Internet Payment Gateway

Java API Developer Guide

st.george

Table of Contents

Disclaimer 3
Confidentiality 3
Document Purpose / Intended Audience 3
Related Documents 3
1 Introduction 4
1.1 Minimum System Requirements 4
2 Installation 5
2.1 Specialised or non-standard implementation details 5
3 Testing your Installation 6
3.1 St. George Bank Specific Information and Files 6
3.2 The Test Programs 6
3.2.1 Supplying input to the TestTransaction program 7
3.2.2 Supplying input to the BaseTest program 8
3.3 Analysis of the response 8
3.4 Troubleshooting 9
3.4.1 Installation 9
3.4.2 SSL/Security 9
3.4.3 Connectivity 9
3.4.4 System 10
3.4.5 Parameter values / Transaction data 10
3.4.6 Response codes 10
4 Java API Reference 11
4.1 Available Functions/Methods 11
4.2 Error / Exception Handling 16
5 Sample Application 17
5.1 Source Code 17
5.1.1 Purchase Example 17
5.1.2 Refund Example 20
5.1.3 Servlet Example 22
6 Getting Help 25
6.1 IPG Web Site 25
6.2 Contacting St. George Bank 25
Appendix A – Files included in this kit 26

Disclaimer

This document, including attachments, is not for general circulation, but for distribution to a limited number of specific corporations and individuals that are known to St.George Bank ("Company"). It is in draft form and its contents are subject to change, including changes of substance. Any person choosing to act on information contained in this document is advised to verify the information.

Confidentiality

By accepting this document, the recipient agrees to keep the information contained in it permanently confidential. This document is intended for use by the recipient only and may not be copied, reproduced or distributed to others at any time without the prior written consent of the Company. It cannot be used for any purpose except that for which it is given to you i.e. as user documentation.

If you do not agree with any of the above conditions, you must return this document immediately.

Document Purpose / Intended Audience

This guide is one component of the St. George Bank Transaction Server document set. It intended to act a reference point for developers seeking to implement the St. George Bank transaction functionality into merchant web applications. It provides specific technical information about the typical deployment of the St. George Bank Java API and delivers valuable insights into more user-specific deployments.

It is assumed that the parties implementing this API will be experienced in implementing web applications in a Java development environment, and this document is written principally toward parties with this expertise.

Related Documents

- **St. George Bank Generic API Developer Guide**
- **St. George Bank Win32 Developer Guide**
- **St. George Bank .Net Developer Guide**
- **St. George Bank Perl Developer Guide**
- **St. George Bank Linux Developer Guide**
- **St. George Bank PHP Developer Guide**

1 Introduction

The St. George Bank Java Client API is a collection of objects for use with the *St. George Internet Payment Gateway*. The API enables web site and application developers to safely and efficiently add credit card transaction processing capabilities to their products.

The St. George Bank Java API can be used with any operating system or platform that has an implementation of Sun's JDK 1.2.2 or greater. At time of writing JDK 1.4.2 is recommended as the standard platform for this implementation.

This document should be used in conjunction with the St. George Bank Generic API developer Guide, which includes details on parameters required for credit card transactions, implementation guidelines and a list of response codes.

This document also includes common configuration settings, test applications and protocols, and sample code that will assist developers in incorporating the API into their chosen application.

Communication between the St. George Bank API and St. George Bank Transaction Server Gateway uses SSL connectivity through server addresses and defined ports.

NOTE: If your application is located behind a firewall or proxy server, you must ensure that the relevant addresses and ports are not being filtered or blocked.

It is important to note that the St. George Bank Transaction Server utilises two distinct transaction processing environments. These environments are defined as **LIVE** and **TEST**. The **LIVE** environment connects directly to the live banking processing system and performs real-time transactions. The **TEST** environment is a simulated environment controlled within the transaction server that effectively mimics the live processing environment; **TEST** is principally used for integration testing and training.

Using the parameters provided by your St. George Bank you representative must define and test your application and API integration against the **TEST** environment to ensure compliance.

1.1 Minimum System Requirements

The following minimum system configuration is required for implementation of the St. George Java API.

Operating System	Windows 2000 & XP / RedHat Enterprise Linux 4 (kernel 2.6.9-22.0.1.EL) with glibc 2.3.4 **NOTE: Win95/98/ME not supported**
Processor	Pentium 4
Processor Speed	1.5Ghz
Memory	512Mb
Java JRE/JDK	1.4.2

Please note:

- The Java API has been certified for a machine built exactly as above and is not supported for any other configurations.
- It is assumed that the parties implementing this API will meet the above requirements and be experienced in implementing web applications in a Java development environment, and this document is written principally toward parties with this expertise.
- Memory requirements are dependant on the number of applications running on the machine and the specific application or application server that the API is integrated with.

2 Installation

Before the Webpay Java Client API may be used JDK 1.4.2 or greater must be installed and configured, if required you can download the JDK SDK to suit your environment from **java.sun.com**, or contact your platform vendor.

Installation is delivered as an **Installshield for Windows platforms** and a **tar.gz file for other platforms**. The install contains all the necessary components including sample source code, documentation, and supporting libraries.

The Installshield provides a simple point-and-click process that installs all files into the appropriately specified locations and performs the requisite registration of dependencies, classes and libraries.

The tar.gz file can simply be extracted (using the following command: **tar -xzvf webpayJava-1.11.tar.gz**) to a chosen location and the necessary files moved where appropriate.

2.1 Specialised or non-standard implementation details

The following section is provided for those parties who wish to tailor the installation to suit a specialised or non-standard implementation.

