Internet Payment Gateway

Swig API Developer Guide

(PHP, Perl and C++)

st.george

Table of Contents

Overview

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 Transaction Server document set. It intended to act a reference point for developers seeking to implement the St.George transaction functionality into merchant applications. It provides specific technical information about the typical deployment of the St.George 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 linux environment, and this document is written principally toward parties with this expertise.

Related Documents

- **St.George Bank Win32 Developer Guide**
- **St.George Bank .Net Developer Guide**
- **St.George Bank Java Developer Guide**

1 Introduction

The St.George SWIG API enables web site and application developers to efficiently add credit card transaction processing capabilities to their products.

The base of the St.George SWIG API is the C++ library libwebpayclient.so. This library can be used directly from C++ code or it can be wrapped using SWIG to enable scripting languages such as Perl and PHP to have access to its interface.

St.George SWIG can be used with a Linux based operating system that has an implementation of SWIG 1.3.36 (or later) installed. SWIG (Simplified Wrapper and Interface Generator) is defined by www.swig.org as:

"*SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. In addition, SWIG provides a variety of customization features that let you tailor the wrapping process to suit your application.*"

The power of using SWIG is that it gives merchants the ability to use the core St.George code, libwebpayclient.so, in a variety of different languages and versions. This API provides example implementations for C++, PHP 5 and Perl 5 however merchants can implement any of the other languages supported by SWIG.

As additional interfaces and transaction types become available, St.George Bank will issue merchants and developers with the relevant information regarding the new parameters and other settings needed to adopt these transaction types. If the Merchant decides to implement this new functionality, it is simply a matter of recompiling the swig interface and copying the required files into the required directories.

This document should be used in conjunction with the appropriate ***Transaction Specification*** document to suit the transaction types required for your implementation.

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

Communication between the St.George API and St.George 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 should ensure that the relevant addresses and ports are not being filtered or blocked.

It is important to note that the St.George 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 WTS 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 representative you 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 SWIG API.

Operating System	**Linux** with glibc 2.5
Processor	Pentium 4
Processor Speed	3 Ghz
Memory	1 Gb
Additional Requirements	Open SSL 0.9.8b SWIG 1.3.36 or later (1.3.36 is required for support of php5) C compiler - gcc-4.1.2 Both 32bit and 64bit operating environments are supported.

The following sections detail the dependencies required for SWIG API. Depending upon the configuration of your system you may need to follow these steps.

1.1.1 OpenSSL Libraries

All St.George APIs depend on the OpenSSL libraries. These libraries should be OpenSSL version 0.9.8b or greater, and must be correctly installed on your machine before the St.George client software will execute successfully.

These files will commonly be installed as part of your distributions build.

The OpenSSL libraries have names similar to **libssl.so.6** and **libcrypto.so.6**. They may be installed as part of the operating system and usually reside in the **/lib** or **/usr/lib** folder. If your system lacks these files, please download pre-compiled versions from a trusted site or download the OpenSSL source and compile it on your machine.

If you experience problems with SSL connectivity, you **may** need to create a symbolic link to the library binaries. For example, the following lines create the necessary links:

```
ln -s /usr/lib/libcrypto.so.0.9.8b /lib/libcrypto.so.6
ln -s /usr/lib/libssl.so.0.9.8b /lib/libssl.so.6
```

1.1.2 SWIG

SWIG (Simplified Wrapper and Interface Generator) is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, PHP, Python, Ruby, and Tcl. SWIG has been used in this instance to create wrapper methods to the SWIG API client library that can be called directly from a PHP script.

You must have SWIG installed on your system before commencing installation of this API.

If you require PHP 5 support you will need SWIG version 1.3.36 or higher. If you package manager does not support this version check the SWIG website for a more recent release.

For more information please visit the swig homepage http://www.swig.org

1.1.3 GCC

A C compiler is required to compile the St.George SWIG Libraries. GCC is often installed as part of a Development Tools package from your package manager.

