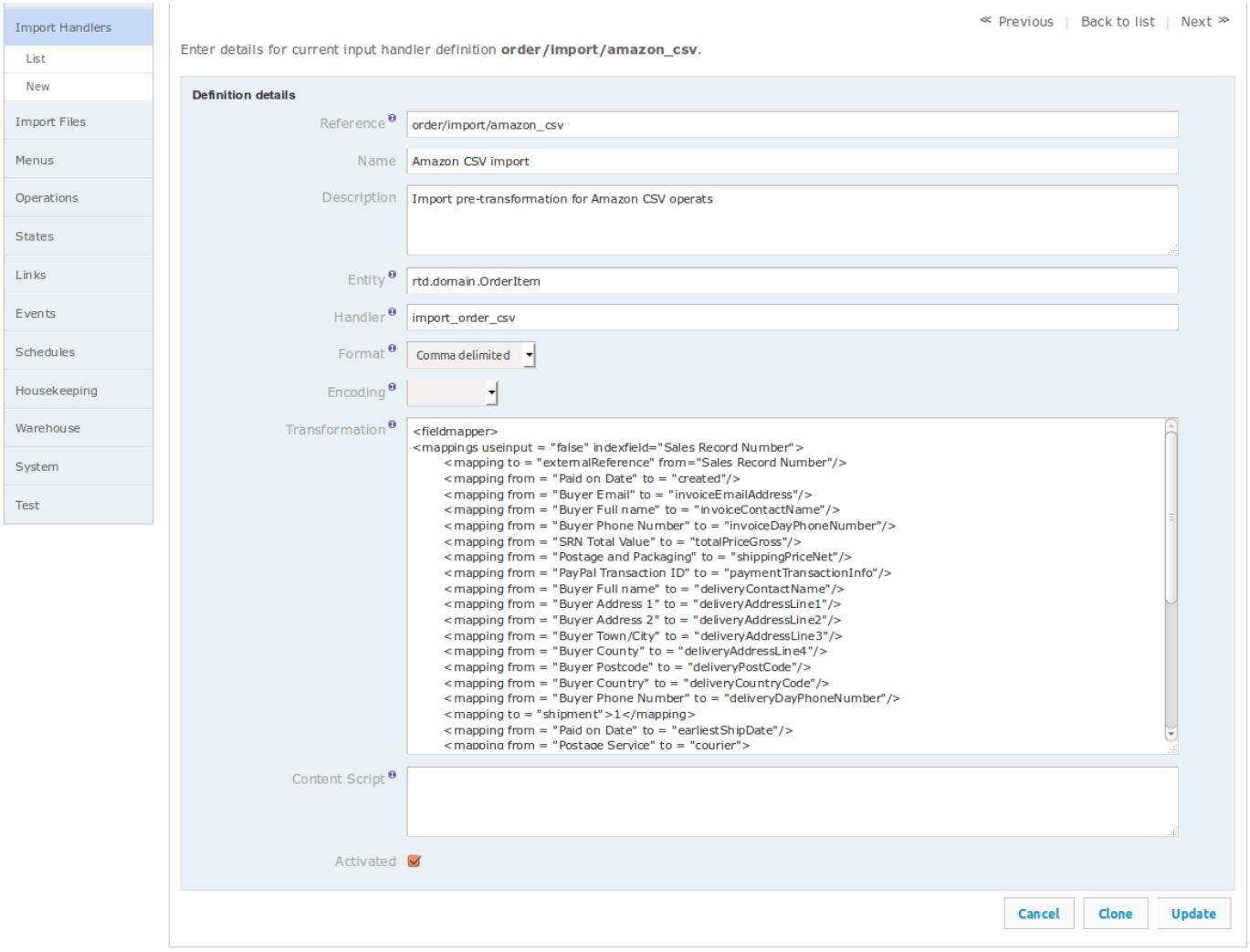
Native XML Import Handlers

Name	Entities	Usage
import_properties_xml	Any	Allows for data to be imported in the native 'properties XML' format
import_structured_xml	Any	Allows for data to be imported in the native full format
import_validated_xml	Any	Similar to 'Structured XML'

Note that for the native XML input handlers, transformations typically aren't necessary, as the data is already in a format that can be read and understood by OrderFlow.

Configuration

The configuration screen for a single input handler is shown below. You can use the 'info' icon on this screen to get more detail on the purpose of the individual fields.



For the purposes of OrderFlow scripting, there are two fields that are important:

- **Handler Transformation**: an optional handler specific transformation mapping that will be applied to the data in the received custom format.
- **Content Script**: a script that can be applied directly to the incoming text.

Input Handler Mapping

The two main types of input handler mapping are CSV and XSLT transformations, which are described respectively below.

CSV Input Handler Mapping

CSV-based documents are text documents which have the following tabular structure:

- the first row is the *header* row which contains the field names for the rows in the document.
- subsequent rows contain the data. Successive values are separated by a delimiter, which is most often a comma, hence the name CSV (Comma-separated values).
- individual data values cannot be multi-line, as it is assumed that each row only uses a single line. There is a way around this, described below.
- if the values themselves contain the delimiter, then the value will need to be enclosed in quotation marks, as in the example shown below.

```
externalReference,"Description"
TEST_123,"A product with a lengthy description, including a comma"
```

While the comma is the most commonly used delimiter, other delimiters can be used, such as the tab and pipe (|) characters.

The transformation used in a CSV input handler transformation looks very similar to imnport mapping pre-translation, as they both use the *fieldmapper* XML structure. There are close similarities, but also some notable differences.

An example CSV input handler transformation is shown below:

```
<fieldmapper>
<mappings useinput = "false" indexfield="Sales Record Number">
    <mapping to = "externalReference" from="Sales Record Number"/>
    <mapping from = "Paid on Date" to = "created"/>
    <mapping from = "Buyer Email" to = "invoiceEmailAddress"/>
    <mapping from = "Buyer Full name" to = "invoiceContactName"/>
    <mapping from = "Buyer Phone Number" to = "invoiceDayPhoneNumber"/>
    <mapping from = "SRN Total Value" to = "totalPriceGross"/>
    <mapping from = "Postage and Packaging" to = "shippingPriceNet"/>
    <mapping from = "PayPal Transaction ID" to = "paymentTransactionInfo"/>
    <mapping from = "Buyer Full name" to = "deliveryContactName"/>
    <mapping from = "Buyer Address 1" to = "deliveryAddressLine1"/>
    <mapping from = "Buyer Address 2" to = "deliveryAddressLine2"/>
    <mapping from = "Buyer Town/City" to = "deliveryAddressLine3"/>
    <mapping from = "Buyer County" to = "deliveryAddressLine4"/>
    <mapping from = "Buyer Postcode" to = "deliveryPostCode"/>
    <mapping from = "Buyer Country" to = "deliveryCountryCode"/>
    <mapping from = "Buyer Phone Number" to = "deliveryDayPhoneNumber"/>
    <mapping to = "shipment">1</mapping>
    <mapping from = "Paid on Date" to = "earliestShipDate"/>
    <mapping to = "orderLine">input["Sales Record Number_index"]</mapping>
    <mapping from = "Item Number" to = "product.externalReference"/>
    <mapping from = "Quantity" to = "quantity"/>
    <mapping from = "item-price" to = "totalPriceNet"/>
    <mapping from = "item-tax" to = "totalTax"/>
</mappings>
</fieldmapper>
```

The key similarity with import pre-translations is at the level of the individual *mapping* element. You can use the *to* and *from* fields for the source and target field names. The mapping elements can contain scripted or literal values. If you are comfortable with import pre-translations, you are well on your way with input handler transformations.

There are a number of notable **differences** or new elements.

The useinput attribute

With import pre-translations, it is generally safe to allow *useinput* to be `true`, as you can normally assume that the import data file has been constructed with a reasonable knowledge of the OrderFlow data structure. With input handler transformations, it's normally best to leave *useinput* to false, as you would normally want to explicitly map all of the fields from the bespoke format to the OrderFlow format.

No qualifier

There is no qualifier present in the input handler transformation; here, we are working with a single set of data items per row of data.