1. If performing a non-standard implementation ensure that the Webpay Java components can be found in the application CLASSPATH. Of particular note is the **WebpayClient.jar** file.
2. Systems running Pre JDK 1.4

   If your system is running a JDK version earlier than 1.4 you will have to download and install the following packages (NB: Both these products have been integrated into JDK 1.4).

   - Java Secure Sockets Extension (JSSE) and
   - Java Cryptography Extension (JCE) provider classes.

   Then check the registration of the JSSE security provider in the security properties file…

   `<java-home>\lib\security\`**`java.security`** `[Win32]`

   `<java-home>/lib/security/`**`java.security`** `[Solaris]`

   This file may be edited with a text editor and the following line must appear in the file. The line may be allocated to any position in the file.

   `security.provider.`*`n`*`=com.sun.net.ssl.internal.ssl.Provider`

   Where ***n*** is a number greater than 1 and defines each provider in order of preference. For more information, see **java.sun.com/products/jsse/install.html**, particularly ***Section 5 – "Static Registration of JSSE providers"***

   Please note that this security provider will also be used by other applications sharing the same JVM (such as Tomcat).

For more information on JSSE and JCE, visit **java.sun.com/products/jsse/** and **java.sun.com/products/jce/**.

3 Testing your Installation

3.1 St. George Bank Specific Information and Files

Your St. George Bank representative will have supplied you with information and files specific to the St. George Bank Gateway you will be using, including:

- St.George Bank Gateway address (www.gwipg.stgeorge.com.au). The Gateway offers multiple streams of servers for load balancing and failover. Please see Appendix B.
- Gateway port numbers: Test (3007) and Live (3006)

  NOTE: If your application is located behind a firewall or proxy server, you should ensure that the relevant addresses and ports are not being filtered or blocked
- Your St.George Bank Client ID
- St.George Bank client key store appropriate for the St.George Bank gateway and the password protecting this keystore file.

You must place the St. George Bank key store somewhere on your machine so that the St. George Bank client software can access it. It is recommended that you place the file in the same folder with all your other St. George Bank client files or with other files that make up your own application.

3.2 The Test Programs

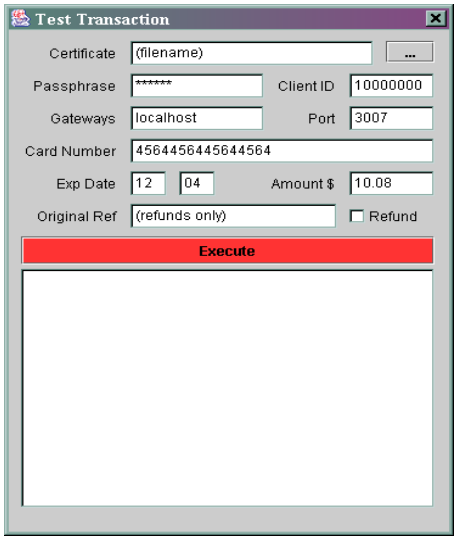
The St. George Java API comes with two test programs.

TestTransaction	TestTransaction is a GUI application that can be invoked by simply double clicking on the WebpayClient.jar file.
BaseTest	The BaseTest program should only be run if the GUI TestTransaction is not available.

You must run one of the about test programs to ensure that:

- The Sun JSSE jars are correctly installed (Pre JDK 1.4 installations only)
- The basic Webpay Client jar has been installed
- There are no other basic connectivity problems with your machine
- Are the necessary ports opened in your firewall to allow communications
  NOTE: If your application is located behind a firewall or proxy server, you must ensure that the relevant addresses and ports are not being filtered or blocked.

3.2.1 Supplying input to the TestTransaction program



Alternatively you can load the GUI by entering the following java command on the command line.

Java –jar WebpayClient.jar

The following table outlines the description of the required fields for the GUI Test Program.

Field	Description
Certificate	The name of your St. George Bank client certificate file, e.g. "/home/StGeorge/client.cert". You can use the full path or a relative path, as long as the name resolves correctly.
Passphrase	The password protecting your St. George Bank client keystore.
ClientId	The Client ID issued to you
Gateways	The address of your St. George Bank gateway. See Appendix B.
Port	The port number provided to you. Ensure that you specify the correct port number for either the **test (3007)** or **live (3006)** environment.
CardNumber	A credit card number. In the testing environment the card number **4111 1111 1111 1111**(without spaces) should prove valid. A list of valid **test** credit card numbers is contained in the Generic API Developer Guide. **NOTE: Do not use a real credit card number in the testing environment as it will not be recognised as valid.**
Exp Date	The test credit card number's expiry date. This can be any date in the future.
Amount	A whole dollar amount for this transaction including cents value but excluding currency symbols. For example $10.00 would be **10.00**. **NOTE:** The cents value specified controls the response received. Only a zero cents value will return an approved transaction. A list of response codes is contained in the Generic API Developer Guide.
Original Ref	(Used for Refunds only) This is the Original transaction reference number of the tranaction that is being refunded.
Refund	Check this box to perform a Refund transaction.

3.2.2 Supplying input to the BaseTest program

If you are unable to run the GUI Test Program, than you can run the Command Line test program instead.

