

## Java Generic Amazon Flexible Payment System Component 1.0 Component Specification

### 1. Design

The purpose of this component is to abstract the complexity of using Amazon FPS, and to allow for its easy integration into web applications.

This component can be used when some amount should be charged immediately after performing the client authorization. Optionally future charges can be authorized and later performed with this component. Also it supports payment reservation followed by settling or cancellation operation; similarly completed payments can be refunded with this component.

The following terms are used in this specification:

- (Consuming) web application – the web application that uses this component programmatically to perform the charges;
- Client – the user of the consuming web application that needs to be charged using Amazon FPS;
- External system – the external application that needs to receive notifications about payment events.

This component consists of the following parts:

- AmazonPaymentManager interface – this is the main interface of this component; it is expected to be used by web application that initiates payment authorization and triggers charges. [Additionally it defines methods for retrieving details about all previously performed authorizations and payments, and getting the payments associated with specific authorization.](#)
- AmazonPaymentManagerImpl class – the only implementation of AmazonPaymentManager provided by this component.
- AuthorizationPersistence and PaymentPersistence interfaces – these interfaces are used by AmazonPaymentManagerImpl for storing required authorization and payment details in persistence.
- DatabaseAuthorizationPersistence and DatabasePaymentPersistence – the only implementations of AuthorizationPersistence and PaymentPersistence interfaces provided in this component; they use JDBC to access data in a database.
- PaymentEventSubscriber – this interface represents callback objects used for notifying about payment events.
- JMSAmazonPaymentEventSubscriber and JMSAmazonPaymentEventReceiver – these classes are expected to be used when payment events need to be transferred to another system; they use JMS and configured queue for this.
- PaymentAuthorizationData, PaymentAuthorizationRequest, PaymentEvent and PaymentDetails entities – these entities are used for communication between this component, calling web application and remote payment event handlers.
- Authorization, Payment and PaymentOperation entities – these entities are internal, they are used for communication between AmazonPaymentManagerImpl and persistence implementations.

#### 1.1 Design Patterns

**Strategy pattern** – AmazonPaymentManager together with its implementation can be used in some external strategy context; AmazonPaymentManagerImpl uses pluggable AuthorizationPersistence, PaymentPersistence and PaymentEventSubscriber implementation instances.

**Proxy pattern** – this component accesses Amazon FPS service via AmazonFPSClient proxy.



**DAO/DTO pattern** – [AmazonPaymentManager](#) AuthorizationPersistence and PaymentPersistence are DAOs for Authorization and Payment DTOs; PaymentEvent is a DTO used by JMSAmazonPaymentEventSubscriber and JMSAmazonPaymentEventReceiver for transferring payment event data to another system.

**Observer pattern** – AmazonPaymentManagerImpl holds a list of registered PaymentEventSubscriber instances that are notified about payment events.

**Delegate pattern** – methods [getAllAuthorizations\(\)](#), [getAllPayments\(\)](#) and [getPaymentsByAuthorization\(\)](#) of AmazonPaymentManagerImpl simply delegate execution to the namesake methods of AuthorizationPersistence and PaymentPersistence.

## 1.2 Industry Standards

JDBC, JavaBeans, JNDI, JAXB, JMS, XML

## 1.3 Required Algorithms

### 1.3.1 Logging

This component must perform logging in all public business methods of AmazonPaymentManagerImpl, DatabaseAuthorizationPersistence, DatabasePaymentPersistence, JMSAmazonPaymentEventSubscriber and JMSAmazonPaymentEventReceiver.

All information described below must be logged using log:Log attribute or logger retrieved with getLog() method. If log value is null, then logging is not required to be performed.

In all mentioned methods method entrance with input arguments, method exit with return value and call duration time must be logged at DEBUG level. It's not required to log method exit when method throws an exception.

When logging entities defined in com.topcoder.payments.amazonfps.model, values of all their properties must be logged recursively. It's allowed to override toString() method of the entities to achieve this.

All thrown exceptions and errors must be logged at ERROR level.

Warnings obtained from Co-Branded service must be logged at WARN level in AmazonPaymentManagerImpl#handleRequestFromCoBrandedService() method.

All payment events produced by AmazonPaymentManagerImpl must be logged at INFO level.

### 1.3.2 DB Schema

This section defines a DB schema for Informix database used by DatabaseAuthorizationPersistence and DatabasePaymentPersistence classes.

```
CREATE TABLE authorizations (
    id DECIMAL(19,0) NOT NULL ,
    multiple_use DECIMAL(1,0) NOT NULL,
    token_id VARCHAR(65),
    amount_left DECIMAL(10,2) NOT NULL,
    fixed_amount DECIMAL(10,2),
    cancelled DECIMAL(1,0) NOT NULL,
    create_date DATETIME YEAR TO FRACTION DEFAULT CURRENT YEAR TO FRACTION
);
```

```
CREATE TABLE payments (
    id DECIMAL(19,0) NOT NULL ,
    authorization_id DECIMAL(19,0) NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    transaction_id VARCHAR(35),
    status VARCHAR(13) NOT NULL,
    create_date DATETIME YEAR TO FRACTION DEFAULT CURRENT YEAR TO FRACTION
);
```

```
CREATE TABLE payment_parameters (
    payment_id DECIMAL(19,0) NOT NULL,
```



```
key VARCHAR(50) NOT NULL,
value VARCHAR(500),
create_date DATETIME YEAR TO FRACTION DEFAULT CURRENT YEAR TO FRACTION
);

CREATE TABLE payment_operations (
  id DECIMAL(19,0) NOT NULL,
  payment_id DECIMAL(19,0) NOT NULL,
  request_id VARCHAR(64),
  type VARCHAR(7) NOT NULL,
  success DECIMAL(1,0) NOT NULL,
  create_date DATETIME YEAR TO FRACTION DEFAULT CURRENT YEAR TO FRACTION
);

ALTER TABLE authorizations ADD CONSTRAINT
  PRIMARY KEY (id)
  CONSTRAINT pk_authorizations;

ALTER TABLE payments ADD CONSTRAINT
  PRIMARY KEY (id)
  CONSTRAINT pk_payments;

ALTER TABLE payment_parameters ADD CONSTRAINT
  PRIMARY KEY (payment_id, key)
  CONSTRAINT pk_payment_parameters;

ALTER TABLE payment_operations ADD CONSTRAINT
  PRIMARY KEY (id)
  CONSTRAINT pk_payment_operations;

ALTER TABLE payments ADD CONSTRAINT
  FOREIGN KEY (authorization_id)
  REFERENCES authorizations (id)
  CONSTRAINT fk_payment_authorization;

ALTER TABLE payment_parameters ADD CONSTRAINT
  FOREIGN KEY (payment_id)
  REFERENCES payments (id)
  CONSTRAINT fk_payment_parameter_payment;

ALTER TABLE payment_operations ADD CONSTRAINT
  FOREIGN KEY (payment_id)
  REFERENCES payments (id)
  CONSTRAINT fk_payment_operation_payment;
```