Alternatively the latest version of gcc can be obtained from http://gcc.gnu.org

1.1.4 sed (The stream Editor) that supports the -i flag

The sed command is only used during the make process and is used to insert some version information into the source files. If you systems does not support sed, then a version can be obtained from http://www.gnu.org/software/sed/

1.1.5 Optional Libraries

The following libraries will need to be installed based if your specific implementation requires them.

1.1.5.1 PHP-DEVEL

The php-devel package is required to compile the St.George PHP Library.

Php-devel can be installed using your distributions package manager: e.g.:

yum install php-devel.

1.1.5.2 PERL

The perl package is required to compile the St.George Perl Library.

Perl can be installed using your distributions package manager: e.g.:

yum install perl.

2 Installation

The St.George SWIG API has been thoroughly tried and tested under a number of different Linux distributions using the PHP 5 and Perl 5 languages. The API will also work with other languages; however, these are **not officially supported**.

The installation is delivered as a gzip tarball, named **webpaySWIG-X.X.tar.gz** (Where X.X is the current version number), which contains all the necessary components including sample source code, and supporting libraries.

Before commencing installation,

- please check that your system meets the minimum system requirements listed above.
- please note that all commands are case-sensitive.
- NB: you may require administrative privileges to perform some steps during this install. (i.e. if the generated libraries are required to be installed into system folders)

Follow these steps to install the API:

1 Copy the **webpaySWIG-X.X .tar.gz** file to your workspace

2 In your workspace, decompress the file by typing: **`tar –xzvf webpaySWIG-X.X.tar.gz`**

3 The file will now decompress the following component files into a directory structure: See Appendix A for details about each of these files.

```
makefileC++
makefilePhp5
makefilePerl5
README
libwebpayclient.Y.Y.so.i386
libwebpayclient.Y.Y.so. x86_64
swig\webpay.c
swig\webpay.h
swig\webpay.i
c++\BaseTest.cpp
c++\BPlatform.h
c++\BTypes.h
c++\README
c++\run_base_test.sh
c++\run_status_test.sh
c++\StatusCHeck.cpp
c++\webpayclient.h
php5\statusCheck.php
php5\test.php
php5\webpayForm.php
php5\webpayFormHandler.php
perl5\statusCheck.pl
perl5\test.pl
perl5\webpayForm.pl
perl5\webpayFormHandler.pl
```

3 Building the APIs

Please note that the SWIG API can generate code for a variety of different application languages. The three most common languages used with the SWIG API are listed below. Please follow the steps related to your choice of application language or alternatively use the steps below as a guide for implementing an alternative language (such as python).

3.1 Building the C++ API

The C++ API is a base level API in that it does not require SWIG to interface with the Core Webpay library. The C++ interface uses the same base library as all the other API's however it does not need any code to link that library to the C++ runtime. Follow the steps below to build the C++ API.

1. To build the C++ API run the following command :

   ```
   make –f makefileC++
   ```

   Note: you may be required to modify some values in the C++ code as specified in the next step. If so you may need to recompile the C++ code using this step again.

2. You are now ready to test the compiled code. Edit the test scripts in the c++ directory (see Supplying Input to the Test Script) and then perform the test by running :

   ```
   ./run_base_test.sh, or
   ./run_status_test.sh
   ```

The install is now complete. If you ran into problems during the build process please verify your system minimum requirements. See section Minimum System Requirements and Testing your Installation below for details.

3.2 Building the PHP 5 API

Building the PHP 5 API is a simple process using SWIG, and the GCC compiler to link the PHP code and the Webpay Core library. GNU Makefiles are used to simplify the steps involved. Follow the steps below to build the PHP 5 API.