Field ordering

With import pre-translations, the order of the mapped fields generally does not matter too much, as the ordering is largely determined by the *qualifier* attribute.

The ordering of the fields in the input handler transformation is very important, especially with order imports.

In the above example, the first set of fields, starting with `<mapping to = "externalReference" from="Sales Record Number"/>` are mapped to the top level entity, which in the above example is the *order*.

The point at which this changes is at the line below:

```
<mapping to = "shipment">1</mapping>
```

The value of the *to* attribute - `shipment` - here is immediately recognised as the qualifier for shipments. From then onwards, the next set of mappings apply to shipments.

This changes again with the following line:

```
<mapping to = "orderLine">input["Sales Record Number_index"]</mapping>
```

Again, the system recognises that the mapping is to the `orderLine` qualifier, so mappings that follow apply to order lines.

The indexfield attribute

The *indexfield* attribute is not present in import mapping pre-translations. It is used to identify separate top level entities in imported data.

The the CSV which corresponds with the example input handler translation we displayed earlier.

```
"Sales Record Number","Buyer Full name","Buyer Phone Number","Buyer Email"..."Item Title","Quantity"
TEST_12146571,"DREW MARROW","(01249) 750 564","demo@realtimedespatch.co.uk"...,"FLSH8GB",1
TEST_12146571,"DREW MARROW","(01249) 750 564","demo@realtimedespatch.co.uk"...,"BATT2112",2
TEST_12146581,"DAVIS GORMLEY","(01249) 750 564","demo@realtimedespatch.co.uk"...,"FLSH8GB",1
```

From inspection, it is fairly clear that the first two data records relate to the same order, while the third relates to a different order. There needs to be a field which is common to all lines in the order, which can identify the order. This field is known as the index field.

In our mapping transformation, the *indexfield* value is 'Sales Record Number', which also maps to the order *externalReference*.

```
<fieldmapper>
<mappings useinput = "false" indexfield="Sales Record Number">
    <mapping to = "externalReference" from="Sales Record Number"/>
    ... order fields
    <mapping to = "shipment">1</mapping>
    ... shipment fields
    <mapping to = "orderLine">input["Sales Record Number_index"]</mapping>
    ... order line fields
</mappings>
</fieldmapper>
```

Note also that the field 'Sales Record Number' appears again in the *orderLine* mapping entry, where it is used to derive a special variable `Sales Record Number_index` (note the `_index` suffix) which can be used to identify the order line number in the order.

One current limitation of CSV order import is that t only supports import of single shipment orders (note how the value for the shipment mapping is set to `1`). However, it does of course support multiline orders and shipments.

XSLT Input Handler Mapping

OrderFlow also support input content transformation for XML documents using XSLT.

For this, the handler needs to be `import_xslt`.

An **example** of an order document in a bespoke format is shown below:

```
<ORDERS>
    <ORDER>
        <ORDER_ID>TEST_xslt_1</ORDER_ID>
        <ORDER_DATE>2014-04-01</ORDER_DATE>
        <RECIPIENT>
        <ADDRESS>
            <ADRESS_LINE_1></ADRESS_LINE_1>
            <COUNTRY></COUNTRY>
            <POSTCODE></POSTCODE>
        </ADDRESS>
        <NAME>Phil</NAME>
        <MOBILE>0789 123456</MOBILE>
        </RECIPIENT>
        <ORDER_LINE>
        <ORDER_LINE_ID>1</ORDER_LINE_ID>
        <SKU>DVD-ABUG</SKU>
        <QUANTITY>3</QUANTITY>
        </ORDER_LINE>
        <ORDER_LINE>
        <ORDER_LINE_ID>2</ORDER_LINE_ID>
        <SKU>DVD-MATR</SKU>
        <QUANTITY>2</QUANTITY>
        </ORDER_LINE>
    </ORDER>
</ORDERS>
```

The fields in the example above map easily enough to OrderFlow fields.

An XSLT transformation can be set up to transform the data into the OrderFlow canonical format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" indent="no"/>

<xsl:template match="/">
<imports>
        <xsl:for-each select="ORDERS/ORDER">
<import type="order" operation="insert">
        <xsl:attribute name="externalReference">
        <xsl:value-of select="ORDER_ID"/>
        </xsl:attribute>
state=created
placed=<xsl:value-of select="ORDER_DATE"/> 00:00:00
validated=true
currency=GBP
channel=magento
deliveryAddressLine1=<xsl:value-of select="RECIPIENT/ADDRESS/ADDRESS_LINE_1"/>
deliveryAddressLine2=<xsl:value-of select="RECIPIENT/ADDRESS/ADDRESS_LINE_2"/>
deliveryAddressLine3=<xsl:value-of select="RECIPIENT/ADDRESS/TOWN"/>
deliveryAddressLine4=<xsl:value-of select="RECIPIENT/ADDRESS/COUNTY"/>
deliveryCountryCode=<xsl:value-of select="RECIPIENT/ADDRESS/COUNTRY"/>
deliveryPostCode=<xsl:value-of select="RECIPIENT/ADDRESS/POSTCODE"/>
deliveryContactName=<xsl:value-of select="RECIPIENT/NAME"/>
deliveryMobilePhoneNumber=<xsl:value-of select="RECIPIENT/MOBILE"/>
invoiceAddressLine1=<xsl:value-of select="RECIPIENT/ADDRESS/ADDRESS_LINE_1"/>
invoiceAddressLine2=<xsl:value-of select="RECIPIENT/ADDRESS/ADDRESS_LINE_2"/>
invoiceAddressLine3=<xsl:value-of select="RECIPIENT/ADDRESS/TOWN"/>
invoiceAddressLine4=<xsl:value-of select="RECIPIENT/ADDRESS/COUNTY"/>
```

```
invoiceCountryCode=<xsl:value-of select="RECIPIENT/ADDRESS/COUNTRY"/>
invoicePostCode=<xsl:value-of select="RECIPIENT/ADDRESS/POSTCODE"/>
invoiceContactName=<xsl:value-of select="RECIPIENT/NAME"/>
invoiceMobilePhoneNumber=<xsl:value-of select="RECIPIENT/MOBILE"/>
shipment.1.priority=0
shipment.1.deliverySuggestionCode=<xsl:value-of select="DELIVERY_SKU"/>
shipment.1.deliveryInstruction=<xsl:value-of select="DELIVERY_INSTRUCTIONS"/>
shipment.1.orderItem=entity:order
<xsl:for-each select="ORDER_LINE">
orderLine.<xsl:value-of select="ORDER_LINE_ID"/>.product.externalReference=<xsl:value-of select="SKU"/>
orderLine.<xsl:value-of select="ORDER_LINE_ID"/>.quantity=<xsl:value-of select="QUANTITY"/>
orderLine.<xsl:value-of select="ORDER_LINE_ID"/>.state=created
orderLine.<xsl:value-of select="ORDER_LINE_ID"/>.unitPriceGross=<xsl:value-of select="UNIT_PRICE"/>
orderLine.<xsl:value-of select="ORDER_LINE_ID"/>.shipment=entity:shipment.1
</xsl:for-each>
</import>
<xsl:text>&#xA;</xsl:text>
        </xsl:for-each>
</imports>
</xsl:template>

</xsl:stylesheet>
```

Input Handler Content Script

The import handler content can be set to manipulate the text content directly. There are two ways in which this script is typically used:

- to apply generally small modifications to the incoming text itself.
- for complex bespoke incoming data formats.

One example of the former involves interactions with third party systems that send XML data with a *byte order mark* (BOM) with characters in the UTF-8 encoding.