Table setup for ID Generator:

```
CREATE TABLE id_sequences (
  name VARCHAR(254),
  next_block_start DECIMAL(12,0) not null,
  block_size DECIMAL(10,0) not null,
  exhausted DECIMAL(1,0) default 0 not null
);

INSERT INTO id_sequences (name, next_block_start, block_size)
VALUES ('authorization', 1, 20);
INSERT INTO id_sequences (name, next_block_start, block_size)
VALUES ('payment', 1, 20);
INSERT INTO id_sequences (name, next_block_start, block_size)
VALUES ('paymentOperation', 1, 20);
```

### 1.3.3 *Integration to consuming web application*

This section describes how the consuming web application is expected to use this component.

This component can work with web applications based on custom servlets and with web applications that use any web framework (Spring MVC, Struts, etc).



The web application should hold an instance of `AmazonPaymentManagerImpl` and use it as described below.

- All payment authorizations are expected to be started by the client. I.e. the web application is expected to start authorization when the client presses some button or follows some link.
- When this happens, the application must call `initiatePaymentAuthorization()` method.
  - The returned `authorizationUrl` should be used by the web application for forwarding the client to the Amazon authorization page.
  - The returned `authorizationId` must be saved by the web application in case if future payments are expected to be performed without re-authorizing the client.
  - The returned `paymentId` must be saved by the web application in case if payment is reserved and needs to be settled/cancelled in future, or payment is expected to be refunded in future.
- The web application should define a page to which the client is redirected after completing the authorization on the Amazon CBUI site.
  - The URL of this page must be specified in `PaymentAuthorizationRequest#redirectUrl` property used on the previous step. This URL must not contain the following query parameters: `authorizationId`, `paymentId`, `reserve`, `errorMessage`, `expiry`, `status`, `tokenId`, `warningCode`, `warningMessage` (they are reserved by CBUI service and this component).
  - Before rendering this page the web application must call `handleRequestFromCoBrandedService()` method by passing all request parameters to it. This method saves authorization details and performs the payment/reservation in case if authorization was successful.
- In future to perform the payment/reservation without authorization of the client the web application can use `processAuthorizedPayment()` method. In case if `PaymentAuthorizationRequest#futureChargesFixedAmount` was specified when calling `initiatePaymentAuthorization()`, `PaymentDetails#amount` passed to `processAuthorizedPayment()` must be equal to the previously specified fixed amount.
- To settle the previously reserved payment the web application can call `settlePayment()` method.
- To cancel the previously reserved payment the web application can call `cancelPayment()` method.
- To refund the previously completed payment the web application can call `refundPayment()` method.
- To cancel the multiple-use authorization the web application can call `cancelAuthorization()` method.
- To get information about the previously performed authorizations and executions the application can call `getAllAuthorizations()`, `getAllPayments()` and `getPaymentsByAuthorization()` methods of the manager.
- The web application should handle critical internal errors by catching `AmazonFlexiblePaymentManagementException` thrown by `AmazonPaymentManagerImpl` methods.
- To receive information about all external errors (like payment rejection) and successful payment events the web application must register a custom `PaymentEventSubscriber` implementation instance within `AmazonPaymentManagerImpl` using `registerPaymentEventSubscriber()` method.
- The web application can associate a map of strings with each payment/reservation. This map can contain client ID, payment reason and other payment details that can be useful for payment event handlers.

## 1.3.4 *Notifying external systems about payment events*

This section describes how an external system can be notified about payment events generated by this component.

To transfer the payment events between applications the consuming web application must register a `JMSAmazonPaymentEventSubscriber` instance within `AmazonPaymentManagerImpl` using `registerPaymentEventSubscriber()` method.

The external system must instantiate `JMSAmazonPaymentEventReceiver` and periodically call `receivePaymentEvents()` method to get all the available payment events.

Payment events are transferred with use of queues and JMS framework.

## 1.4 **Component Class Overview**

### **AmazonPaymentManager [interface]**

This interface represents an Amazon payment manager. It defines methods that make usage of Amazon Co-Branded API easier when performing single-use or multiple-use payment authorization. Also it defines methods for performing, reserving, cancelling, settling and refunding payments with use of Amazon FPS Service after the authorization is performed. [Also the manager defines methods for retrieving details about all previously performed authorizations and payments, and getting the payments associated with specific authorization.](#)

### **AmazonPaymentManagerImpl**

This class is an implementation of `AmazonPaymentManager` that uses pluggable persistence implementations to store authorization and payment data. It uses Amazon FPS Java Library for accessing Amazon services. This class uses Logging Wrapper component to log errors and debug information. Note that this class throws exceptions specific to persistence and illegal usage, all other errors specific to authorization and payment rejection/failure are reported via the corresponding payment events.

### **Authorization**

This class is a container for information about a single authorization stored in persistence. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

### **AuthorizationPersistence [interface]**

This interface represents an authorization persistence. It defines methods for creating, updating and retrieving authorizations in/from persistence.

### **BaseDatabasePersistence [abstract]**

This is a base class for database persistence implementations provided in this component. It simply holds `DBConnectionFactory` and connection name specified within the configuration and provides a method for establishing database connections. Additionally it holds a Logging Wrapper logger instance if logging is required according to the configuration.