1. Edit the file `makefilePhp5` and make the appropriate changes to the following fields:

Field Name	Default Value	Comment
PHP_EXTENSIONS	/usr/lib64/php/modules/	PHP_EXTENSIONS is the location where the Webpay extension will be stored. This value is defined in phpinfo() as extension_dir. Typical value would be: /usr/lib/php/modules or /usr/lib64/php/modules. If your system supports the dl() function and loading extensions from outside the specified PHP extensions directory (PHP versions less than 5.2.5) than this can be any path on your system.
PHP_INCLUDE_DIR	/usr/include/php/	PHP_INCLUDE_DIR is the path to your PHP include directory. Typical value would be: /usr/include/php/
LIB_LOCATION	$(PHP_EXTENSIONS)	LIB_LOCATION is the location of the Webpay Client library. Usually this will be stored in the same location as the Webpay extension (webpay_php.so)

2. To build the PHP API run the following command :

   ```
   make –f makefilePhp5
   ```

   Please note that if your PHP_EXTENSIONS directory is a system directory you may need to run the above command with administrative privileges to ensure the generated library files can be copied successfully.

   ```
   sudo make –f makefilePhp5
   ```

3. You are now ready to test the compiled code. Edit the test files in the php5 directory (see Supplying Input to the Test Script) and then perform the test by issuing the following command from the php5 directory:

```
php test.php
```

4. The test php files found in the php5 directory of the distribution are configured to look for the Webpay php library file in the local directory. If the default configuration does not work you have the following options:

   a. If your PHP version does not allow paths to be set in the dl() function (PHP version 5.2.5 and above). Then the libraries must be loaded from the php extensions directory. First confirm the libraries libwebpayclient.so and webpay_php.so are located in your php extensions directory and then change the load_library function to dl()

      To locate the extensions directory review the **extension_dir** property displayed via the php function phpinfo().

   b. If you are running on a system that does not have the dl() function enabled or is running in SafeMode then you will have to load the Webpay library at system startup by adding it as an extension to the php.ini file.

      i.e. edit /etc/php.ini

      Go to the location where the extensions are defined and add the following:

      ```
      extension=webpay_php.so
      ```

      Save the file and restart apache. In this instance the Webpay Extension will appear in phpinfo() and will not need to be loaded in the transaction page. The Webpay methods can simply be invoked directly.

The install is now complete. If you ran into problems during the build process please verify your system minimum requirements. See section Minimum System Requirements and Testing your Installation below for details.

3.3 Building the Perl 5 API

Building the Perl 5 API is a simple process using SWIG, and the GCC compiler to link the Perl code and the Webpay Core library. GNU Makefiles are used to simplify the steps involved. Follow the steps below to build the Perl 5 API.

1. To build the Perl 5 API run the following command:

```
make –f makefilePerl5
```

2. You are now ready to test the compiled code. Edit the test files in the perl5 directory (see section Supplying Input to the Test Script) and then perform the test by issuing the following command:

```
make –f makefilePerl5 test
```

   Alternatively you may run the test directly by issuing the following command:

```
perl test.pl
```

The install is now complete. If you ran into problems during the build process please verify your system minimum requirements. See section Minimum System Requirements and Testing your Installation below for details.

3.4 Additional Tips when Compiling and Linking

If you are having difficulty compiling and linking in your local environment check the following steps are being followed.

- Check that the file: libwebpayclient.so has been created and it is linked to the correct library file for your operating environment: The two options are (where Y.Y is current version of the Core Webpay Library):
    - libwebpayclient.Y.Y.so.i386 - The 32 bit version of the Core Webpay Library
    - libwebpayclient.Y.Y.so.x86_64 - The 64 bit version of the Core Webpay Library
- Place the shared object file, libwebpayclient.so, in the appropriate folder. In an uncontrolled environment where the user has Administrative permissions the recommended path is /usr/lib, however if the environment is a controlled hosted environment the path will need to be set to a directory in the user home directory.
- Ensure that parameter LINUX is defined for the pre-compiler (-DLINUX or #define LINUX).
- When the makefile is run and the source is compiled the Swig generated library must be told how to link to the Core Webpay library (libwebpayclient.so). The sample makefiles are setup with two examples of how to link the libraries.