- Make your working directory the BaseTest directory. **cd BaseTest**
- You can then simply run the BaseTest program straight from the command line

The Command Line test program requires the following inputs:

BaseTest <certificate file> <certificate password> <clientid> <cardnumber> <amount> <card expiry date > <servers> <port>

Field	Description
Certificate file	The name of your St. George Bank client certificate file, e.g. "/home/StGeorge/client.cert". You can use the full path or a relative path, as long as the name resolves correctly.
Certificate password	The password protecting your St. George Bank client keystore.
ClientId	The Client ID issued to you
Card number	A credit card number. In the testing environment the card number **4111 1111 1111 1111**(without spaces) should prove valid. A list of valid **test** credit card numbers is contained in the Generic API Developer Guide. **NOTE: Do not use a real credit card number in the testing environment as it will not be recognised as valid.**
Card Expiry Date	The test credit card number's expiry date. This can be any date in the future.
Amount	A whole dollar amount for this transaction including cents value but excluding currency symbols. For example $10.00 would be **10.00**. **NOTE:** The cents value specified controls the response received. Only a zero cents value will return an approved transaction. A list of response codes is contained in the Generic API Developer Guide.
Servers	The address of your St. George Bank gateway. See Appendix B.
Port	The port number provided to you. Ensure that you specify the correct port number for either the **test (3007)** or **live (3006)** environment.

Example:

Win32

```
java -cp .;..\WebpayClient.jar BaseTest c:\keystore password 10000000 4111111111111111 1.00 0505 192.0.0.1 3007
```

Linux

```
java -cp .:../WebpayClient.jar BaseTest c:\keystore password 10000000 4111111111111111 1.00 0505 192.0.0.1 3007
```

3.3 Analysis of the response

Execute one of the test programs.

NOTE: The test may take some time to establish the SSL connection.

The output of the test program will look similar to this…

- Response Code = "00"
- Response Text = "Approved (TEST TRANSACTION ONLY)"
- Error Message = "(null)"
- Transaction Reference = 0301000000123469

If the output of the test program contains the word "APPROVED", the installation of the API has been successful.

The following gives a brief overview of the fields involved:

Field	Explanation
RESPONSECODE	A two digit code used to determine the outcome of the transaction. The code number matches the RESPONSETEXT field. "**00**", "**08**" or "**77**" designate an **approved** transaction. A complete list of response can be found in Appendix A of the Generic API Developer Guide.
RESPONSETEXT	A meaningful text message that explains the response code
ERROR	Any error additional messages received from the gateway.
TXNREFERENCE	A unique reference number generated for each transaction.

NOTE: Please follow the guidelines in the Generic API Developer Guide on how to determine the outcome of a transaction.

3.4 Troubleshooting

If your output does not look as expected, please review the steps you have taken so far. The following questions may assist in determining common API problems:

3.4.1 Installation

- Are the SSL jar files somewhere on the classpath?
- Is the webpayclient.jar somewhere on the classpath?
- Are the security providers appropriately designated in the java.security file supplied with the JDK and JRE? You may need to edit the java.security file and check that the sun.security provider is assigned first in the list of providers. This file lists all available security providers and each may be numbered in order of priority. Changing the order of providers may alleviate any connection problems. For more information on security providers, visit http://java.sun.com.

3.4.2 SSL/Security

If you encounter the error "UNABLE TO INITIALISE SSL" or similar, please check the following:

- Do you have the correct keystore?
- Are you using the correct keystore passphrase?
- Path to certificate file: Have you specified the full path, eg. \home\stg\java.cert
- Trusted certificate file: leave this parameter blank
- Do you have read access to the keystore?

3.4.3 Connectivity

- Do you have connectivity to the gateway?
    - Windows: Launch Internet Explorer on the machine the API is installed on and go to http://www.gwipg.stgeorge.com.au 3006. If you can see the 5 squares, you can reach the gateway.
    - Linux/Unix: Try the command "telnet www.gwipg.stgeorge.com.au 3006". If it lets you connect, you can talk to the gateway. See Appendix B for more information.
- Are you behind a firewall? You may need to contact your Security Administrator to ensure the API ports are open for traffic in both directions.
- Are you using any packet shapers or bandwidth management products? If not configured properly, these might affect connectivity to the gateway.
- Commercial and freeware tools (eg. Packet Sniffers) can be found on the Internet, to assist you in testing your network connectivity to the St. George Bank the Secure Transaction Servers.

3.4.4 System

- Are you using the latest drivers for you network interface card?
- Are your TCP/IP libraries up to date?
- Have you applied the latest patches to your Operating System?
- Do you have enough RAM on your machine to accommodate for concurrency and/or other applications running on that machine?
- Do you have any applications running that might interfere with the API?

3.4.5 Parameter values / Transaction data

- Is the right location of the St. George Bank keystore file specified? You may need to specify the file's full path.
- Is the password correct for the St. George Bank client keystore?
- Is the correct address and test port number specified for the St. George Bank gateway? You may want to use "nslookup" to ensure that the address resolves correctly. If it does resolve correctly, the IP address of the gateway will be displayed on the screen. If it does not, you will get an "unknown host" message.
- Are you using the correct Client ID?
- Are you sending all the required fields for a transaction? Refer to the generic developer guide for required fields.
- Do all the fields in the transaction bundle contain valid data? Refer to the generic developer guide for field restrictions.

3.4.6 Response codes

- In some cases, the system may return errors when you are expecting an approved or other code during testing. Please check the response code you receive against those contained in the Generic API Developer Guide.
- In the Test environment, the response code is determined by the cents amount, eg. $1.51 = "51 INSUFFICIENT FUNDS"
- Some response codes may be generated at any time and this behaviour is perfectly normal, for example, the system may return the code 'IP' for 'In progress' if the transaction is still being processed.