### **ConfigurablePaymentEventSubscriber [interface]**

This interface represents a configurable payment event subscriber. It simply extends `PaymentEventSubscriber` and defines `configure()` method that should be used for configuring subscriber instances with use of Configuration API component.

### **DatabaseAuthorizationPersistence**

This class is an implementation of `AuthorizationPersistence` that manages authorization data in database persistence using JDBC and DB Connection Factory component. This class uses Logging Wrapper component to log errors and debug information.

### **DatabasePaymentPersistence**

This class is an implementation of `PaymentPersistence` that manages payment and payment operation data in database persistence using JDBC and DB Connection Factory component. This class uses Logging Wrapper component to log errors and debug information.

### **JMSAmazonPaymentEventReceiver**



This class serves as a receiver of payment events sent via a queue with use of JMS. This class should be used in conjunction with JMSAmazonPaymentEventSubscriber when payment is initiated by one system, and payment event should be handled by another system.

#### **JMSAmazonPaymentEventSubscriber**

This class is an implementation of ConfigurablePaymentEventSubscriber that simply sends all obtained payment events to a configured queue with use of JMS. This class uses Logging Wrapper component to log errors and debug information.

#### **MapAdapter**

This class is subclass of XmlAdapter that supports marshalling and unmarshalling of Map<String, String> properties with use of MapEntry[] **value** type. In this component such adapter is required to support serialization and deserialization of PaymentDetails#parameters property.

#### **MapEntry**

This class represents a single entry of a string map. It is used as a **value** type by MapAdapter.

#### **Payment**

This class is a container for information about a single payment stored in persistence. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentAuthorizationData**

This class is a container for information that can be used by the user of this component for performing the payment authorization and initiating future payments. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentAuthorizationRequest**

This class is a container for information about a single payment authorization request. It holds information provided by the consuming application. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentDetails**

This class is a container for information about a single payment or payment reservation that is expected to be provided by the user of this component. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentEvent**

This class is a container for information about a single payment event that is used when notifying the registered subscribers. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentEventSubscriber [interface]**

This interface represents a subscriber that is expected to listen to and handle payment events.

#### **PaymentEventType [enum]**

This is an enumeration for payment event types.

#### **PaymentOperation**

This class is a container for information about a single payment operation stored in persistence. Its main purpose is to store request IDs generated by Amazon FPS Service for future troubleshooting purposes. It is a simple JavaBean (POJO) that provides getters and setters for all private attributes and performs no argument validation in the setters.

#### **PaymentOperationType [enum]**



This is an enumeration for payment operation types.

**PaymentPersistence [interface]**

This interface represents a payment persistence. It defines methods for creating, updating and retrieving payments in/from persistence. Additionally it defines a method for auditing payment operations.

**PaymentStatus [enum]**

This is an enumeration for payment statuses.

## 1.5 Component Exception Definitions

**AmazonFlexiblePaymentManagementException**

This exception is thrown by implementations of AmazonPaymentManager when some critical error that is not expected to be handled via payment event occurs. Also this exception is used as a base class for other specific custom exceptions.

**AmazonFlexiblePaymentSystemComponentConfigurationException**

This exception is thrown by classes and implementations of interfaces defined in this component when some configuration specific error occurs while initializing a class instance.

**AmazonPaymentEventReceivingException**

This exception is thrown by JMSAmazonPaymentEventReceiver when some error occurs while receiving payment events from the queue.

**AuthorizationNotFoundException**

This exception is thrown by implementations of AmazonPaymentManager and AuthorizationPersistence when authorization with the specified ID doesn't exist.

**AuthorizationPersistenceException**

This exception is thrown by implementations of AuthorizationPersistence when some error occurs while accessing the persistence.

**PaymentNotFoundException**

This exception is thrown by implementations of PaymentPersistence and AmazonPaymentManager when payment with the specified ID doesn't exist.

**PaymentPersistenceException**

This exception is thrown by implementations of PaymentPersistence when some error occurs while accessing the persistence.

## 1.6 Thread Safety

This component is thread safe when used correctly.

Implementations of AmazonPaymentManager and PaymentEventSubscriber are required to be thread safe when entities passed to them are used by the caller in thread safe manner.

Implementations of PaymentPersistence, AuthorizationPersistence and ConfigurablePaymentEventSubscriber are required to be thread safe when configure() method is called just once right after instantiation and entities passed to them are used by the caller in thread safe manner.

BaseDatabasePersistence is mutable, but thread safe when configure() method is called just once right after instantiation.

AmazonPaymentManagerImpl holds a mutable subscribers collection. Thus all access to the content of this collection is synchronized. AmazonPaymentManagerImpl is thread safe when collections and entities passed to it are used by the caller in thread safe manner.

DatabaseAuthorizationPersistence and DatabasePaymentPersistence are mutable, but thread safe when configure() method is called just once right after instantiation and entities passed to them are used by the caller in thread safe manner. They use thread safe DBConnectionFactory and Log instances.





JMSAmazonPaymentEventSubscriber is mutable, but thread safe when configure() method is called just once right after instantiation and entities passed to it are used by the caller in thread safe manner.

JMSAmazonPaymentEventReceiver is immutable and thread safe. But note that in most cases it doesn't make sense to perform multiple read operations on the same queue from multiple threads.

MapAdapter is thread safe when collections passed to it are used by the caller in thread safe manner.

PaymentDetails, PaymentAuthorizationData, PaymentAuthorizationRequest, PaymentEvent, Authorization, Payment, PaymentOperation and MapEntry are mutable and not thread safe entities.

PaymentStatus, PaymentEventType and PaymentOperationType are immutable and thread safe enumerations.

Transactions are used by all persistence methods that execute multiple DML statements.

## 2. Environment Requirements

### 2.1 Environment

Development language: Java 1.6

Compile target: Java 1.6, J2EE 1.5

QA Environment: JBoss, Informix 10, Informix 11

### 2.2 TopCoder Software Components

**Base Exception 2.0** – is used by custom exceptions defined in this component.

**Configuration API 1.1** – is used for configuring this component.

**Configuration Persistence 1.0.2** – is used for reading configuration from file.

**DB Connection Factory 1.1** – is used for creating database connections.

**ID Generator 3.1** – defines IDGenerator interface and IDGeneratorFactory class used in this component for generating entity IDs.