  The PHP example uses the –Wl flag to pass the $ORIGIN token through to the linker. i.e.

  ```
  -Wl,-R,'$$ORIGIN'
  ```

  The $ORIGIN flag indicates to the linker to look in the current directory and can be used with relative paths. i.e.

  ```
  -Wl,-R,'$$ORIGIN/../lib'
  ```

  The Perl example uses the –Xlinker and –rpath flags .i.e.

  ```
  gcc –shared $(LINK_PATH) $(WEBPAY_LIB) languageImpl.cpp
  ```

  The variables are:

  ```
  LINK_PATH = -Xlinker -rpath ./
  WEBPAY_LIB =./libwebpayclient.so
  ```

  The LINK_PATH tells the runtime code where to look to find the libwebpayclient.so and the WEBPAY_LIB tells the compiler which shared library to use. When the LINK_PATH is a relative path care must be taken to determine which directory the runtime is running in when executed. If a relative path does not work try using an absolute path and make sure the executing process has access to that directory. When the WEBPAY_LIB path is a relative path make sure it exists in the correct location and is accessible by the process. If unsure change it to an absolute path to remove any doubt.
- When building each language you will need to specify the list of directories that contain all the required header files. Check the makefile to ensure they are correct.

  For the PHP 5 API the specified directories are:

  ```
  -I/usr/include/php/main -I/usr/include/php/Zend/
  -I/usr/include/php/ -I/usr/include/php/TSRM/
  ```

4 Testing your Installation

4.1 SWIG API Specific Information and Files

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

- The address of the St.George Gateway
- The Test (and Live) Gateway port numbers
- Your St.George Client ID
- A St.George client certificate file (e.g., "cert.cert") appropriate for your St.George gateway and the password protecting this certificate file

You must place the St.George client certificate file somewhere on your machine so that the St.George client software can access it. It is recommended that you place the file in the same folder with all your other St.George client files.

4.2 The Base St.George Client Test Script

A basic test script is included with all the API development kits (i.e. Perl, PHP, C++, Java).

The PHP script is called: **test.php** and it is located in the php5 directory. The perl script is called: **test.pl** and is located in the perl5 directory. You should run the relevant test script to ensure that:

- The OpenSSL libraries are correctly installed
- The OpenSSL libraries are of OpenSSL version 0.9.8b or greater
- The basic Webpay Client library has been installed.
- The Webpay PHP or Perl Library was successfully created by the Make file.
- There are no other basic connectivity problems with your machine

See the troubleshooting section of this guide for more information.

4.2.1 Supplying Input to the Test Script

Open the test files using any standard text editor and you will notice a number of defined constants, for example:

- `define("THE_CLIENT_ID", "10000000"); // PHP code`
- `use constant THE_CLIENT_ID => "10000000"; // PERL code`
- `CLIENT_ID=10000000 // in C++ bat file that invokes the C++ program`

Replace the value of the constant with those appropriate for your environment. Using the line above as an example, the value **10000000** would be replaced with your assigned Client Id.

Typically, the hard-coded values you will need to change are set using the following functions:

Parameter	Used by function	Description
THE_CLIENT_ID	put_ClientID	Set the designated Client ID
THE_CERT_PATH	put_CertificatePath	The name and path and of the certificate file. The path is relative to the current directory. If the cert.cert file resides in the root directory /cert.cert would be specified.
THE_CERT_PASSWORD	put_CertificatePassword	The passphrase that protects the certificate
THE_PORT	setPort	The designated port number for the St.George gateway. The port number used will be determined on whether the client is

		connecting to the live or test environment.
THE_SERVER	setServers	The IP address or DNS of the St.George gateway server. Multiple gateways may be separated by commas.

4.2.2 Transaction Response Analysis

The output of the test program should be similar to …

- Response Code = “00”
- Response Test = “Approved”
- Error Message = “”
- Transaction Reference = 1000000123469

Response Code	A two digit code used to determine success or failure. The code number matches the response text field. “**00**”, “**08**” or “**77**” designate an **Approved** transaction. Response codes are contained in The *Transactions Specifications* documents that accompany this kit. A response code < 0 means that a request was not processed properly and should be tried again. Users will need to consult the logs to determine what the error was that caused the transaction to fail.
Response Text	A meaningful text message that explains the response code
Error Message	Any errors received
Transaction reference	A unique reference number generated for each transaction.