If you are still unable to identify the problem, call your St. George Bank representative. Refer to chapter 6, "Getting Help" for contact details.

4 Java API Reference

This section contains reference material, functions, methods and source code used to implement the Java API

For information on Credit Card transactions and parameters refer to the *Generic API Developer Guide.*

4.1 Available Functions/Methods

This section contains the JavaDoc defined by the St. George Bank Class and includes available methods and member variables, class hierarchy, information on exceptions, and other details.

webpay.client

Class Webpay

java.lang.Object

|

+--**webpay.client.Webpay**

---

public class **Webpay**

extends java.lang.Object

This object provides methods to perform transactions with the St. George Bank gateway array, or a specified gateway array. Transactions are implemented securely using SSL and a supplied digital certificate.

Constructor Summary
Webpay(java.lang.String clientID, java.lang.String certificatePath, java.lang.String certificatePassphrase) Constructor for the Webpay object.
Webpay(java.lang.String clientID, java.lang.String certificatePath, java.lang.String storeFormat, java.lang.String certificatePassphrase)

Method Summary	
Void	**completeTransaction**() This method is used to complete a transaction request that is currently in progress and has been initialised using the initTransaction method.
Void	**execute**() This method has the same effect as calls to initTransaction() and completeTransaction.
Java.lang.String	**get**(java.lang.String name) Returns a value from the response bundle.
webpay.net.TransactionBundle	**getBundle**() Returns the TransactionBundle from the Webpay client object
Java.lang.String[]	**getResponseNames**() Returns a String array of the names for values in the response bundle.
Java.lang.String	**initTransaction**() Initialises a transaction against the St. George Bank Transaction Servers.
Void	**put**(java.lang.String name, java.lang.String value) Put a value in the internal data bundle for transmission.

`Void`	**setDebugLevel** `(int level)` Sets the amount of logging to enable within the webpay object.
`Void`	**setPort** `(int port)` Sets and overrides the default St. George Bank server port.
`Void`	**setPort** `(java.lang.String port)` Sets and overrides the default St. George Bank server port.
`Void`	**setServers** `(java.lang.String[] servers)` Sets and overrides the default St. George Bank server array.
`Java.lang.String`	**toString** `()` Outputs the Transaction Bundle in the form of value pairs separated by a newline (crlf).

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Webpay

public **Webpay**(java.lang.String clientID,
java.lang.String certificatePath,
java.lang.String certificatePassphrase)
throws java.lang.Exception

Constructor for the Webpay object. Throws one of many exceptions relating to the SSL certificate or file system.

Parameters:

`clientID` - The client identification number.

`certificatePath` - The location on disk of the clients digital certificate or certificate keystore.

`certificatePassphrase` - The passphrase for the digital certificate or certificate keystore.

Throws:

`java.lang.Exception` - Throws an Exception if the object could not be set up correctly.

Webpay

public **Webpay**(java.lang.String clientID,
java.lang.String certificatePath,
java.lang.String storeFormat,
java.lang.String certificatePassphrase)
throws java.lang.Exception

Method Detail

execute

public void **execute**()

throws java.io.IOException

This method has the same effect as calls to initTransaction() and completeTransaction. The get() method must be used to obtain the results of the transaction after this method returns. Protocol Flow is as follows:

```
        |                      |
1       | ------------------>  |
        |                      |
2       | <------------------  |
        |                      |
3       | ------------------>  |
        |                      |
4       | <------------------  |
        |                      |
```

1. Send FULL request. 2. Receive Transaction Reference. 3. Send Transaction Reference back to server. 4. Receive FULL response.

If you do not receive a response(4) then you can do a status transaction against the server, to get the transaction status.

Throws:

`java.io.IOException` - if a communications error occurs

---

initTransaction

public java.lang.String **initTransaction**()

throws java.io.IOException

Initialises a transaction against the St. George Bank Transaction Servers. This method must be used in conjunction with completeTransaction. These methods are useful for logging transaction status mid way through the transaction.

Returns:
String containing the transaction reference.

Throws:
`java.io.IOException` - when a communications error occurs with the engine. It is safe to assume that the transaction has not been processed.

---

completeTransaction

public void **completeTransaction**()

throws java.io.IOException

This method is used to complete a transaction request that is currently in progress and has been initialised using the initTransaction method. If this method fails or throws an exception, it must be logged, and a status transaction should be performed to check the outcome. It must be assumed that the transaction was processed until it is confirmed otherwise.

Throws:

`java.io.IOException` - on a communications error

---

toString

public java.lang.String **toString**()

Outputs the Transaction Bundle in the form of value pairs separated by a newline (crlf).

Overrides:

`toString` in class `java.lang.Object`

Returns:

String containing the contents of the TransactionBundle

---

getBundle

public webpay.net.TransactionBundle **getBundle**()

Returns the TransactionBundle from the Webpay client object

Returns:

TransactionBundle

---

put

public void **put**(java.lang.String name, java.lang.String value)

Put a value in the internal data bundle for transmission. This bundle will be sent on a call to doTransaction(). This method can be used to replace the parametized transaction implementations. Transactions can be implemented with Name / Value combinations published by St. George Bank. Matching sets of values also exist for the response data. see `get()`.

Parameters:

`name` - The name of the value to be set

`value` - The transaction value to be set e.g. Credit card number

---

get

public java.lang.String **get**(java.lang.String name)

Returns a value from the response bundle.

Parameters:

`name` - The name of the value to be extracted from the response bundle

Returns:

The requested string value. May return `null` if the value has not been set

---

getResponseNames

public java.lang.String[] **getResponseNames**()

Returns a String array of the names for values in the response bundle. This allows you to determine all the values which have been returned by the gateway for the last transaction. If no values are available or no transaction has occurred, a String array of length zero is returned.