**JNDI Context Utility 2.0** – defines JNDIUtils class used in this component.

**Logging Wrapper 2.0** – is used for logging errors and debug information.

**Object Factory 2.2** – is used for creating pluggable object instances.

**Object Factory Configuration API Plugin 1.1** – is used for creating predefined objects defined with use of Configuration API component.

**TopCoder Commons Utility 1.0** – defines ParameterCheckUtility, ValidationUtility, LoggingWrapperUtility and JDBCUtility used in this component.

Also there is runtime (but not compile time) dependency on the following TCS components:

**Object Formatter 1.0.0, Typesafe Enum 1.1.0, Configuration Manager 2.1.5**

*NOTE: The default location for TopCoder Software component jars is `../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION` relative to the component installation. Setting the `tcs_libdir` property in `topcoder_global.properties` will overwrite this default location.*

### 2.3 Third Party Components

**Amazon FPS Java Library (API version – 2010-08-28)** (<http://aws.amazon.com/code/Amazon-FPS/5090796688019801>)

*NOTE: The default location for 3<sup>rd</sup> party packages is `../lib` relative to this component installation.*





Setting the `ext_libdir` property in `topcoder_global.properties` will overwrite this default location.

### 3. Installation and Configuration

#### 3.1 Package Name

com.topcoder.payments.amazonfps  
com.topcoder.payments.amazonfps.model  
com.topcoder.payments.amazonfps.persistence  
com.topcoder.payments.amazonfps.persistence.db  
com.topcoder.payments.amazonfps.subscribers  
com.topcoder.payments.amazonfps.subscribers.jms

#### 3.2 Configuration Parameters

##### 3.2.1 Configuration of AmazonPaymentManagerImpl

The following table describes the structure of ConfigurationObject passed to the constructor of AmazonPaymentManagerImpl class (angle brackets are used for identifying child configuration objects). This ConfigurationObject can be optionally read from a configuration file using Configuration Persistence component.

Parameter	Description	Values
loggerName	The name of Logging Wrapper logger to be used for logging errors and debug information. When not provided, logging is not performed.	String. Not empty. Optional.
<objectFactoryConfig>	This section contains configuration of Object Factory used by this class for creating pluggable object instances.	ConfigurationObject. Required.
authorizationPersistenceKey	The Object Factory key that is used for creating an instance of AuthorizationPersistence to be used by this manager.	String. Not empty. Required.
<authorizationPersistenceConfig>	The configuration for AuthorizationPersistence instance.	ConfigurationObject. Required.
paymentPersistenceKey	The Object Factory key that is used for creating an instance of PaymentPersistence to be used by this manager.	String. Not empty. Required.
<paymentPersistenceConfig>	The configuration for PaymentPersistence instance.	ConfigurationObject. Required.
<paymentEventSubscriberX>, where X can be any substring	The configuration for the registered payment event subscriber.	ConfigurationObject. Multiple. Optional.
<paymentEventSubscriberX>.subscriberKey	The Object Factory key that is used for creating an instance of ConfigurablePaymentEventSubscriber to be used by this manager.	String. Not empty. Required.
<paymentEventSubscriberX>.<subscriberConfig>	The configuration for ConfigurablePaymentEventSubscriber instance.	ConfigurationObject. Required.
awsAccessKey	The access key used when accessing Amazon services.	String. Not empty. Required.
awsSecretKey	The secret key used for signing the requests to Amazon services.	String. Not empty. Required.

paymentMethods	The supported payment methods. Should be a comma-separated list of any of the following values: CC, ACH, ABT.	String. Not empty. Required.
----------------	---	------------------------------

### 3.2.2 Configuration of BaseDatabasePersistence subclasses

The following table describes the structure of ConfigurationObject passed to the configure() method of BaseDatabasePersistence subclasses (angle brackets are used for identifying child configuration objects).

Parameter	Description	Values
loggerName	The name of Logging Wrapper logger to be used for logging errors and debug information. When not provided, logging is not performed.	String. Not empty. Optional.
dbConnectionFactoryConfig	The configuration to be used for creating DBConnectionFactoryImpl instance.	ConfigurationObject. Required.
connectionName	The connection name to be passed to the connection factory when establishing a database connection. If not specified, a default connection is used.	String. Not empty. Optional.

### 3.2.3 Configuration of DatabaseAuthorizationPersistence

The following table describes the structure of ConfigurationObject passed to the configure() method of DatabaseAuthorizationPersistence (angle brackets are used for identifying child configuration objects).

Parameter	Description	Values
authorizationIdGeneratorName	The name of generator of authorization IDs to be used in persistence. This name is passed to IDGeneratorFactory to get IDGenerator instance.	String. Not empty. Required.

See additional configuration parameters in the section 3.2.2.

### 3.2.4 Configuration of DatabasePaymentPersistence

The following table describes the structure of ConfigurationObject passed to the configure() method of DatabasePaymentPersistence (angle brackets are used for identifying child configuration objects).

Parameter	Description	Values
paymentIdGeneratorName	The name of generator of payment IDs to be used in persistence. This name is passed to IDGeneratorFactory to get IDGenerator instance.	String. Not empty. Required.
paymentOperationIdGeneratorName	The name of generator of payment operation IDs to be used in persistence. This name is passed to IDGeneratorFactory to get IDGenerator instance.	String. Not empty. Required.

See additional configuration parameters in the section 3.2.2.

### 3.2.5 Configuration of JMSAmazonPaymentEventSubscriber and JMSAmazonPaymentEventReceiver

The following table describes the structure of ConfigurationObject passed to the configure() method of JMSAmazonPaymentEventSubscriber and the constructor of JMSAmazonPaymentEventReceiver (angle brackets are used for identifying child configuration objects).