```
return rtd.service.infile.transform.InputScriptUtils.maybeStripByteOrderMark(content);
```

Note the sandboxing rules have been relaxed to allow for some useful functionality to support the string manipulation that is likely to be required when this script is used. Specifically, the following Java packages can be used in scripts:

- `rtd.service.infile.transform`
- `org.apache.commons.collections`

The class `rtd.service.infile.transform.InputScriptUtils` has some useful static methods which can be accessed in the content translation script:

InputScriptUtils Methods

Name	Returns Type	Usage
groupLines(List, String groupDelimiter)	List>	Group lines into several lists, based on the group delimiter.
readLines(String input)	List	Reads the input string into a list of lines.
writeLines(Collection, String ending)	String	Writes the list of lines into a single string, with each line followed by the *ending* value.
stripFirstLines(List, int linesToRemove)	List	Strips the first n lines from the inputted list of lines.
stripLastLines(List, int linesToRemove)	List	Strips the last n lines from the inputted list of lines.
extractQuotedText(String)	String	Extracts string if it is quotation characters, otherwise returns the string as is.
maybeStripByteOrderMark(String)	String	Strips the Byte Order Mark (BOM) from a string, if it is present.
startsAndEndsWith(String, String)	Boolean	Returns true if the string starts and ends with the specific characters.

Note that the content script has no knowledge of the OrderFlow domain model, not does it expect the data to be in any particular format when run.

The content script can also be useful if the need arises to translate an input from a highly custom or bespoke format that does not lend itself easily to more familiar translations involving CSV mappings or XSLT.

The content script can also be used to do more basic transformations such as the following:

- strip the first and/or line(s) from the input text
- remove every second line from the input text

Note that the input content script, if present, will be run *before* the input mapping translation.

Example Usage

Some further examples usages are shown below.

The following example shows how to **strip lines** from the input text, specifically removing the first two lines.

```
def lines = rtd.service.infile.transform.InputScriptUtils.readLines(content);
def newLines = rtd.service.infile.transform.InputScriptUtils.stripFirstLines(lines, 2);
```

```
def output = rtd.service.infile.transform.InputScriptUtils.writeLines(newLines, '\n');
return output;
```

The following example shows how to **filter out lines** that start with the character '#':

```
def lines = rtd.service.infile.transform.InputScriptUtils.readLines(content);
def newLines = [];

for (line in lines) {
    if (!line.startsWith('#')) {
        newLines.add(line);
    }
}

def output = rtd.service.infile.transform.InputScriptUtils.writeLines(newLines, '\n');
return output;
```

Scripting Context

The scripting context for the content script includes the following variables:

input

A reference to the input text string.

content

An alias to the same text string.

Return value

The content script needs to return a text string, which is then used as the input text for further operations.

Note that if no transformation is necessary, the script can return a `null` value, in which case the original input string is used as the input for further operations.

Courier Selection

Courier Selection Script

A courier selection script is the means by which a courier and courier service can be automatically associated with a shipment. This association typically takes place at the point at which the shipment is received, but can also be done at a later stage in the order processing workflow.

Courier selection may be based on a number of factors, including any or all of the factors below (often a combination of these) * whether the shipment is international or domestic * the shipment weight * the customer's choice * the value of the order * the type of products in the shipment * the customer's expected service delivery

All of this information can be determined programatically in a Groovy Script using the OrderFlow API.

A Simple Example

The example below performs a courier selection based primarily on the customer's choice, but also on the shipment weight.

```
def customerchoice = value.deliverySuggestion.code;
def weight = value.weight;
def courierOptions = '';

if (customerchoice == null) {
    throw new IllegalStateException("Unable to determine courier choice.
        Please ensure 'deliverySuggestionCode' shipment attribute is set");
}

customerchoice = customerchoice.toLowerCase();

def courierReference;
def serviceCode;

if (customerchoice.contains("dpd")) {
    courierReference = "dpd";
    serviceCode = "dpd_classic";
}

if (customerchoice.contains("interlink")) {
    s
    if (weight <= 5) {
        courierReference = "dpd";
        serviceCode = "interlink_nextday";

    } else {
        courierReference = "dpd";
        serviceCode = "interlink_nextday_parcel";
    }
}

if (courierReference == null) {
    courierReference = "generic";
    serviceCode = "";
}

values.courierDeliveryInfo.courierReference=courierReference;
values.courierDeliveryInfo.serviceCode=serviceCode;
values.courierDeliveryInfo.courierOptions=courierOptions;

return null;
```

A few points to note:

- the customer's choice is determined here using the Groovy expression `value.deliverySuggestion.code`. If none is set, the script will return an error.
- the shipment weight is expected to be populated.
- if no valid courier selection is made, the courier selection is returned

Scripting Context

SCRIPTING VARIABLES

value

Holds an instance of the Shipment for which the courier selection is being performed.

values

Holds an instance of the context map for the courier selection script. Doesn't typically get used directly, but does hold additional referencable data as described next.

values.courierDeliveryInfo

The `values.courierDeliveryInfo` holds an instance of `rtd.courier.info.CourierDeliveryInfo`. The fields in this class hold data which maps to the selected courier, service and options, as summarised in the table below.

Courier Delivery Info Fields

Name	Description	Maps To
courierReference	The `externalReference` value for the selected courier for the shipment	*shipment.courier.externalReference*
serviceCode	The selected service code from the courier	*shipment.deliveryMethod.serviceCode*
courierOptions	A comma separated list of options	*shipment.deliveryMethod.courierOptions*
mailFormat	Mail format set for this shipment	*shipment.deliveryMethod.mailFormat*
lineHaulSiteReference	Only applies if line hauling is used. If *shipment.lineHaulDestinationSite.externalReference*	

Note the 'Maps To' column above denotes the Groovy expression that can be used to extract from the shipment the value populated using the courier selection script.

Returning to the initial courier selection script example, we can see how the `values.courierDeliveryInfo` expression is populated.

```
values.courierDeliveryInfo.courierReference=courierReference;
values.courierDeliveryInfo.serviceCode=serviceCode;
```

```
values.courierDeliveryInfo.courierOptions=courierOptions;
return null;
```

Note that the **return** statement in the courier selection script is recommended for OrderFlow 3.8.2 and below in order to ensure that the courier selection result does not get interpreted as JSON text (as per a legacy implementation of the courier selection script that is no longer in widespread use).

Further Examples

The example below selects between domestic and international couriers and services depending upon destination and weight.

```
def courierReference;
def serviceCode;

def weight = value.weight;
if (weight == null) {
    throw new IllegalStateException("Unable to determine weight for shipment.");
}

def genericNonInternationalCountryCodes = ['UK','GB','IM','GG','JE','IE'];

def countryCode = value.implicitAddress.countryCode;
if (countryCode == null) {
    throw new IllegalStateException("Unable to determine country code for shipment.");
}

def international = !genericNonInternationalCountryCodes.contains(countryCode);

if (international) {
    courierReference = "royalmail_international_netdespatch";
    serviceCode = "IE1E"
}
else {
    if (weight <= 5) {

        courierReference = "royalmail_domestic_netdespatch";
        serviceCode = "TPN01N";

    }
    else {

        courierReference = "royalmail_domestic_netdespatch";
        serviceCode = "TPH01N";

    }
}

if (courierReference == null) {
        courierReference = "generic";
        serviceCode = "";
    }

values.courierDeliveryInfo.courierReference=courierReference;
values.courierDeliveryInfo.serviceCode=serviceCode;
values.courierDeliveryInfo.courierOptions='';

return null;
```

Useful Expressions

The following Groovy expression are useful when writing courier selection scripts.

Customer Choice

The OrderFlow convention is to receive the delivery suggestion via the shipment delivery suggestion code, as shown in the script below.

```
def customerchoice = value.deliverySuggestion.code;
```

Note that the customer choice can sometimes be mapped to an explicit choice in courier. On other occasions, it may contain information on the service level, or expected time to delivery. It tends to be very environment specific.

Country Code

A reliable way to get the ISO two character country code for a shipment is using the script below:

```
def customerchoice = value.implicitAddress.countryCode;
```

Note the use of `implicitAddress`; if a delivery address has been set at the shipment level, then this will be used. Otherwise, the order delivery address will be used.

Post Codes

The post code can retrieved in a similar way to the country code:

```
def customerchoice = value.implicitAddress.postCode;
```

The post code is often useful to identify shipments coming from more remote destinations on the British Isles, including the Channel Islands (Jersey, Guernsey), Isle of Man, as well as regions such as Northern Ireland and the Scottish Highlands. For some carriers, different rules and restrictions need to be applied for some of these shipments compared to those in more heavily populated regions of Great Britain.

Order Value

The total price for the order paid by the customer can be found using the following expression:

```
def price = value.orderItem.totalPrice.gross;
```

As the total gross price, it includes goods and shipping, as well as all taxes.

Shipping Charges

The shipping charges are set against the order in a similar way:

```
def shippingCharge = value.orderItem.shippingPrice.gross;
```

The amount paid by the customer for shipping may certainly affect the courier choice in some environments.

Priority

The shipment priority field is often used to influence courier choice.