Returns:

A String array containing the names of the values that are available within the object.

setServers

public void **setServers**(java.lang.String[] servers)

Sets and overrides the default St. George Bank server array. This method should only be used if transactions are to be passed through a gateway other than the standard St. George Bank server array. This method must be called before any transaction attempts are made and should not be reset once transactions have begun through this instance of a Webpay object.

Parameters:

`servers` - A String array of domain names and/ or IP addresses.

setPort

public void **setPort**(java.lang.String port)

Sets and overrides the default St. George Bank server port. This method must be called before any transaction attempts are made and should not be reset once transactions have begun through this instance of a Webpay object. The port number determines whether the transaction is sent to the "**Live**" or "**Test**" environment.

Parameters:

`port` - port - the TCP port number to connect to

setPort

public void **setPort**(int port)

Sets and overrides the default St. George Bank server port. This method must be called before any transaction attempts are made and should not be reset once transactions have begun through this instance of a Webpay object. The port number determines whether the transaction is sent to the "**Live**" or "**Test**" environment.

Parameters:

`port` - the IP port number of the St. George Bank servers to connect to.

setDebugLevel

public void **setDebugLevel**(int level)

Sets the amount of logging enabled within the webpay object.

Parameters:

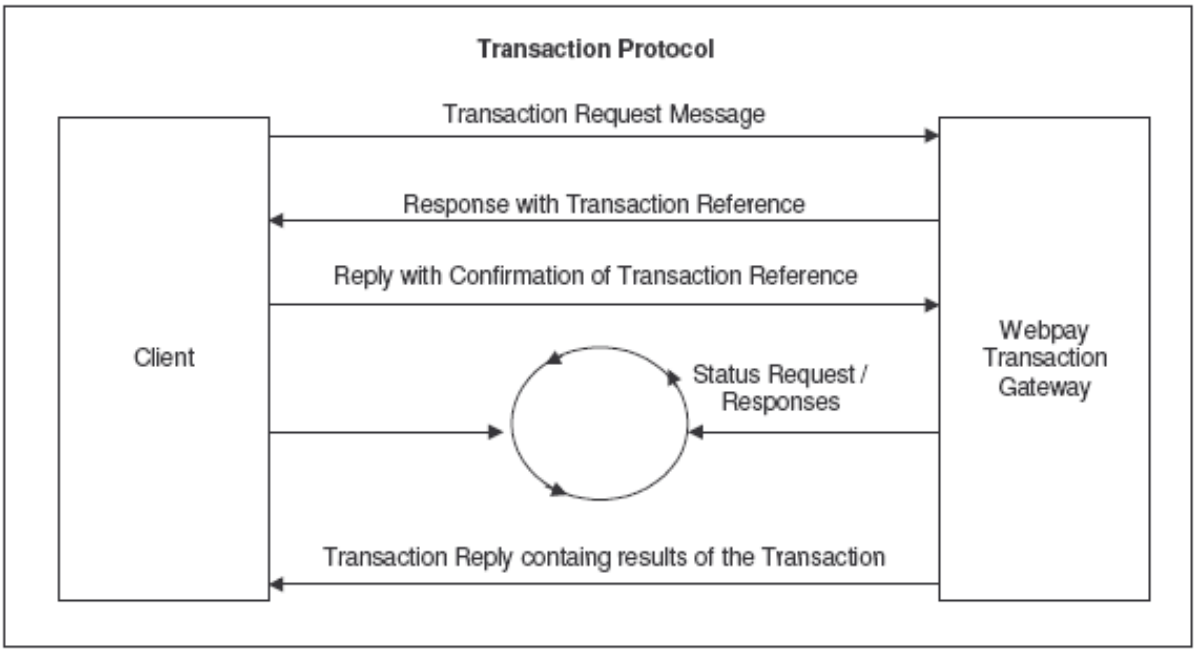
`level` - int specifying the amount of debugging. 0 = off, 1 = lowest, 3 = highest.

4.2 Error / Exception Handling

The St. George Bank Java API handles all errors internally. If any errors are encountered during processing, the transaction will fail, and error messages can be obtained by investigating the ERROR field.

Full details of all errors thrown are included in the JavaDoc that accompanies this guide.

If a transaction attempt fails and a Transaction Reference has been returned by the St. George Bank Servers, the execute method automatically queries the status up to three more times. Beyond this, execute fails, and the transaction reference is available to the application. The merchant's application should be also query the status of such a transaction, and be able to do so at any time.



5 Sample Application

5.1 Source Code

The code included here is the most basic sample implementation possible and contains hard-coded values that display the responses to the user. It is not intended as a definitive solution but only to demonstrate the basic concepts involved in using the St. George Bank Java API for creditcard transactions. Your production application will need to collect user or other input, and log the transaction responses to a database and/or display them to the appropriate parties.

Three code samples are included below: **Purchase Example**, **Refund Example**, and **Servlet Example**.

All sample code includes extensive comments, thereby negating the need for a code walkthrough.