Parameter	Description	Values
loggerName	The name of Logging Wrapper logger to be used for logging errors and debug information. When not provided, logging is not performed.	String. Not empty. Optional.
jmsConnectionFactoryName	The JNDI name of the JMS connection factory to be used by this class.	String. Not empty. Required.
queueName	The JNDI name of the queue to be used by this class.	String. Not empty. Required.

### 3.3 Dependencies Configuration

Please see docs of dependency components to configure them properly.

#### 3.3.1 Switching between Sandbox and Production modes

Switching between sandbox and production modes is performed in the configuration of Amazon FPS Java Library.

Please find config.properties files in the “src” folder of the library.

In the production mode this file must contain the following parameters:

```
AwsServiceEndPoint=https://fps.amazonaws.com  
CBUIServiceEndPoint=https://authorize.payments.amazon.com/cobranded-ui/actions/start
```

In the sandbox mode this file must contain the following parameters:

```
AwsServiceEndPoint=https://fps.sandbox.amazonaws.com  
CBUIServiceEndPoint=https://authorize.payments-sandbox.amazon.com/cobranded-ui/actions/start
```

Note that “AwsAccessKey” and “AwsSecretKey” parameters are not required to be specified in this file (they are used by sample classes only). Instead they must be specified in the configuration of AmazonPaymentManagerImpl.

## 4. Usage Notes

### 4.1 Required steps to test the component

- Extract the component distribution.
- Follow [Dependencies Configuration](#).
- **Setup informix database:**
  - a. **Run test\_files/db\_setup.sql to create the tables.**  
**test\_files/db\_drop.sql can be used to drop the tables.**
  - b. **The db user:**  
**User: informix, Password: 1nf0rm1x**
  - c. **There are a lot of configuration tests files in this component with database settings that should be updated to match local environment. The database name used in configuration files is 'test'. At the moment there are ~30 places in configuration files that should be updated.**
- **This component uses Apache ActiveMQ 5.5.1 messaging server as JMS/JNDI provider. It can be downloaded from <http://activemq.apache.org/activemq-551-release.html> . Unpack the distribution to the local folder.**

# [TOPCODER]

- Update `test_files/jndi.properties` to match your local environment (in most cases it's not necessary if ActiveMQ is running with default setting, i.e. uses port 61616 and runs on the local machine).
- Run Apache ActiveMQ 5.5.1 messaging server.
- Amazon multi-use tokenID setup: the `test_files/tokenid.properties` file defines Amazon multi-use tokenID used for testing. It should match `awsAccessKey` and `awsSecretKey` defined in configuration files from `test_files/test_configs_amazon_payment_manager` folder (and possible in other places). Provided with this submission values won't be very useful since they are related to developer's personal amazon account which is protected by password. To update this parameters the one should register at Amazon AWS services and get Amazon FPS Sandbox account.
- Tips how to get multi-use token ID for testing after Sandbox account is ready (it should be put in `test_files/tokenid.properties`):
  - a) the simplest way is to use `demo_web_app` demo application described in demo section and request authorization with 'Multiple-use authorization' option selected. But the demo itself is requires some additional configuration before usage (demo section explains how to do this).
  - b) maybe a bit dirty but workable way: set breakpoint in `test_initiatePaymentAuthorization_2` test from `AmazonPaymentManagerImplTest.java` after `PaymentAuthorizationData` initialization and retrieve tokenID from this data.
- Execute 'ant test' within the directory that the distribution was extracted to.

## 4.2 Required steps to use the component

Please see the demo.

## 4.3 Demo

### 4.3.1 Sample `AmazonPaymentManagerImpl` configuration file

```
<?xml version="1.0"?>
<CMConfig>
  <Config name="com.topcoder.payments.amazonfps.AmazonPaymentManagerImpl">
    <Property name="loggerName">
      <Value>myLogger</Value>
    </Property>
    <Property name="objectFactoryConfig">
      <!-- Put Object Factory configuration here -->
    </Property>
    <Property name="authorizationPersistenceKey">
      <Value>DatabaseAuthorizationPersistence</Value>
    </Property>
    <Property name="authorizationPersistenceConfig">
      <Property name="loggerName">
        <Value>myLogger</Value>
      </Property>
      <Property name="dbConnectionFactoryConfig">
        <!-- Put DB Connection Factory configuration here -->
      </Property>
      <Property name="connectionName">
        <Value>myConnection</Value>
      </Property>
      <Property name="authorizationIdGeneratorName">
        <Value>authorization</Value>
      </Property>
    </Property>
    <Property name="paymentPersistenceKey">
      <Value>DatabasePaymentPersistence</Value>
    </Property>
    <Property name="paymentPersistenceConfig">
      <Property name="loggerName">
        <Value>myLogger</Value>
      </Property>
      <Property name="dbConnectionFactoryConfig">
        <!-- Put DB Connection Factory configuration here -->
      </Property>
    </Property>
  </Config>
</CMConfig>
```

```

</Property>
<Property name="connectionName">
  <Value>myConnection</Value>
</Property>
<Property name="paymentIdGeneratorName">
  <Value>payment</Value>
</Property>
<Property name="paymentOperationIdGeneratorName">
  <Value>paymentOperation</Value>
</Property>
</Property>
<Property name="paymentEventSubscriber1">
  <Property name="subscriberKey">
    <Value>JMSAmazonPaymentEventSubscriber</Value>
  </Property>
  <Property name="subscriberConfig">
    <Property name="loggerName">
      <Value>myLogger</Value>
    </Property>
    <Property name="jmsConnectionFactoryName">
      <Value>myJmsConnFactory</Value>
    </Property>
    <Property name="queueName">
      <Value>paymentEventQueue</Value>
    </Property>
  </Property>
</Property>
<Property name="awsAccessKey">
  <Value>weoiw8example45ow7e</Value>
</Property>
<Property name="awsSecretKey">
  <Value>2re4kf098523r2oi340w098e1</Value>
</Property>
<Property name="paymentMethods">
  <Value>CC,ACH,ABT</Value>
</Property>
</Config>
</CMConfig>

```

## 4.3.2 Usage scenarios

This section shows how this component is expected to be used in the consuming web application.

Assume that the consuming web application is built using custom servlets. Exception handling code is omitted in this section for brevity.