```
def shippingCharge = value.priority;
```

The priority value is set on a numerical scale, with a higher value used for higher priority shipments. The possible priority values and gradations are not set in stone, and will vary according to need. It is certainly common to at least define a distinction between normal and high priority shipments.

Expected Delivery Date

On occasion, the front end eCommerce system may capture or record an expected delivery date by which the customer is expecting to receive their order.

If set, it can be obtained using the following expression:

```
def requestedDeliveryDate = value.requestedDeliveryDate;
```

Country Group

As of OrderFlow 4.2.0.1 the following expression can be used to determine whether the destination country is in a particular country group. This can be useful for scripts that will apply different courier selection for EU countries as opposed to countries outside of the EU.

See below for an example:

```
def countryLookup = values.countryLookup;
if (!countryLookup.isInCountryGroup('eu',value.implicitAddress.countryCode)) {
    options = 'international_only';
}
```

Unit Testing

> **A Note on Unit Testing**
>
> This section requires the that you have in place the OrderFlow integration and scripting environment. If you are interested in having this set up in your environment to enable you to write your own unit tests, please contact the OrderFlow support team.
>
> For complex courier selection scripts, a accompanying unit test is highly recommended (and mandatory if implemented by OrderFlow Technical Staff).
>
> Writing a unit test should generally be done at the same time or even before the courier selection script is written, in line with the principles of Test Driven Development.

WRITE TEST

Unit testing of courier selection can be done using a unit test which extends `BaseCourierSelectionScriptTest`.

This test which defines a `run(packageName, scriptName, shipment)` method, which provides a convenient way of setting up the scripting context required for the courier selection script.

The **packageName** parameter is the package name containing the script. The **scriptName** is the name of the file containing the courier selection script. The **shipment** is an instance of `rtd.domain.Shipment`, which needs to be instantiated in code.

An example of a unit test is shown below:

```
/**
* Courier selection script test for 'orderflow.courier.selection.script'.
*
* @author Phil Zoio
*/
public class OrderFlowCourierSelectionScriptTest extends BaseCourierSelectionScriptTest {

    private Shipment shipment;
    private OrderItem orderItem;

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        setupShipment();
    }

    /**
    * Do basic shipment setup
    */
    void setupShipment() {
        shipment = new Shipment();
        orderItem = new OrderItem();
        shipment.setOrderItem(orderItem);
        orderItem.getTotalPrice().setGross(100D);
    }

    void runSelection() {
        run("rtd.orderflow.courier", "orderflow.courier.selection.script", shipment);
    }

    public void testGBHighValue() throws Exception {

        shipment.getAddress().setCountryCode("GB");

        String expectedServiceCode = null;
        String expectedOptions = null;
        expectCourier("dpd", expectedServiceCode, expectedOptions);
    }

    public void testGBLowValue() throws Exception {

        orderItem.getTotalPrice().setGross(10D);
        shipment.getAddress().setCountryCode("GB");

        expectCourier("royalmail_tracked", "TPN01", "serviceOption=signature");
    }

    public void testEU() throws Exception {

        shipment.getAddress().setCountryCode("DE");

        String expectedServiceCode = null;
        String expectedOptions = null;
        expectCourier("ups", expectedServiceCode, expectedOptions);
    }

    public void testUS() throws Exception {

        shipment.getAddress().setCountryCode("US");

        String expectedServiceCode = null;
        String expectedOptions = null;
        expectCourier("ups_international", expectedServiceCode, expectedOptions);
    }

    void expectCourier(String expectedCourier, String expectedServiceCode, String expectedOptions) {
        runSelection();
        System.out.println(courierDeliveryInfo);
```

```
        assertEquals(expectedCourier, courierDeliveryInfo.getCourierReference());
        assertEquals(expectedServiceCode, courierDeliveryInfo.getServiceCode());
        assertEquals(expectedOptions, courierDeliveryInfo.getCourierOptions());
    }
}
```

Note that in writing the unit test, your responsibilities are:

- identify a scenario that needs to be tested. For this, define the *outcome*, through a call to the `expectCourier(String expectedCourier, String expectedServiceCode, String expectedOptions)` method above.
- set up the `shipment` instance so that it meets to preconditions for testing the outcome.

You will want to repeat this process for each of the outcomes to be tested, in order that all outcomes are adequately tested.

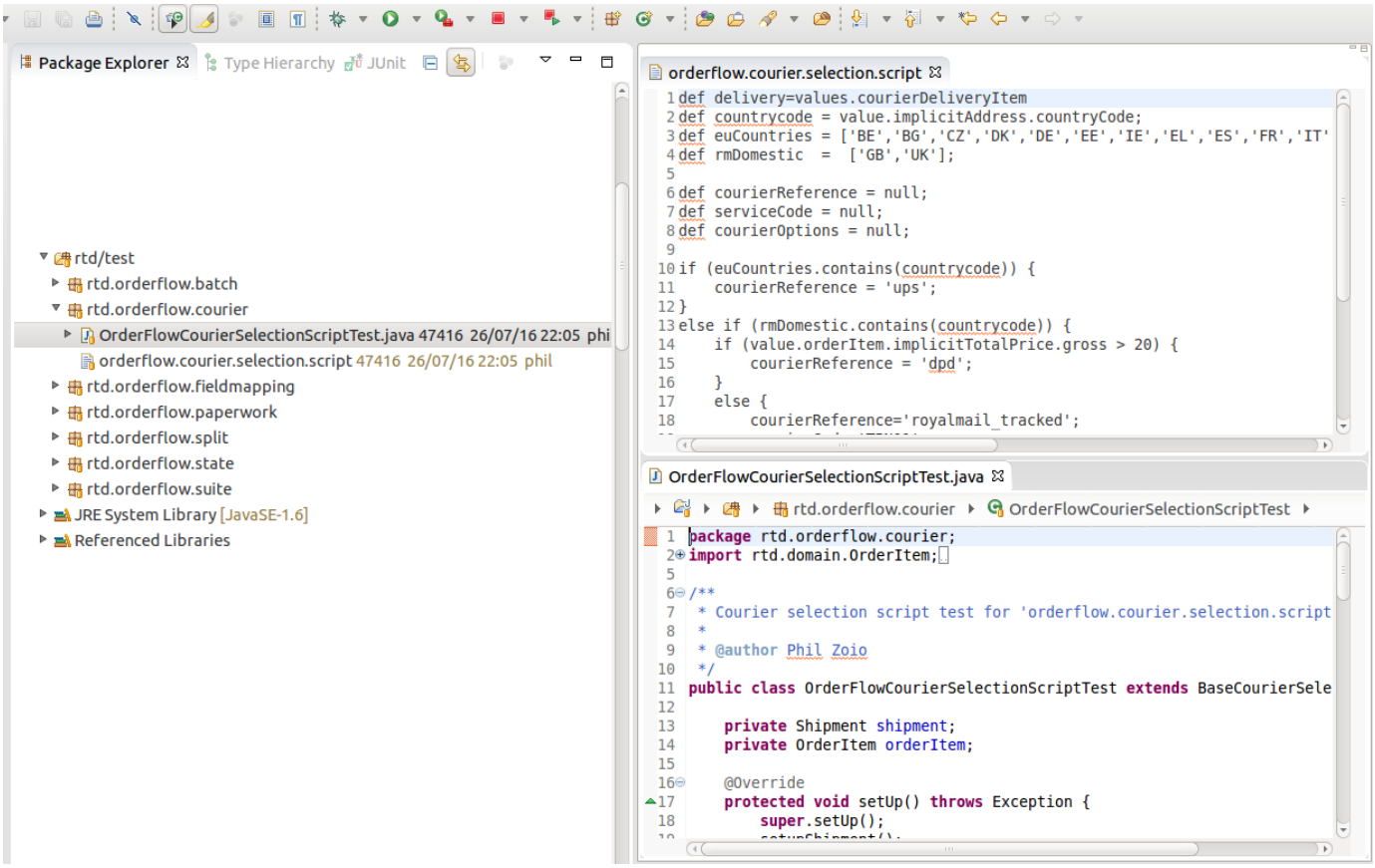
WRITE SCRIPT

You will then need to write the script in a way that satisfies the outcome. The example script tested using the code above:

```
def delivery=values.courierDeliveryItem
def countrycode = value.implicitAddress.countryCode;
def euCountries = ['BE','BG','CZ','DK','DE','EE','IE','EL','ES','FR','IT','CY',
                   'LV','LT','LU','HU','MT','NL','AT','PL','PT','RO','SI','SK','FI','SE']
def rmDomestic  =  ['GB','UK'];

def courierReference = null;
def serviceCode = null;
def courierOptions = null;

if (euCountries.contains(countrycode)) {
    courierReference = 'ups';
}
else if (rmDomestic.contains(countrycode)) {
    if (value.orderItem.implicitTotalPrice.gross > 20) {
        courierReference = 'dpd';
    }
    else {
        courierReference='royalmail_tracked';
        serviceCode='TPN01';
        courierOptions='serviceOption=signature';
    }
} // Test to capture all country codes not specifically listed in euCountries and rmDomestic
else {
    courierReference = 'ups_international';
}

values.courierDeliveryInfo.courierReference=courierReference;
values.courierDeliveryInfo.serviceCode=serviceCode;
values.courierDeliveryInfo.courierOptions=courierOptions;
```

IDE SETUP

A screenshot showing these in an Eclipse IDE project environment is shown below.



Note the use of the test package naming convention, which follows the along the lines: ... In line with this convention, the test and script itself a re contained in the *courier* subpackage.

MetaPack Booking Code

Documentation to be added.

Hypaship Delivery Group

Documentation to be added.

Batch Selection

Batch Selection Script

The batch selection script is the script which is used to determine which shipments get picked together, and presented to the packer, in a group.

The batch selection script determines the *type* of the shipment batch, which in turn defines the grouping mechanism.

For more details on shipment batching, see the 'Batching' chapter in the OrderFlow Advanced Concepts Guide.

Batching Strategies

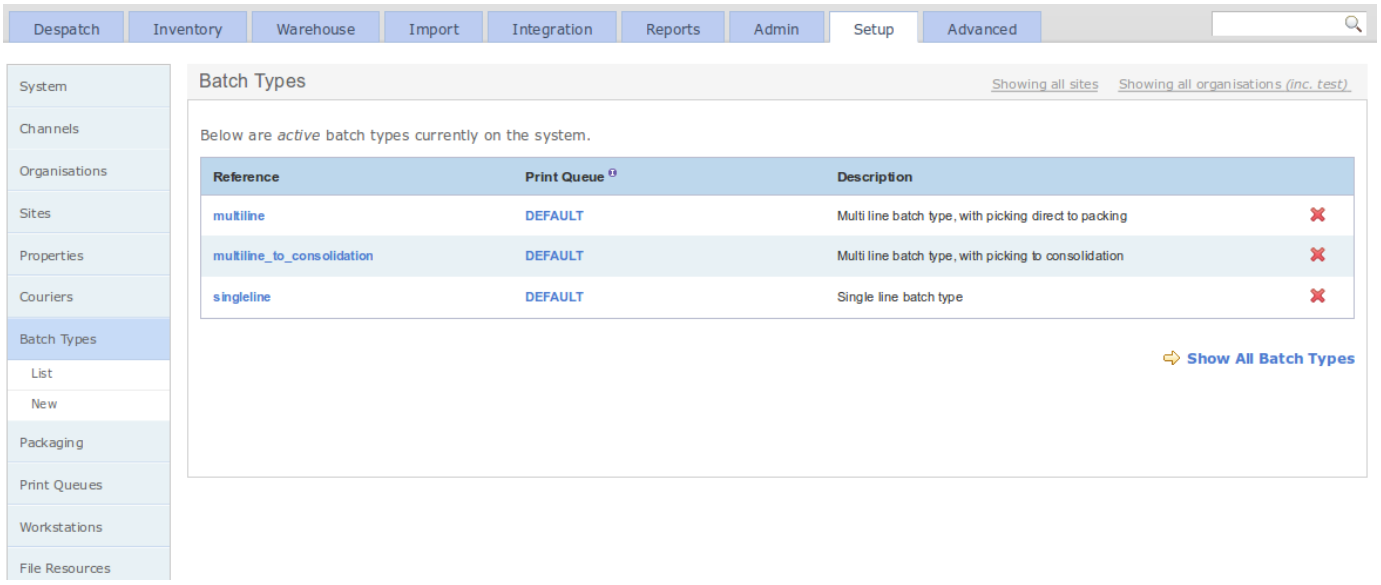
A wide range of strategies may be employed to determine the batch type of a shipment:

- whether the order is single or multiline, or even the number of lines
- the courier
- the priority (e.g urgent vs normal)
- products with a particular set of characteristics
- order lines being picked from a particular area in the warehouse

The actual logic to be used will depend on business requirements, which may in turn either be driven by commitments to the end customer, or a drive for more efficient warehousing operations.

Batch Types

At a technical rather than business level, the purpose of the batch selection script is to identify the appropriate batch type to be used. Available batch types can be found from the **Setup -> Batch Types** menu, as shown below.



The identifier for the batch types is in the first column, e.g. `singleline`, `multiline`. A valid batch selection script will apply business rules to determine one of the available batch types.

If the batch type needed to implement the required business logic is not present, part of the batch selection implementation task will be to create the new batch type(s).

A Simple Example

The example below is a script which implements very simple logic to select the `multiline` batch type for multiline shipments, and the `singleline` batch type for single line shipments.

```
if (value.orderLineCount > 1) return 'multiline';
else return 'singleline';
```

Note that the **return value** for the batch selection script is important; the script needs to return the identifier for the required batch type.

Scripting Context

The following scripting variables are available for use in a batch selection script.

value

Holds an instance of the Shipment for which the courier selection is being performed.

values

Holds an instance of the context map for the courier selection script. Doesn't typically get used directly, but does hold additional referencable data as described next.

values.populators

Holds a reference to populators that can be used to further populate data already in the scripting context.

Consider for example, the *shipment*. The product associated with the first order line in the shipment can be found for each order line using code such as the following:

```
def orderLine = shipment.orderLines.iterator().next();
def product = orderLine.product;
```

However not all fields in the products are automatically populated for efficiency reasons. For example, product attributes would not be directly referencable.

In order to get access to product attributes, you would need to fully populate the product, using the following script.

```
def orderLine = shipment.orderLines.iterator().next();
def product = orderLine.product;
def populatedProduct = values.populators['product'].populate(product);
```

Note that the populator defined above will not necessarily be available in all scripting environments, but will be available for batch selection.

Useful Expressions

The following expressions and snippets are useful in batch selection scripts:

PRIORITY

The following expression will retrieve the priority of the shipment, which can be used to make decisions on the shipment's urgency.

```
def priority = value.priority;
```

COURIER

The courier and service reference are often used in batch selection scripts where shipments to go out with particular couriers need to be picked together.

```
def courier = value.courier?.externalReference;
def serviceCode = value.deliveryMethod?.serviceCode;
```

LINE AND ITEM COUNT

The following expressions can be used to determine whether a shipment has multiple lines or even multiple *items*. (A shipment with a single line with a quantity of 2 is considered *multi-item*.)

```
def multiLine = (value.orderLineCount > 1);
def multiQuantity = value.orderItemCount > 1;
```

ADDRESS INFORMATION

The following expressions retrieve the country code and post code for a shipment, using the `implicitAddress` expression for generality. The post code value is often used for shipments in more remote destinations on the British Isles.