5.1.1 Purchase Example

```
package webpay.client.Webpay;
import java.io.*;
import java.lang.*;
import java.util.*;
import java.security.*;
import webpay.client.*;

/**
* Class used for testing the behaviour of the Webpay java client (webpay.client.Webpay)
* Demonstrates the following:
*
* - Overriding default server list and port
* - Constructing the Webpay object
* - Building the transaction bundle
* - Executing the transaction
* - Polling on a status of "IN PROGRESS"
* - Catching exceptions
*/
public class WebpayExample {
       public static void main ( String[] args ) {
              String responseCode = null;
              try {

                     // Check that the necessary arguments are present. The last two are
              optional.
                     if ( args.length < 4 ) {
                            System.out.println( "Usage: WebpayExample [clientid] [cardnumber]
                     [value as XX.XX] [card expiry date as XXXX] ([servers]) ([port]) " );
                            System.exit( 1 );
                     }

                     // Construct the webpay object. The parameters are clientid, certpath,
cert_passphrase

                     Webpay webpay = null;
                     try {
                            webpay = new Webpay( args[0] , "webpay", "webpay" );
                     } catch (Exception wt_ex ) {
                            System.err.println( "Error creating the Webpay object" );
                            wt_ex.printStackTrace();
                     }

                     // If server list is present in argument list, override the default
              server

                     if ( args.length > 4 ) {
                            String[] servers = matrix.tools.StringTools.split( args[4], ',' );
                            webpay.setServers( servers );
                     }

                     // If port is present in argument list, override the default port
```

```
if ( args.length > 5 )
       webpay.setPort( args[5] );

// Put other arguments into Webpay object

webpay.put( "CARDEXPIRYDATE", args[3] );
webpay.put( "CARDDATA", args[1] );
webpay.put( "TOTALAMOUNT", args[2] );
webpay.put( "TRANSACTIONTYPE", "PURCHASE" );
webpay.put( "INTERFACE", "CREDITCARD" );
webpay.put( "TERMINALTYPE", "0" );

// Execute the transaction.

try {
       webpay.execute();
} catch ( IOException io_ex ) {

       // The transaction has failed at a protocol level.

       System.err.println( "Error executing transaction." );
       io_ex.printStackTrace();

       // If TXNREFERENCE has been set, then the transaction was
       initialised
       // before failing. Let the transaction continue to see if it
       brings a
       // result back.

       if ( webpay.get( "TXNREFERENCE" ) != null ) {
              responseCode = "IP";
       }
}

// No exception has occurred, so the transaction has returned a
// response code. Check to see if the transaction is still "IN PROGRESS".
// If it is, poll server with status requests until either the response
// code changes or to a maximum of three status requests.

if ( responseCode == null ) responseCode = webpay.get( "RESPONSECODE" );
for ( int statusCheckCount = 0; statusCheckCount < 3;
statusCheckCount++){
       if ( "IP".equals( responseCode )) {

              // The transaction is still in progress. Send a status
       request.

              webpay.put( "TRANSACTIONTYPE", "STATUS" );
                     try {
                            Thread.sleep( 500 ); // Half-second polling
                     delay
                            webpay.execute();
                     } catch ( IOException io_ex2 ) {

                            // The transaction has failed at a protocol
                     level.
                            System.err.println( "Error executing
                            transaction." );
                            io_ex2.printStackTrace();
                     }
                     responseCode = webpay.get( "RESPONSECODE" );
              } else {

                     // Break out of for loop, we don't need poll anymore
              as the transaction
                     // is no longer IN PROGRESS

                     statusCheckCount = 3;
              }
       }

       // We have finished polling server.
```

```
                        if ( "IP".equals( responseCode )) {

                        // The transaction was still in progress after three iterations
                        through
                        // the polling loop. Kill the session.

                        System.out.println( "The server failed to respond in a timely
                        fashion.");
                        System.out.println( "Try sending transaction again." );
                        System.exit( 0 );
                  } else {

                        // The transaction has completed. Display results.

                        System.out.println ( "Transaction Reference: " + webpay.get (
"TXNREFERENCE" ) );
                        System.out.println ( "Response Code: " + webpay.get (
                        "RESPONSECODE" ) );
                        System.out.println ( "Response Text: " + webpay.get (
                        "RESPONSETEXT" ) );
                        System.out.println ( "Authorisation Code: " + webpay.get (
                        "AUTHCODE" )
);
                        System.out.println ( "Pre Auth Number: " + webpay.get (
                  "PREAUTHNUMBER" )
);
                        System.out.println ( "Error Message: " + webpay.get ( "ERROR" ) );
                  }
            } catch ( Exception e ) {

                  // General exception caught.

                  System.out.println ( "Exception caught." );
                  e.printStackTrace();
            }
      }
}
```