The following scenarios are demonstrated below:

- Scenario 1. The client is immediately charged with specific amount (just once).
- Scenario 2. The client is immediately charged with specific amount. Later the client can be charged with fixed authorized amount.
- Scenario 3. The client is immediately charged with specific amount. Later the client can be charged with variable authorized amount.
- Scenario 4. The payment amount is reserved for the client immediately. Later the reservation can be settled or cancelled.

One of the following methods can be used for creating AmazonPaymentManagerImpl instance to be later used by servlets:

```

// Create an instance of AmazonPaymentManagerImpl using the default configuration file
AmazonPaymentManager amazonPaymentManager = new AmazonPaymentManagerImpl();

// Create an instance of AmazonPaymentManagerImpl using the custom configuration file
amazonPaymentManager = new AmazonPaymentManagerImpl("my_config.xml", "custom_config");

// Create an instance of AmazonPaymentManagerImpl using a custom configuration
ConfigurationObject configuration = ...
amazonPaymentManager = new AmazonPaymentManagerImpl(configuration);

```

The following servlet method can be used when the client clicks "Pay" button:

# [TOPCODER]

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // Set request parameters common for all scenarios
    PaymentAuthorizationRequest authRequest = new PaymentAuthorizationRequest();
    PaymentDetails paymentDetails = new PaymentDetails();
    authRequest.setPaymentDetails(paymentDetails);
    Map<String, String> parameters = new HashMap<String, String>();
    parameters.put("projectId", request.getAttribute("projectId"));
    parameters.put("clientId", request.getAttribute("clientId"));
    paymentDetails.setParameters(parameters);
    authRequest.setRedirectUrl("https://myserver.com/processAuthResult.do");
    paymentDetails.setAmount(request.getAttribute("amount"));
    // Scenario 1
    // Nothing else is required to be specified
    // Scenario 2
    // authRequest.setFutureChargesAuthorizationRequired(true);
    // authRequest.setFutureChargesFixedAmount(request.getAttribute("futureAmount"));
    // authRequest.setTotalChargesThreshold(request.getAttribute("threshold"));
    // Scenario 3
    // authRequest.setFutureChargesAuthorizationRequired(true);
    // authRequest.setTotalChargesThreshold(request.getAttribute("threshold"));
    // Scenario 4
    // paymentDetails.setReservation(true);

    PaymentAuthorizationData paymentAuthorizationData =
        amazonPaymentManager.initiatePaymentAuthorization(authRequest);
    long authorizationId = paymentAuthorizationData.getAuthorizationId();
    long paymentId = paymentAuthorizationData.getPaymentId();
    // Save paymentId and authorizationId, e.g. to project or contest
    ...
    String authorizationUrl = paymentAuthorizationData.getAuthorizationUrl();
    // Send the client to the authorization page
    response.sendRedirect(authorizationUrl);
}
```

To accept the response (that is actually sent as an HTTP **GET** request to the web application) from the Amazon Co-Branded service, the following servlet method can be used:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    Map<Object, Object> requestParams = request.getParameterMap();
    amazonPaymentManager.handleRequestFromCoBrandedService(requestParams);
    if (isLastPaymentEventError()) {
        response.sendRedirect("https://myserver.com/paymentSuccess.do");
    } else {
        response.sendRedirect("https://myserver.com/paymentFailure.do");
    }
}
```

Later according to the scenarios 2 and 3 the following code can be used for performing another payment using the same authorization:

```
// Get previously saved authorization ID
long authorizationId = ...

// Process another authorized payment
PaymentDetails paymentDetails = new PaymentDetails();
Map<String, String> parameters = new HashMap<String, String>();
parameters.put("projectId", projectId);
parameters.put("clientId", clientId);
paymentDetails.setParameters(parameters);
paymentDetails.setAmount(amount);
long paymentId =
    amazonPaymentManager.processAuthorizedPayment(authorizationId, paymentDetails);

...

// Refund the payment if required
amazonPaymentManager.refundPayment(paymentId);

// Cancel authorization when no future changes are expected
amazonPaymentManager.cancelAuthorization(authorizationId);
```



The following code can be used for settling or cancelling the reservation according to the scenario 4:

```
// Get previously saved payment ID
long paymentId = ...

if (cancel) {
    // Cancel the payment
    amazonPaymentManager.cancelPayment(paymentId);
} else {
    // Settle the payment
    amazonPaymentManager.settlePayment(paymentId);
}
```

The code provided below shows how handling of payment events can be performed:

```
private ThreadLocal<PaymentEventType> lastPaymentEventType =
    new ThreadLocal<PaymentEventType>();

private static class CustomPaymentEventSubscriber implements PaymentEventSubscriber {
    public void processPaymentEvent(PaymentEvent paymentEvent) {
        PaymentEventType eventType = paymentEvent.getEventType();
        lastPaymentEventType.set(eventType);
    }
}

private boolean isLastPaymentEventError() {
    PaymentEventType paymentEventType = lastPaymentEventType.get();
    return (paymentEventType == PaymentEventType.AUTHORIZATION_CANCELLATION_FAILURE)
        || (paymentEventType == PaymentEventType.AUTHORIZATION_FAILURE)
        || (paymentEventType == PaymentEventType.PAYMENT_CANCELLATION_FAILURE)
        || (paymentEventType == PaymentEventType.PAYMENT_FAILURE)
        || (paymentEventType == PaymentEventType.REFUND_FAILURE)
        || (paymentEventType == PaymentEventType.RESERVATION_FAILURE)
        || (paymentEventType == PaymentEventType.SETTLEMENT_FAILURE);
}

...

// Register the custom payment event handler
PaymentEventSubscriber subscriber = new CustomPaymentEventSubscriber();
amazonPaymentManager.registerPaymentEventSubscriber(subscriber);
```

The code provided below shows how to get information about previously performed authorizations and payments:

```
// Retrieve all authorizations from persistence
List<Authorization> authorizations = amazonPaymentManager.getAllAuthorizations();

// Retrieve payments for each authorization
for (Authorization authorization : authorizations) {
    long authorizationId = authorization.getId();
    List<Payment> payments =
        amazonPaymentManager.getPaymentsByAuthorization(authorizationId);
    ...
}

// Retrieve all payments directly
List<Payment> payments = amazonPaymentManager.getAllPayments();
```