```
def countryCode = value.implicitAddress?.countryCode?.toUpperCase();
def postCode = value.implicitAddress?.postCode?.toUpperCase();
```

A More Complex Example

The example below returns batch types according the following logic:

Batch Types

Batch	Priority	Quantity	Destination
priority-singleitem	High	Single item	GB and the Channel Islands
standard-singleitem	Standard	Single item	GB and the Channel Islands
priority-multiitem	High	Multi-item	GB and the Channel Islands
standard-multiitem	Standard	Multi-item	GB and the Channel Islands
priority-singleitem-de	High	Single item	Germany
standard-singleitem-de	Standard	Single item	Germany
priority-multiitem-de	High	Multi-item	Germany
standard-multiitem-de	Standard	Multi-item	Germany
priority-singleitem-eu	High	Single item	Rest of EU
standard-singleitem-eu	Standard	Single item	Rest of EU
priority-multiitem-eu	High	Multi-item	Rest of EU
standard-multiitem-eu	Standard	Multi-item	Rest of EU

A script which implements this logic is show below:

```
def multiQuantity = value.orderItemCount > 1;
def countryCode = value.implicitAddress?.countryCode?.toUpperCase();
def domesticCountryCodes = ['GB','UK','JE','GG','IM'];

def priority = shipment.priority;

def suffix = '';
if (countryCode == 'DE') {
    suffix = '-de';
} else if (!domesticCountryCodes.contains(countryCode)) {
    suffix = '-eu';
}

if (priority >= 10) {
    if (multiQuantity) {
```

```
        return 'priority-multiitem'+suffix;
    } else {
        return 'priority-singleitem'+suffix;
    }
} else {
    if (multiQuantity) {
        return 'standard-multiitem'+suffix;
    } else {
        return 'standard-singleitem'+suffix;
    }
}
```

Unit Testing

> **A Note on Unit Testing**
>
> This section requires the that you have in place the OrderFlow integration and scripting environment. If you are interested in having this set up in your environment to enable you to write your own unit tests, please contact the OrderFlow support team.
>
> For complex batch selection scripts, a accompanying unit test is highly recommended (and mandatory if implemented by OrderFlow Technical Staff).
>
> Writing a unit test should generally be done at the same time or even before the batch selection is written, in line with the principles of Test Driven Development.

WRITE TEST

Unit testing of batch selection can be done using a unit test which extends `BaseBatchSelectionScriptTest`.

```
public class OrderFlowBatchSelectionScriptTest extends BaseBatchSelectionScriptTest {

    private Address address;

    protected String getScriptPackageName() {
        String packageName = "rtd.orderflow.batch";
        return packageName;
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        address = new Address();
        address.setLine1("line1");
        shipment.setAddress(address);
    }

    public void testGBShipment() throws Exception {
        address.setCountryCode("GB");

        shipment.getDeliveryMethod().setServiceCode("first");

        shipment.addOrderLine(newOrderLine());
        expect("orderflow-singleitem", shipment, "orderflow.batch.selection.script");

        shipment.addOrderLine(newOrderLine());
        expect("orderflow-multiitem", shipment, "orderflow.batch.selection.script");
    }

    public void testDEShipment() throws Exception {
        address.setCountryCode("De");

        shipment.addOrderLine(newOrderLine());
        expect("orderflow-singleitem-de", shipment, "orderflow.batch.selection.script");

        shipment.addOrderLine(newOrderLine());
        expect("orderflow-multiitem-de", shipment, "orderflow.batch.selection.script");
    }

    public void testEUShipment() throws Exception {
        address.setCountryCode("fr");

        shipment.addOrderLine(newOrderLine());
```

```
        expect("orderflow-singleitem-eu", shipment, "orderflow.batch.selection.script");

        shipment.addOrderLine(newOrderLine());
        expect("orderflow-multiitem-eu", shipment, "orderflow.batch.selection.script");
    }

    private void expect(String expected, Shipment shipment, String scriptFile) {
        assertEquals(expected, run(scriptFile, shipment, expected, false));
    }

    private String run(final String scriptFile, Shipment shipment, String expected, boolean highVolume) {
        Map<String,Object> parameters = new HashMap<String, Object>();
        parameters.put("expected", expected);
        parameters.put("highVolume", highVolume);
        return run(scriptFile, shipment, parameters);
    }

}
```

The `BaseBatchSelectionScriptTest` defines a `run(scriptFile, shipment, parameters)` method which is useful for setting up the scripting context for the batch selection script.

Note that the `run()` method returns the batch selection returned from the batch selection script, which can be compared with an expected result.

The responsibility of the script developer is to return identify all of the scenarios that need to be covered in the selection script, and ensure that the correct value is returned for each scenario.

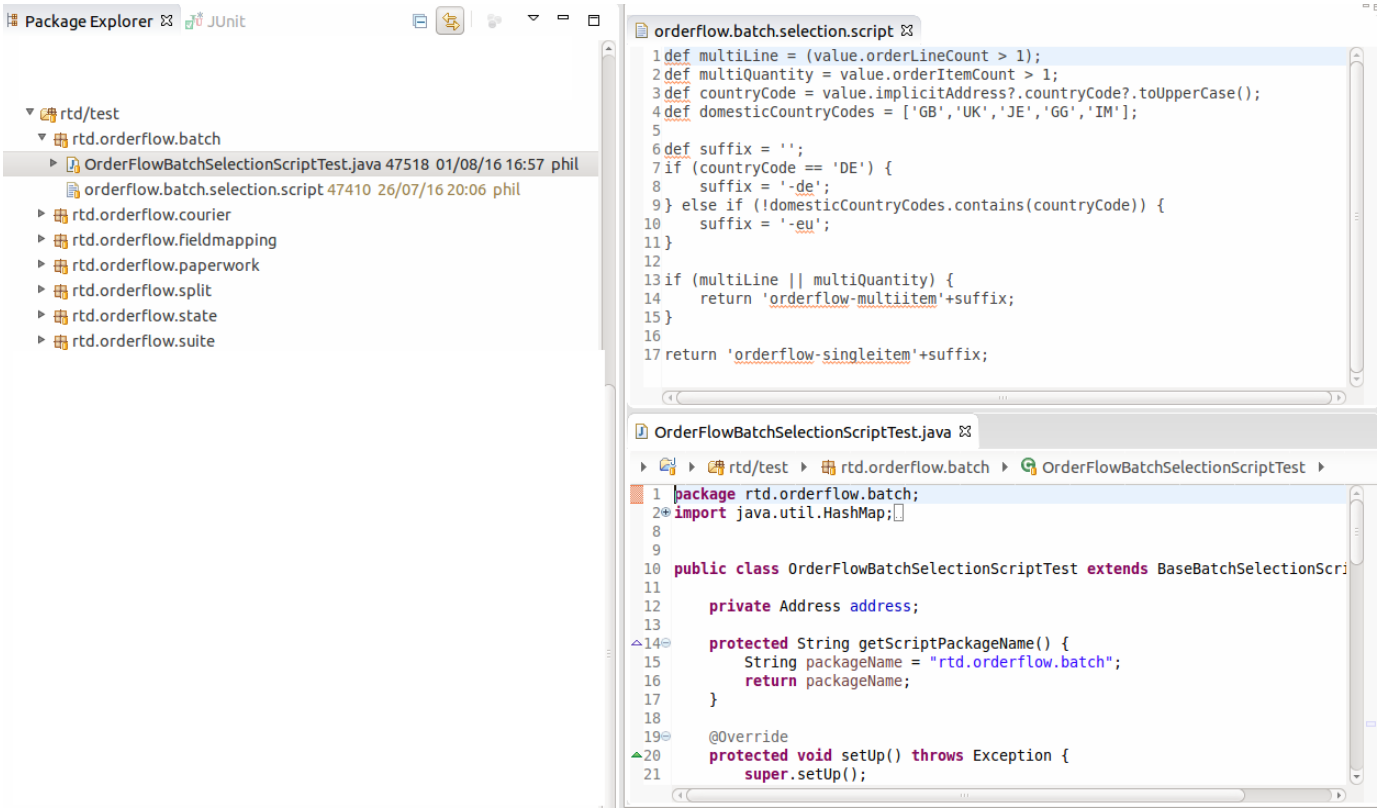
WRITE SCRIPT

The script developer will then write a script for which all of the tests pass. The script corresponding to the above unit test is shown below:

```
def multiLine = (value.orderLineCount > 1);
def multiQuantity = value.orderItemCount > 1;
def countryCode = value.implicitAddress?.countryCode?.toUpperCase();
def domesticCountryCodes = ['GB','UK','JE','GG','IM'];

def suffix = '';
if (countryCode == 'DE') {
    suffix = '-de';
} else if (!domesticCountryCodes.contains(countryCode)) {
    suffix = '-eu';
}

if (multiLine || multiQuantity) {
    return 'orderflow-multiitem'+suffix;
}

return 'orderflow-singleitem'+suffix;
```

IDE SETUP

A screenshot showing these in an Eclipse IDE project environment is shown below.



Note the use of the test package naming convention, which follows the along the lines: ... In line with this convention, the test and script itself a re contained in the *batch* subpackage.

Printing and Paperwork

Despatch Note Keys

As part of the pack and despatch process, customer paperwork invariably needs to be generated to be included with the outgoing shipment. This paperwork is typically referred to as the 'despatch note', and sometimes as the 'customer invoice'.

OrderFlow provides a report-based mechanism for implementing this, which is discussed in a lot more detail in the Despatch Note chapter of the OrderFlow Report Writers' Guide.

Once the despatch note has been implemented and added to the appropriate instance of OrderFlow, there needs to be a mechanism by which the despatch note is associated with a shipment, so that when the packer clicks on the Print Despatch Note button, the correct paperwork comes out the printer.

The association of the shipment with the appropriate paper at this point in the process is done through another Groovy script, called the *despatch note keys* script. The despatch note keys script applies logic to determine which report key, or set of compound report keys, need to apply for the output of the appropriate despatch note content.

The content of the despatch note keys script, once written, is applied to application property called *despatch.note.keys*.

Examples

The simplest despatch note keys script simply returns a single literal value:

```
mydespatchnote
```

Technically, this is not even a script, but a single literal value. The scripted equivalent of the above is:

```
return 'mydespatchnote'
```

In both cases, the script assumes that there is a despatch note report with the identifier or key 'mydespatchnote'.

The non-trivial but still very simple implementation might involve the case where more than one *brand* is sold through a single sales channel, and each of these brands requires its own despatch note. An example is shown below:

```
def shipment = value;
def order = shipment.orderItem;
def brand = order.brand;

if (brand == 'coolshades') {
    return 'coolshades_despatchnote';
} else if (brand == 'classyeyeware') {
    return 'classyeyeware_despatchnote';
}
throw new UnsupportedOperationException('Despatch note printing not supported for unknown brand ' + brand);
```

Scripting Context

The key input in the scripting context is the *shipment*, which can be accessed in the script using the **value** variable. As the above example shows, the order can easily be navigated, and using other properties in the OrderFlow data model, so can the order line data.

The **return value** for the despatch note keys script will always be a text string, which will identify the required report key(s).

Applying the Script

The script can be applied by updating the application property *despatch.note.keys*. Note that this property is *scoped*, which means that different values can be set up for different organisations, channels and sites. The script associated with the same scope as that of the shipment will be used when selecting the correct instance of the script to apply.

Scripts with subreports

The despatch note keys script can be set up to return a compound report key, which can be used for despatch notes with compound reports which may contain separate subreports for embedded courier or return labels, return forms or other bespoke elements.

An example of a returned value in this scenario may be:

```
return
'masterreport,despatch_note=despatchnote_report,courier_label=generic_label,return_form=basic_return_form_report
```

In the case above, the following reports are used:

Compound Reports

Report Key	Decription
masterreport	The master report, used as a container for the contained subreports
despatchnote_report	The report key associated with the 'despatch_note' subreport parameter in the master report
generic_label	The report key associated with the 'courier_label' subreport parameter in the 'despatch_note' report
basic_return_form_report	The report key associated with the 'return_form' subreport parameter in the master report

As the above example indicates, it is possible to have subreports that container other subreports, at the cost of some extra complexity in the configuration and maintenance of these reports.

See the Despatch Note section of the OrderFlow Report Writers Guide for more details on how subreports are set up in OrderFlow despatch notes.

Dealing with Courier Label Subreports

When integratied stationery is used for courier labels, then one of the subreports described in the previous section will often be for a courier label. The problem here is that the identity of the courier label report will often depend on the courier selected, which in turn means that the *despatch.note.keys* script gets 'polluted' with courier-specific logic.

From OrderFlow 3.7.4 for this can be addressed by using the placeholder `[courier_label]` in the *despatch.note.keys* script, instead of the courier-specific value, as shown in the example below.

```
return
'masterreport,despatch_note=despatchnote_report,courier_label=[courier_label],return_form=basic_return_form_repo
```

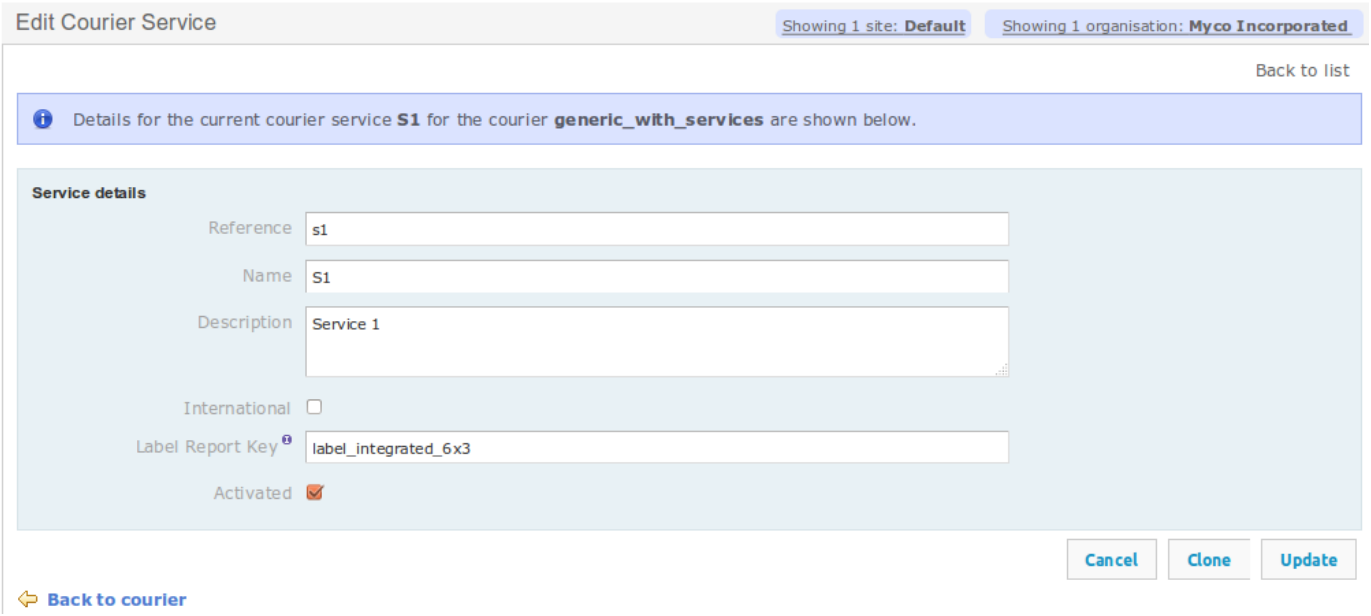
Setting up the actual courier label to be used can now be done using the courier configuration, from the **Setup -> Courier** menu.

For this, there are two ways that the courier label subreport key can be identified, described in the next sections.

Note that if the *despatch.note.keys* has a `[courier_label]` placeholder, the system will raise an error if neither of the mechanisms described below is able to resolve the courier label report key to use.

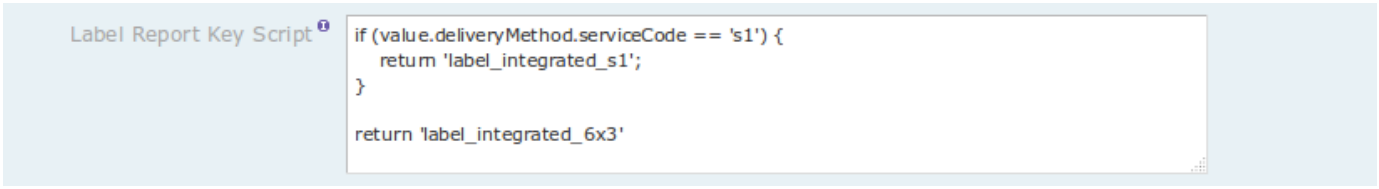
USING A COURIER SERVICE VALUE

If courier service entries are present, then the 'Label Report Key' property can be used to identify the report for the courier label. This is done on a per service basis. Note that the present in this field must be a specific or 'literal' value, rather than a scripted value.



USING A SCRIPT VALUE AT THE COURIER LEVEL

If no courier service value can be found using the mechanism just described, the system will use the scripted value set at the *Courier* level using the 'Label Report Key Script' property, and example of which is shown below.



The courier level mechanism is suitable if individual service entries have not been set for the courier, or if it is easier to specify the values to be used at the courier level. An example of where this might occur is if only a single label report key needs to be used for all services.

Note that **scripting context** binds the shipment to the variable `value`. In the example script, the expression `value.deliveryMethod.serviceCode` returns the value of shipment's service code.

UNIT TESTING

> **A Note on Unit Testing**
>
> This section requires the that you have in place the OrderFlow integration and scripting environment. If you are interested in having this set up in your environment to enable you to write your own unit tests, please contact the OrderFlow support team.

An example unit test for an earlier example is shown below:

```
public class BrandDespatchNoteSelectionScriptTest extends BaseDespatchNoteSelectionScriptTest {

    protected String getCondition() {
        return ClasspathResourceUtils.readClassPathResource("rtd.orderflow.paperwork",
"brand.despatch.note.script");
    }

    public void testBrand() {
        orderItem.setBrand("classyeyeware");
        assertEquals("classyeyeware_despatchnote", runScript());

        orderItem.setBrand("coolshades");
        assertEquals("coolshades_despatchnote", runScript());

        orderItem.setBrand("unknownspecs");
        try {
        runScript();
        fail();
        } catch (UnsupportedOperationException e) {
        assertEquals("Despatch note printing not supported for unknown brand: unknownspecs.", e.getMessage());
        }
    }
}
```

Note the following points on this unit test:

- the script file itself is located in a file called *brand.despatch.note.script* in the package *rtd.orderflow.paperwork*.
- the test class extends `BaseDespatchNoteSelectionScriptTest`, which defines a `runScript()` method, which deals with setting up the scripting context.
- the test methods(s) modify the orders and shipments as required. The assertions then verify that the script returns the expected report keys value.
- if an exception is thrown during script execution, this can also be tested, as shown in the example above.

Shipment Label Keys

Document Keys

Workstation Printer Property Lookup

Workflow

Submitted Order Validation

State Transition

Pickable Shipment

An example of this is as follows, based on the 'despatch.can.pick.shipment.script' application property:

```
def shipment = value;
if (shipment.paidFor) {
   return true;
} else {
   return false;
}
```

Shipment Split

Consolidation Picking Queue

Groovy Scripting

The main scripting language in use in OrderFlow is Groovy.

The advantages of using Groovy are as follows:

- Groovy is a Java Virtual Machine (JVM) based language, and integrates seamlessly with the rest of OrderFlow, which is built using Java.
- Groovy is both immensely powerful but also simple to learn.

Groovy Basics

There are plenty of online tutorials on Groovy. For OrderFlow scripting we tend to rely only on its most basic features, in areas such as variable assignment, logical expressions, conditional expressions, and looping.

Variable Assignment

Done using the `def` keywork:

```
def count = 0; //integer

def state = 'ready'; //string

def map = ['key1':'value1', 'key2':'value2'];
map['key3'] = 'value3';

def list = [1,2];
list.add(3);
```

Note the format used for map and list literals.

Note that map values can also be retrieved using 'dot' notation, as in the example below.

```
def map = ['key1':'value1', 'key2':'value2'];
map.key3 = 'value3';
print map.key3;
```

Note that there are **restrictions** in the names of keys that can be used in dot notation.

Consider the example:

```
def map = ['two word key':'value'];
print map['two word key'];
```

You cannot use as a substitute for the second line:

```
print map.two word key;
```

However, the example below with the single quotes will work:

```
print map.'two word key';
```

Logical expression

Logical expressions are usually done using the **if** statement, followed by a conditional expression.

```
if (count > 10 || state == 'ready') {
    //do something
} else if (count <= 10 && state == 'ready') {
    //do another thing
} else {
    //do something else
}
```

The conditional expression can be a compound statement, with elements joined by `&&` (and) or `||` (or), operators.

Note that it is also possible to use a *switch* statement for:

```
switch (input["storeId"]) {
    case "1": return "Default Store";
    case "2": return "US Store";
    case "3": return "European Store";
    default:  throw new IllegalArgumentException("Unrecognised Store Id");
}
```

In general, our recommendation is to use `if` statements if the number of outcomes is no more than three. The `switch` statement is more intuitive if there are a large number of potential outcomes.

Looping

There are different ways to do looping. The most commonly used looping construct is with the `for` keyword, as shown below:

```
for (line in shipment.orderLines) {
    println line.product.externalReference;
}
```

Return values

The value returned from a script execution is best controlled using a `return` statement, as below:

```
return 'somevalue';
```

Note that if no return statement is contained, then the script will return the value of the most recently evaluated expression. As it is not always very easy to identify what this will be, it is best practice to explicitly use return statement when relying on the value returned from a script.

Null Values

Groovy can access information from the OrderFlow data model using compound expressions such as:

```
shipment.site.externalReference
```

Note that if the site has not yet been set on the site, the code above will fail with a `NullPointerException` (NPE).

Groovy provides a very simple and useful way to write 'NPE safe' expressions, using the `?` operator. In the case above, the following code will not fail with a `NullPointerException`, but will instead will return a `null` value.

```
shipment.site?.externalReference
```

Formatting and Parseing

Groovy has some useful formatting functionality which can be useful, for example, for formatting dates and numbers:

Date Formatting

Dates can be formatted using code such as the following:

```
def dateString = new java.text.SimpleDateFormat('yyyy-MM-dd hh:mm:ss').format(shipment.created);
println date;
```

The date format can obviously be set to the required value. There are many tutorials on the internet on how to use `SimpleDateFormat` for this purpose.

Date Parseing

The class `java.text.SimpleDateFormat` can also parse a date from a text string. This is useful for converting dates from a different format.

```
def date = new java.text.SimpleDateFormat('dd/MM/yy hh:mm:ss').parse('30/10/15 23:11:15');
def newDateString = new java.text.SimpleDateFormat('yyyy-MM-dd hh:mm:ss').format(date);
println newDateString;
```

Number Formatting

Groovy scripts can be used for number formatting in a number of ways. A simple example is shown below, which formats a decimal with two decimal character.

```
println String.format('%.2f', 1.345)
```

which will output the value `1.35`.

Scripting Context

An important concept to understand with scripting is the *scripting context*.

Sometimes you may see scripts that variables that don't appear to have been 'defined'. An example is shown below:

```
if (input['key'] == 'testvalue') {
    ...
}
```

In the above snippet, there is no declaration of the `input` variable. Instead, the `input` variable has been added to the *scripting context* independently of the script itself.

Scripts in OrderFlow almost always rely on some scripting context to achieve a result. For example, for a script that need to operate on or extract data from a `shipment`, the shipment will invariably be present in the scripting context.

The details of the scripting context for the different types of scripts run in OrderFlow are covered in the different sections of this document.

Sandboxing

The Groovy that can be executed within OrderFlow environments is tightly controlled to avoid the risk of undesirable or risky code being added to scripts.

However, by default, the following applies:

- methods or constructors cannot be called in any of the 'system' classes in the `java.lang` package, including `System`, `ClassLoader`, `Thread`, `Runtime` or `Security`.
- objects can only be instantiated from the packages `java.lang`, `java.util` and `java.text`.

For certain scripting context, these limitations are relaxed slightly to allow for access to methods in specific packages to support functionality likely to be required in that scripting environment.