4.2.3 Troubleshooting

If your output fails any of the tests above, please review the steps you have taken so far and ask yourself the following questions:

- Are you using the correct Client ID?
- Are the OpenSSL libraries somewhere in the expected location or symbolically linked?
- Is the basic libwebpayclient.so library in the expected location (i.e. system library folder or php modules directory), does the executing program have permission to access the library?
- Is the location of the SWIG API client certificate file specified correctly? Try the file’s full path if you aren’t already.
- Is the password correct for the SWIG API client certificate?
- Do you have the appropriate permissions? If you are using the internal SWIG API client library logfile ensure the directory where this logfile is created has write permissions.
- Is the correct address and test port number specified for the St.George gateway? You may want to use “nslookup” or “dig (domain information groper)” to ensure that the address resolves correctly.
- Are you behind a firewall? You may need to contact your Security Administrator to ensure the appropriate ports have been defined through the firewall.

If you still can’t identify the problem, call your St.George Bank representative. Be prepared to e-mail the output of the test program.

4.2.4 Network Errors

Commercial and freeware tools can be found on the Internet, to assist you in testing your network connectivity to the Webpay the Secure Transaction Servers.

4.2.5 Unexpected Errors While Testing

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 *Transactions Specifications* documents accompanying this kit.

Some response codes may be generated at any time and this behaviour is perfectly normal. For example, if the system is being subjected to load testing the code `'A6'` for `'Server Busy'` may be expected.

Before reporting errors, ensure that you are able to replicate them, so that our support team can diagnose properly.

4.2.6 Common errors and solutions

The following section outlines some common errors and their solutions.

Error Message	Comment
Warning: dl(): Temporary module name should contain only filename in /home/webpayuser/webpaySWIG-3.2/php5/test.php on line 217	This error message is seen on systems running PHP 5.2.25 or higher or systems running in safe mode. These systems can only load PHP extensions from the system specified extensions directory. This means that the Webpay libraries need to be copied into the system extensions directory and loaded with the dl command (excluding any path information). Additional information: view phpinfo() to find your systems php extensions directory. Then ensure both the libwebpayclient.so and the webpay_php.so library are copied into the extensions directory. Update your php file so that the webpay library is loaded like this: dl("webpay_php.so") This will automatically load the library from the extensions directory.
php: symbol lookup error: /usr/lib/php/module/webpay_php.so: undefined symbol: executeTransaction	This error indicates that the Swig generated Webpay library webpay_php.so cannot communicate with the Core Webpay library libwebpayclient.so. Please ensure that the libwebpayclient.so file is in the same directory as the webpay_php.so library.
swig -php5 php5/webpay.i swig error : Unrecognized option -php5	The version of swig you are running is old and does not support php5. Please update your version of swig and ensure that the makefilePhp5 is updated to point to the correct location for the swig executable. Try the command: **whereis swig** to see if there is more than one version of swig installed on your system. Try each of them with the absolute path and the –version flag. i.e. /usr/local/bin/swig –version Once you have located the version that is greater than 1.3.36 update makefilePhp5 to include the fullpath: SWIG = /usr/local/bin/swig

Running SWIG to create wrapper files /usr/local/bin/swig -php5 php5/webpay.i No module name specified using %module or -module. make: *** [swig] Error 1	This indicates there is a problem with the generated webpay.i file. Please check that the generated file php5/webpay.i is not empty and also check that you system supports the sed command (with the –i flag).
PHP Fatal error: Call to undefined function newBundle() in test.php	This error appears if there is an error in the php.ini configuration for the webpay extension. Solution: ensure the php.ini entry for the webpay extension is correct extension=webpay_php.so OR Load the extension in the calling PHP file with dl("webpay_php.so");