#### 4.3.3 *Receiving and processing payment events on the external system*

The following code can be used on the external system for processing payment events sent via a queue with use of JMSAmazonPaymentEventSubscriber:

```
// Get configuration of JMSAmazonPaymentEventReceiver
// Note that it's the same as configuration of JMSAmazonPaymentEventSubscriber
ConfigurationObject configuration = ...

// Create an instance of JMSAmazonPaymentEventReceiver
JMSAmazonPaymentEventReceiver jmsAmazonPaymentEventReceiver =
    new JMSAmazonPaymentEventReceiver(configuration);
```



# [TOPCODER]

```
while (!terminated) {
    // Receive payment events
    // Wait at maximum 1 second if there are no events
    List<PaymentEvent> paymentEvents =
        jmsAmazonPaymentEventReceiver.receivePaymentEvents(1000);
    for (PaymentEvent paymentEvent : paymentEvents) {
        // Process the payment event
        ...
        // Assume that that paymentEvent is the first event
        // generated in scenario 1, then
        // paymentEvent.getEventType() must be PaymentEventType.AUTHORIZATION_SUCCESS
        // paymentEvent.getPaymentDetails().getParameters() must contain 2 key/value pairs:
        // paymentEvent.getPaymentDetails().getParameters(){key1} must be "projectId"
        // paymentEvent.getPaymentDetails().getParameters(){key2} must be "clientId"
        // paymentEvent.getPaymentDetails().isReservation() must be false
        // paymentEvent.getAuthorizationId() must be equal to authorizationId
        // paymentEvent.getPaymentId() must be equal to paymentId
        //
        // Assume that that paymentEvent is the second event
        // generated in scenario 1, then
        // paymentEvent.getEventType() must be PaymentEventType.PAYMENT_SUCCESS
        // paymentEvent.getPaymentDetails().getParameters() must contain 2 key/value pairs:
        // paymentEvent.getPaymentDetails().getParameters(){key1} must be "projectId"
        // paymentEvent.getPaymentDetails().getParameters(){key2} must be "clientId"
        // paymentEvent.getPaymentDetails().isReservation() must be false
        // paymentEvent.getAuthorizationId() must be equal to authorizationId
        // paymentEvent.getPaymentId() must be equal to paymentId
    }

    // Sleep for 1 minute
    Thread.sleep(60000);
}
```

## 4.3.4 Web application demo

This submission provides two runnable demos and one of them is a Web application demo. This demo allows to use most of the functionality defined by this component (but not all, since it's only a demo). It allows to do payment authorization request (both for payment and reservation), create payment for existed authorization (both payment and reservation) and also the user can perform cancel, settle and refund operations over payments. Parameters can be set via web UI. The demo is build using custom servlets and runs in Tomcat 6 (tomcat was chosen primarily because it is very lightweight, but the demo can be easily ported to JBoss. the thing that can be different is JNDI configuraiton). No validation is performed for user input.

It's a demo home page with all available options:

**initiatePaymentAuthorization zone**

☒ Single Payment ☐ Multiple Payments

☐ Reservation

Enter payment amount:

Enter threshold (for multiple payments only):

Request authorization and perform payment

**processAuthorizedPayment zone**

☐ Reservation

Enter payment amount:

Enter authorization id:  
 (required)

Perform authorized payment

**cancelPayment, settlePayment, refundPayment zone**

☒ Cancel payment ☐ Settle payment ☐ Refund payment

Enter payment id:  
 (required)

Perform operation

Amazon's authorization confirmation page:



AxA

[SIGN IN](#)

[SELECT PAYMENT METHOD](#)

[CONFIRM PAYMENT](#)



#### Payment Authorization

This payment authorization permits AxA to charge you subject to the limits specified below. Please review the following payment details and click **Confirm** to authorize these payment terms.

Payment Summary	
<b>Payment Method:</b> <a href="#">Change</a> Visa: ***- 0779	<b>Billing Address:</b> <a href="#">Change</a> artemalive 123 Main St, 822 Nxk1LP Apt. 2B - 47 PcrR Dallas, TX 73696 United States
<b>Pay:</b> AxA <b>Valid From:</b> December 16, 2011 <b>Valid Until:</b> December 16, 2012	
<b>Payment Limits:</b> <ul style="list-style-type: none"><li>You will not be charged more than \$200.00 for the entire duration of the authorization</li></ul>	

[Confirm](#)

After authorization and first payment we have the following page, which also provides paymentId, authorizationId and tokenID which is very useful for testing (we can use paymentId, and authorizationId to fill corresponding fields on the home page and tokenID can be used for updating toleknid.properties file):

**Congratulations! The authorization and payment operation is completed. Please, use test receiver to see additional payment events info**

**authorizationId =15186**

**paymentId =10087**

**tokenID =P1ATXZL6A1BMD BQG573CFZYUVM3IAZH5UJRHADKPNCG7PTIA8485FSFRFQFVCZEP**

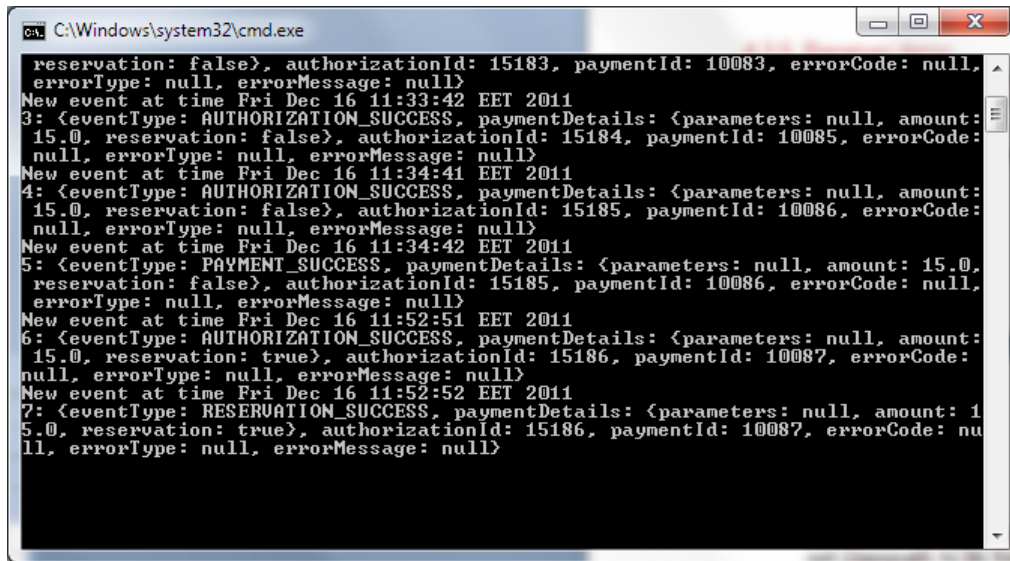
*NOTE: You can write down the authorization id and the payment id and use it later for testing*