5.1.2 Refund Example

```
package webpay.client.Webpay;
import java.io.*;
import java.lang.*;
import java.util.*;
import java.security.*;
import webpay.client.*;

/**
* Class used for testing refunds with the Webpay java client (webpay.client.Webpay)
* Demonstrates the following:
*
* - Performing a refund
* - Choosing whether or not to include card information in the transaction bundle
* manually or retrieving from original transaction logging
* - Calling initTransaction() and completeTransaction() instead of execute()
* - Catching exceptions
*
* Note: The call to "MyDatabaseConnection" must be replaced with a valid call to the
* retrieval mechanism as determined by your storage or logging procedures.
*/
public class RefundExample {
       public static void main ( String[] args ) {
              String responseCode = null;
              String originalTxnreference = null;
              try {

                     // Check that the necessary arguments are present. The last two are
              optional,
                     // but if they appear they must *both* appear or they will be ignored.

                     if ( args.length < 3 ) {
                            System.out.println( "Usage: WebpayExample [clientid] [value as
                     XX.XX][original txnreference] ([cardnumber] [card expiry date as XXXX])"
                     );
                            System.exit( 1 );
                     }

                     // Construct the webpay object. The parameters are clientid, certpath,
cert_passphrase

                     Webpay webpay = null;
                     try {
                            webpay = new Webpay( args[0] , "webpay", "webpay" );
                     } catch (Exception wt_ex ) {
                            System.err.println( "Error creating the Webpay object" );
                            wt_ex.printStackTrace();
                            System.exit( 1 );
                     }

                     // Put other arguments into Webpay object

                     originalTxnreference = args[2];

                     webpay.put( "TOTALAMOUNT", args[1] );
                     webpay.put( "ORIGINALTXNREF", originalTxnreference );
                     webpay.put( "TRANSACTIONTYPE", "REFUND" );
                     webpay.put( "INTERFACE", "CREDITCARD" );
                     webpay.put( "TERMINALTYPE", "0" );

                     if (args.length > 4) {
                            // If card details are present, put them into the bundle
                            webpay.put( "CARDDATA", args[3] );
                            webpay.put( "CARDEXPIRYDATE", args[4] );
                     } else {

                            // Else get the card details from wherever they have been
                     stored...
                            webpay.put( "CARDDATA", MyDatabaseConnection.getFromDatabase(
originalTxnreference, "CARDDATA" ) );
```

```
                        webpay.put( "CARDEXPIRYDATE",
                    MyDatabaseConnection.getFromDatabase(originalTxnreference,
                    "CARDEXPIRYDATE" ) );
                }

                // Initialise the transaction against the Webpay Transaction Servers.
                // It should return a transaction reference.

                try {
                        webpay.initTransaction();
                } catch ( IOException io_ex ) {

                        // The transaction has failed at a protocol level.

                        System.err.println("Error executing transaction:");
                        io_ex.printStackTrace();
                }

                // The transaction reference and status of the transaction may be
                // logged at this point.

                System.out.println( "Transaction initialised." );
                System.out.println( "Txnreference: " + webpay.get( "TXNREFERENCE" ) );
                System.out.println( "Status: " + webpay.get( "RESPONSECODE" ) );


                // Now complete the transaction.
                try {
                        webpay.completeTransaction();
                } catch ( IOException io_ex2 ) {

                        // The transaction has failed at a protocol level.

                        System.err.println("Error executing transaction:");
                        io_ex2.printStackTrace();
                }

                // No exception has occurred, so the transaction has returned a
                // response code. Check to see if the transaction is still "IN PROGRESS".

                if ( "IP".equals( responseCode )) {

                        // The transaction was still in progress.
                        // Don't poll, let's just kill the session.

                        System.out.println( "The server failed to respond in a timely fashion."
);
                        System.out.println( "Try sending transaction again." );
                        System.exit( 0 );
                } else {

                        // The transaction has completed. Display results.

                        System.out.println ( "Transaction Reference: " + webpay.get (
"TXNREFERENCE" ) );
                        System.out.println ( "Response Code: " + webpay.get ( "RESPONSECODE" ) );
                        System.out.println ( "Response Text: " + webpay.get ( "RESPONSETEXT" ) );
                        System.out.println ( "Authorisation Code: " + webpay.get ( "AUTHCODE" )
);
                        System.out.println ( "Pre Auth Number: " + webpay.get ( "PREAUTHNUMBER" )
);
                        System.out.println ( "Error Message: " + webpay.get ( "ERROR" ) );
                }
        } catch ( Exception e ) {
                System.out.println ( "Exception caught: " );
                e.printStackTrace();
                }
        }
}
```