PHP Warning: Module 'webpay' already loaded in Unknown on line 0	This error is displayed if the webpay module is already loaded via php.ini and is attempted to be loaded via dl()
	Use the extension_loaded function to see if the extension has already been loaded before loading it locally:
	if(!extension_loaded('webpay')){
	dl(THE_WEBPAY_PHP_LIB);
	} else {
	print "Webpay Module Already Loaded
	\n";
	}

PHP Warning: PHP Startup: Unable to load dynamic library '/usr/lib64/php/modules/webpay_php.so' - /usr/lib64/php/modules/webpay_php.so: cannot open shared object file: No such file or directory in Unknown on line 0 PHP Fatal error: Call to undefined function newBundle() in test.php	This error is displayed if the webpay extension is loaded via php.ini but the library file isn't available in the extensions directory.
wrap_newBundle not available There has been an initialisation problem. Please check error log	Ensure libwebpayclient.so has correct permissions and is located in the appropriate directory (typically the same directory as your webpay_php.so extension)
webpay_php does not exist: [webpay_php.so]	The webpay_php.so file has not been loaded by php. Check it's location and that it has been configured correctly either in php.ini or in the call to the dl function.
Copying libwebpayclient.so to /usr/lib64/php/modules/ cp -p libwebpayclient.so /usr/lib64/php/modules/ cp: cannot create regular file `/usr/lib64/php/modules/libwebpayclient.so': Permission denied make: *** [copyWebpayLib] Error 1	This error may appear during the make process. This error indicates that the user running the make process does not have sufficient permission to copy the library to the specified location. Try running the make process with sudo i.e. sudo make –f makefilePhp5

4.2.7 Switching on debugging

The APIs includes some inbuilt debug logging, which by default is switched off. The following instructions illustrate how to enable the debug logging should you require it.

1. Edit your php page and add the following lines:

```
put($webpayRef, "DEBUG", "ON");
put($webpayRef, "LOGFILE", "webpay.log");
```

   The sample scripts include these entries but the DEBUG value is set to OFF.

   Please ensure that the process running the php script has write privileges to the location specified in the LOGFILE parameter.

2. You can obtain additional low level logging from the SWIG wrapper library by doing the following

   Edit the swig/webpay.c file

   Find the function: void debug_logging(const char * message, ...)

   Comment out the first line (which is a return statement) so that the function will run through to completion.

   i.e.

```
void debug_logging(const char * message, ... )
{
      //return;
      va_list args;
      va_start(args, message);
      vfprintf( stderr, message, args );
      va_end(args);
}
```

   Then re-run the make command i.e. make –f makefilePhp5 to recreate the SWIG library, and re-run your test.

5 Additional Information

5.1 Security

The *St.George Transaction Server* maintains secure transactions through the use of SSL (Secure Sockets Layer). The SSL Handshake Protocol was originally developed by Netscape Communications Corporation to provide security and privacy over the Internet. Using SSL, we can provide client and server authentication, and ensure 128-bit encryption of all transaction details.

5.2 Performance

The St.George Transaction Servers are clustered to provide high availability, fail-over, and fast processing under load. The average time taken to complete a transaction is 6-7 seconds, but merchants may experience faster or slower transaction completion times due to a number of factors, including: the "distance" (or number of hops) between the merchant's server and the St.George environment, the type and speed of connection, general internet congestion, high server load, etc. The API's automatically manage transaction and connection timeouts.

5.3 Connectivity

Depending upon the WTS implementation, a number of gateways may be utilised for communication with the Client APIs. This implementation ensures maximum connection and system failover. The API may be directed to multiple gateways, and if the first is unavailable the next gateway in sequence will be contacted. This is defined by including each gateway address separated by a comma when invoking the `setServers` function.

5.4 Test/Live Merchant Status

St.George Bank provides a Test and a Live system for merchant use. The Test system is a mirror of the Live system, with the exception that it sends the requests to a *simulated* banking environment.

Initially, you will need to develop and test your code using the test system. When you believe you are ready to "Go Live", you must contact the St.George Bank Support Team to begin the accreditation process.