*NOTE: Operation completion doesn't mean its success*

[Return to main zone](#)

#### 4.3.5. Receiver demo

The second runnable demo is a receiver demo. It's especially useful tool. Web demo shows only workflow of payment operations and provides some control data (ids) about this operations. But it doesn't say whether the operation was successful or not. Exactly for this purpose this small demo receiver tool was created. On windows platform it can be run by running [run.bat](#) file in [test\\_files/demo\\_receiver](#) folder. On other platforms the corresponding java class should be executed + set classpath to lib folder. Demo receiver receives all payment events and it can be very useful for debugging and better understanding of this component. Notice, that receiver runs while(true) loop so it should be closed manually.



```

C:\Windows\system32\cmd.exe
reservation: false}, authorizationId: 15183, paymentId: 10083, errorCode: null,
errorType: null, errorMessage: null}
New event at time Fri Dec 16 11:33:42 EET 2011
3: {eventType: AUTHORIZATION_SUCCESS, paymentDetails: {parameters: null, amount:
15.0, reservation: false}, authorizationId: 15184, paymentId: 10085, errorCode:
null, errorType: null, errorMessage: null}
New event at time Fri Dec 16 11:34:41 EET 2011
4: {eventType: AUTHORIZATION_SUCCESS, paymentDetails: {parameters: null, amount:
15.0, reservation: false}, authorizationId: 15185, paymentId: 10086, errorCode:
null, errorType: null, errorMessage: null}
New event at time Fri Dec 16 11:34:42 EET 2011
5: {eventType: PAYMENT_SUCCESS, paymentDetails: {parameters: null, amount: 15.0,
reservation: false}, authorizationId: 15185, paymentId: 10086, errorCode: null,
errorType: null, errorMessage: null}
New event at time Fri Dec 16 11:52:51 EET 2011
6: {eventType: AUTHORIZATION_SUCCESS, paymentDetails: {parameters: null, amount:
15.0, reservation: true}, authorizationId: 15186, paymentId: 10087, errorCode:
null, errorType: null, errorMessage: null}
New event at time Fri Dec 16 11:52:52 EET 2011
7: {eventType: RESERVATION_SUCCESS, paymentDetails: {parameters: null, amount: 1
5.0, reservation: true}, authorizationId: 15186, paymentId: 10087, errorCode: nu
ll, errorType: null, errorMessage: null}

```

#### 4.3.6 Demos configuration

1. Static IP is required for this demo (the demo will work only in one direction without static IP, i.e. it will send authorization requests but won't receive responses, since Amazon won't be able to send requests to local tomcat server).
2. HTTP port for tomcat server should be opened (again, in order to allow for amazon service communicate with local server).
3. Web app demo is located in [test\\_files/demo\\_web\\_app](#) folder. It should be copied to tomcat's [webapps](#) directory, so we will have the following path: \$(tomcatdir)/webapps/demo\_web\_app
4. Amazon should know which return URL to use when authorization is done. This return url should be specified in [webapp.properties](#) file which located in WEB-INF/classes. It should be the url with static IP of the machine plus name of the web application (and without trailing slash!).
5. Ensure that all database settings and amazon keys are set correctly for demo app.
6. The following files should be copied from ActiveMQ 5.5.1 distribution to tomcat's [lib](#) folder: [activemq-all-5.5.1.jar](#), [slf4j-log4j12-1.5.11.jar](#) and [log4j-1.2.14.jar](#)
7. tomcat's conf/server.xml should be updated in the following way: the following resource definitions should be added to [GlobalNamingResources](#) element (the resource names should be exactly as here, since they are referenced in other files):

```

<Resource name="myJMSConnectionFactory"
    auth="Container"
    type="org.apache.activemq.ActiveMQConnectionFactory"
    description="JMS Connection Factory"
    factory="org.apache.activemq.jndi.JNDIReferenceFactory"
    brokerURL="tcp://localhost:61616"
    brokerName="LocalActiveMQBroker"/>

<Resource name="paymentEventQueue"
    auth="Container"
    type="org.apache.activemq.command.ActiveMQQueue"

```



```
factory="org.apache.activemq.jndi.JNDIReferenceFactory"  
physicalName="myEventQueue"/>
```

8. tomcat's conf/ context.xml should be updated in the following way: the following elements should be added to `Context` element (the resource names should be exactly as here, since they are referenced in other files):

```
<ResourceLink global="myJMSConnectionFactory" name="myJMSConnectionFactory"  
type="javax.jms.ConnectionFactory"/>  
<ResourceLink global="paymentEventQueue" name="paymentEventQueue"  
type="javax.jms.Queue"/>
```

9. Ensure that ActiveMQ server is running.

10. Run tomcat (there should be no errors) and go to url similar to this:  
[http://localhost:8080/demo\\_web\\_app](http://localhost:8080/demo_web_app) (or use static IP instead of localhost, in any case amazon will redirect to url with static ip).

11. Use receiver tool to track payment events. NOTE: only one receiver should run at one moment of time. If more receivers are running they will take messages from each other and each will show random selected messages.

## 5. Future Enhancements

Reporting functionality can be added in future.