5.1.3 Servlet Example

```
package webpay.client.servlet;

import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import webpay.client.Webpay;

/**
* Client class to demonstrate a servlet-based webpay transaction execution.
*/

public class WebpayServlet extends HttpServlet {
       public void doGet( HttpServletRequest request, HttpServletResponse response )
       throws IOException, ServletException {

              // The Webpay object

              Webpay webpay = null;

              // Boolean which is true if execute() returns void, false if an exception is
thrown

              boolean result = false;

              response.setContentType("text/html");
              HttpSession session = request.getSession( true );
              PrintWriter out = response.getWriter();

              try {

                     // See if we are in the process of a transaction for this user...

                     webpay = (Webpay)session.getValue( "webpay" );

                     // ...if a transaction is in progress, "webpay" will not be null. If
                     there is
                     // no transaction in process, "webpay" will be null, and will be built
                     up.

                     if ( webpay == null ) {
                            // Normally, details such as keystore paths, passphrases and
                     server details
                            // would NOT be passed as a parameter from the HTTP POST.
                            // It is done here for convenience whilst testing only.

                            webpay = new Webpay(request.getParameter("clientid"),
                                                request.getParameter("keypath"),
                                          request.getParameter("passphrase"));

                            // The 'servers' can contain more than one element, but for
                            testing we
                            // will accept one.

                            String[] servers = new String[1];
                            servers[0] = request.getParameter("server");
                            webpay.setServers( servers );

                            // Set server port

                            webpay.setPort( request.getParameter("port") );

                            // Put the webpay object into the session, in case the user hits
                            // refresh/reload before the transaction completed. The sooner
                            this is
                            // done, the better.
```

```
                session.putValue( "webpay", (Object)webpay );

                // Put the data into the webpay object

                webpay.put( "CARDEXPIRYDATE",
                request.getParameter("cardexpirydate") );
                webpay.put( "CARDDATA", request.getParameter("carddata") );
                webpay.put( "TOTALAMOUNT", request.getParameter("totalamount") );
                webpay.put( "TRANSACTIONTYPE", "PURCHASE" );
                webpay.put( "INTERFACE", "CREDITCARD" );
                webpay.put( "CLIENTREF", request.getParameter("clientref") );
                webpay.put( "TERMINALTYPE", "0" );

                // Execute the transaction

                try {
                        webpay.execute();
                        result = true;

                        // The transaction executed successfully.
                        // The fact that "result = true" DOES NOT necessarily mean
                        that the
                        // transaction was approved, only that it executed.

                } catch ( Exception ee ) {

                        // The transaction failed, at a protocol level, to execute
                properly.

                        result = false;
                }

                // Put the result into the session so the polling thread can get
        to it.

                session.putValue( "wp_result", new Boolean( result ) );
        }

        int c = 0;
        while ( session.getValue( "wp_result" ) == null && c < 60 ) {

                // Poll for the result for up to 30 seconds.
                // This is a safety measure for checking against the session for
                the
                // result if another thread is doing the transaction.
                // Implementors might wish to pop up a window letting users know
                they
                // need to wait.

                Thread.sleep( 500 ); // Half second polling delay.
                c++;
        }

        if ( session.getValue( "wp_result" ) == null ) {

                // Get the result from the session, if it is not there it failed.

                result = false; // Send error message
        } else {
                result = ((Boolean)session.getValue( "wp_result"
        )).booleanValue();
        }

        // Retrieve and display transaction results

                out.println("<html><body bgcolor=\"lightblue\"><head></head><font
    face=arial size=2>");
                out.println( result ? "<b>Succeeded</b><br>" : "<b><font
color=#440000>Failed</font></b><br>" );
                out.println("Auth code: " + webpay.get( "AUTHCODE" ) + "<br>" );
                out.println("Error: " + webpay.get( "ERROR" ) + "<br>" );
                out.println("Response Code: " + webpay.get( "RESPONSECODE" ) + "<br>" );
```

```
                        out.println("Response Text: " + webpay.get( "RESPONSETEXT" ) + "<br>" );
                        out.println("TxnRef: " + webpay.get( "TXNREFERENCE" ) + "<br>" );
                        out.println("</font></body></html>");

                        // Remove the details from the session as we have completed the
                        transaction.
                        // The out.println() calls should throw an exception if the output stream
                        is closed.
                        // This way, the reload of the page will be able to access the objects
                        // from the session, and report on the result.

                        session.removeValue( "webpay" );
                        session.removeValue( "wp_result" );
                } catch ( Exception e ) {
                        throw new IOException( e.getMessage() );
                }
        }

        public void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException {
                doGet( request, response );
        }
}
```

6 Getting Help

6.1 IPG Web Site

The complete API documentation and software is available from **https://www.ipg.stgeorge.com.au**.

6.2 Contacting St. George Bank

Before reporting errors, **ensure that you are able to replicate them**, so that we are able to diagnose properly.

Be prepared to have available for the Helpdesk or e-mail the following information:

- Operating system + API type (i.e. – W2000 + java API)
- Basic Hardware description (i.e. – P3 800 + 512 meg RAM)
- Debug logs & Log file containing the values of all fields in the transaction
- Description of the error (when and how it happened)
- Error code and Error message

The St. George Bank Technical Support Team can be contacted in the following ways:

- Telephone via the St. George EFTPOS Helpdesk 1300 650 977
- Email to: ipgsupport@stgeorge.com.au
- On-Line at https://www.ipg.stgeorge.com.au

Appendix A – Files included in this kit

The following files are included in this kit and are installed into relevant sub-directories.

Documentation

St. George Java API Developer Guide	This document
Webpay.html	The Javadoc file
README	Quick installation instructions

Base libraries

WebpayClient.jar	The jar file that consists of key utility class files.

Test Files

BaseTest/BaseTest.java	A simple java test source file.
BaseTest/BaseTest.class	The compiled class file of the test source file, BaseTest.
BaseTest/comp_base_test.sh (Linux)	The batch file that builds the test file from the source code.
BaseTest/run_base_test.sh (Linux)	The batch file that executes the test program, using the default values for the input to the variables.
BaseTest/run_base_test.bat (Win32)	The batch file that executes the test program, using the default values for the input to the variables.
TestTransaction	TestTransaction is located in the WebpayClient.jar. It is invoked when the Jar file is double clicked. It can also be run from the command line using the following command. java –jar WebpayClient.jar

NOTE: For security purposes the java keystore containing the client certificate file will be supplied separately from this Kit.

Appendix B – Gateway load balancing

The IPG network consists of two duplicate streams in order to provide failover protection. Due to the way the Java Virtual Machine (JVM) by default caches the DNS lookup, Java Applications may not be making the most of the redundancy the IPG network provides.

There are a few ways to utilise the full failover capability of the IPG Servers:

1. The Network address cache TTL could be altered on your application to force the JVM to not cache the DNS record. See more information on ***networkaddress.cache.ttl*** in the relevant java documentation.

2. St.George has two additional DNS entries to resolve to each IPG stream. Your application can make better use of the redundancy of both server streams by sending through the default server DNS followed by the DNS of each stream.

The Application code would look like this:

String[] servers ={www.gwipg.stgeorge.com.au","www.gwipgk.stgeorge.com.au","www.gwipgg.stgeorge.com.au};
webpay.setServers(servers);

To make use of all steams you will need to pass an array string to the Java API "setServers" property. The API would then self load balance by sending the first attempt to: www.gwipg.stgeorge.com.au then www.gwipgk.stgeorge.com.au then www.gwipgg.stgeorge.com.au.