Cookbook - Custom reports

Overview

The platform provides a flexible mechanism to create report plugins that will naturally fit into existing report wizard interface, thus system integrator only need to concentrate on the business functionality around generating specific report.

Report API provides several interfaces for the report plugin:

- ReportDescriptor - which defines the report unique ID and parameters that is requires/allows
- ReportWorker - which allows to provide options for "select" style parameters and generates the the data for the report
- ReportGenerator - which converts data object created by ReportWorker into a report file

Extension points

By default most of the reports would be configured for the Admin app to provide business users with various statistical and accounting data. All out of the box report definitions are specified in Spring context file **manager-report.xml**. For customisations it is advisable to create your own maven module that would have **core-module-reports** as a dependency and would specify **adm-servlet-ext.xml** to naturally be included in the application.

Namely **reportDescriptors** list and **reportWorkers** map would need to be extended in order to include Spring beans of your custom reports.

In terms of storefront (i.e. customer facing application), you can generate ReportDescriptor object programmatically and trigger report generator with specific data in order to generate report file. We will see how this can be accomplished by looking at the delivery report example.

Delivery report, learn by example

Delivery report represents a PDF invoice that can be generated for any given order. We will look into the particulars of the report configuration, which should give an idea of how the whole report framework works both in Admin app and the storefront.

Admin app

Custom implementation 3.6.0

> This is old style method which is not recommended anymore. It does however show some concepts of reporting framework which are useful. The user is advised to use extension point (see next section) to create custom report implementations and inject additional custom reports.

As mentioned in the extension point section report Spring beans are defined in the **manager-report.xml**. Specifically if we examine how delivery report (with id **reportDelivery**) is configured we see the following bean definitions.

ReportDescriptor bean which is part of the **reportDescriptors** list

```
<bean id="reportDelivery" class="org.yes.cart.report.ReportDescriptor">
      <property name="reportId" value="reportDelivery"/>
      <property name="xslfoBase" value="client/order/delivery"/>
      <property name="parameters">
          <list>
              <bean class="org.yes.cart.report.ReportParameter">
                  <property name="parameterId" value="orderNumber"/>
                  <property name="businesstype" value="String"/>
                  <property name="mandatory" value="true"/>
              </bean>
          </list>
      </property>
  </bean>
```

The descriptor defines **reportId** (unique ID for this report plugin), **xslfoBase** (PDF report specific layout file) and **one mandatory parameter** orde rNumber.

DeliveryReportWorker bean which is part of the **reportWorkers** map

```
      <entry key="reportDelivery">
            <bean class="org.yes.cart.report.impl.DeliveryReportWorker">
                <constructor-arg index="0" ref="customerOrderService"/>
                <constructor-arg index="1" ref="shopService"/>
                <constructor-arg index="2" ref="shopFederationStrategy"/>
            </bean>
        </entry>
```

And **AdminReportGeneratorImpl** bean which is the default implementation of the report generator which is backed by XSLFO to generate PDF files.

Thus with this configuration what will happen is that:

1. Business user opens Reports section in Admin app
2. **ReportService** will generate list of all available reports using **getReportDescriptors()** that will populate the selector
3. Business user chooses specific report and clicks add tab
4. **ReportService** will run the **getParameterValues()** that will populate all available selection values
5. Business user clicks run button to generate the report
6. **ReportService** triggers **generateReport()** with selected parameter values that uses report worker to generate the data object and passes it to the report generator to generate report file. Specifically **AdminReportGeneratorImpl** will serialize data object into XML and then load the XSLFO template (i.e. client/order/delivery.xslfo), then generate the PDF using the Apache FOP library.
7. Business user can use the file download facility to download the generated file

> For generating alternative report files (e.g. CSV, Image charts) you need to override the **reportGenerator** bean and provide a composite **ReportGenerator**implementation that can switch between the different report generator implementation depending on the report descriptor configurations.

Reports via extension 3.7.0+

Reports framework has been simplified in version 3.7.0 to fully use extensions capability.

There are three extension points to allow to reconfigure the system without any changes to the core: **reportDescriptors**, **reportWorkers** and **rep ortGenerators**.

Report descriptor extension allow to add new or override existing report definitions. Similarly report workers extension allow to add new or override existing implementations.

Lastly report generator API is now a facade for report generator plugins, which can be defined in report generators extension point. This allows writing custom generators in custom modules without interfering with core code.

Out of the doc the platform now support two kinds of generator plug-ins: PDF and Excel report generators.

Storefront

Normally storefront would not need custom reports. However there are some cases when a report file needs to be generated. Such as the case with delivery report that can be used for providing customers with downloadable PDF invoice files for the orders they have placed.

Since the report generation is very much dependent on the user journey (i.e. for invoice to be downloaded the customer need to find the order in order history and then click the download PDF button) it makes sense to skip some configurations and provide specific facade functions that encapsulate the programmatic generation of the report file.

Delivery report is fully configured in the **CheckoutServiceFacadeImpl.printOrderByReference()** method that programmatically creates a report descriptor and then triggers the PDF report generator with this descriptor object and the order object from the order history page.

```
 private ReportDescriptor createReceiptDescriptor() {
        final ReportDescriptor receipt = new ReportDescriptor();
        receipt.setReportId("reportDelivery");
        receipt.setXslfoBase("client/order/delivery");
        final ReportParameter param1 = new ReportParameter();
        param1.setParameterId("orderNumber");
        param1.setBusinesstype("String");
        param1.setMandatory(true);
        receipt.setParameters(Collections.singletonList(param1));
        return receipt;
    }

    @Override
    public void printOrderByReference(final String reference, final
OutputStream outputStream) {

        final CustomerOrder order =
customerOrderService.findByReference(reference);
        if (order != null) {
            final Pair data = new Pair(order, order.getDelivery());

            final Map<String, Object> values = new HashMap<>();
            values.put("orderNumber", order.getOrdernum());
            values.put("shop", order.getShop());

            reportGenerator.generateReport(
                    createReceiptDescriptor(),
                    values,
                    data,
                    order.getLocale(),
                    outputStream
            );

        }
    }
```

Resulting file is sent to the output stream thus allowing this method to be used with URL mapping for a downloadable PDF link.

Note that no report worker configuration was needed in this case as we know the exact parameters (i.e. the order number) and we can retrieve the specific order using customerOrderService.findByReference() without the need for a generic report worker.