The St.George Servers produce a wide range of transactional responses. Within the testing regime, these responses are controllable through the allocation of the cents component in the amount value. To fully test your system's ability to handle and control varying responses, you should test all cents values between 00 and 99.

A complete list of all creditcard response codes is contained in The *Transactions Specifications* documents that accompany this kit.

Contact the St.George Bank Support team for the current Server parameters required when using the Test and Live systems.

5.5 Session and Concurrency Issues

When implementing the St.George SWIG API, it is important to consider the effect of multiple users accessing your application concurrently. A new transaction context should be created for every transaction, to provide a mechanism for maintaining a user's session, which will allow each transaction instance to be processed only once.

6 API Reference

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

For information on Credit Card transactions and parameters see the separate *Creditcard Transaction Specifications* document.

6.1 Available Functions/Methods

6.1.1 cleanup($webpayRef);

[returns void]

This will destroy the Transaction object. Ideally, this should be called instead of flushBundle, and then it should be followed by another call to newBundle () to process another transaction.

6.1.2 executeTransaction($webpayRef);

[returns char * - “true” or “false”]

This actually executes the transaction. It will send the information to the server specified, and return true if it was sent okay, or false if it could not be sent. **This return value does not indicate an approved or declined transaction.**

6.1.3 flushBundle($webpayRef);

[returns void *]

This will empty the Transaction object of all data, and allow it’s use by other clients and transactions. Returns false or NULL if it has a problem. For example, if transactions were being processed inside an iterative loop with a single Transaction object being used over and over again with fresh data each time, this is the method to use in order to empty the data. It will return a reference to the Transaction object.

6.1.4 get($webpayRef,value);

[returns char *]

Gets the value specified from the Transaction, eg "RESPONSECODE" will return the response given by the bank to the transaction. This particular example will give the code indicating whether the transaction was approved or declined. For more information on responses try getting the key "RESPONSETEXT".

6.1.5 newBundle();

[returns void * - $webpayRef]

Initialises the Transaction. Once this has been called, information may be added to the Transaction. If there has been a problem it will return false or NULL, in most cases it should return a valid $webpayRef which can then be used to build up a valid transaction.

6.1.6 put($webpayRef,“DEBUG”,“ON”);

[returns void]

Specifies whether debugging is enabled for the transaction. This method must be used in conjunction with “LOGFILE” to stipulate the output location. The valid options are ON and OFF. If not specified the default value is OFF.

6.1.7 put($webpayRef,"LOGFILE",value);

[returns void]

Specifies the output file and location for the debugging log file. This method must be used in conjunction with “DEBUG”. For example, put($webpayRef, “LOGFILE”, “/home/dev/webpay.out”). Care must be taken to verify that the user running the API has write access to the directory specified.

6.1.8 put($webpayRef,name,value);

[returns void]

Puts a value into the Transaction under the name specified. Eg, put($webpayRef, "COMMENT", "testing") will put the value "testing" under the key, or name, "COMMENT".

6.1.9 put_ClientID($webpayRef,newVal);

[returns void]

Puts the specified clientid into the Transaction. This may be retrieved by calling get($webpayRef, "CLIENTID").

6.1.10 put_CertificatePassword($webpayRef,newVal);

[returns void]

Puts the specified certificate password into the Transaction. This is generally a set-and–forget value.

6.1.11 put_CertificatePath($webpayRef,newVal);

[returns void]

Puts the specified certificate path into the Transaction. This is generally a set-and–forget value.

6.1.12 setPort($webpayRef,Port);

[returns void]

Puts the specified (server side) port into the Transaction. This may be retrieved by calling get($webpayRef, "SERVERPORT").

6.1.13 setServers($webpayRef,ServerList);

[returns void]

Puts the specified server list into the Transaction. This may be retrieved by calling get($webpayRef, "SERVER"). The server list is a comma separated list of servers to connect to. The order of execution is always from first to last.

6.2 Error/Exception Handling

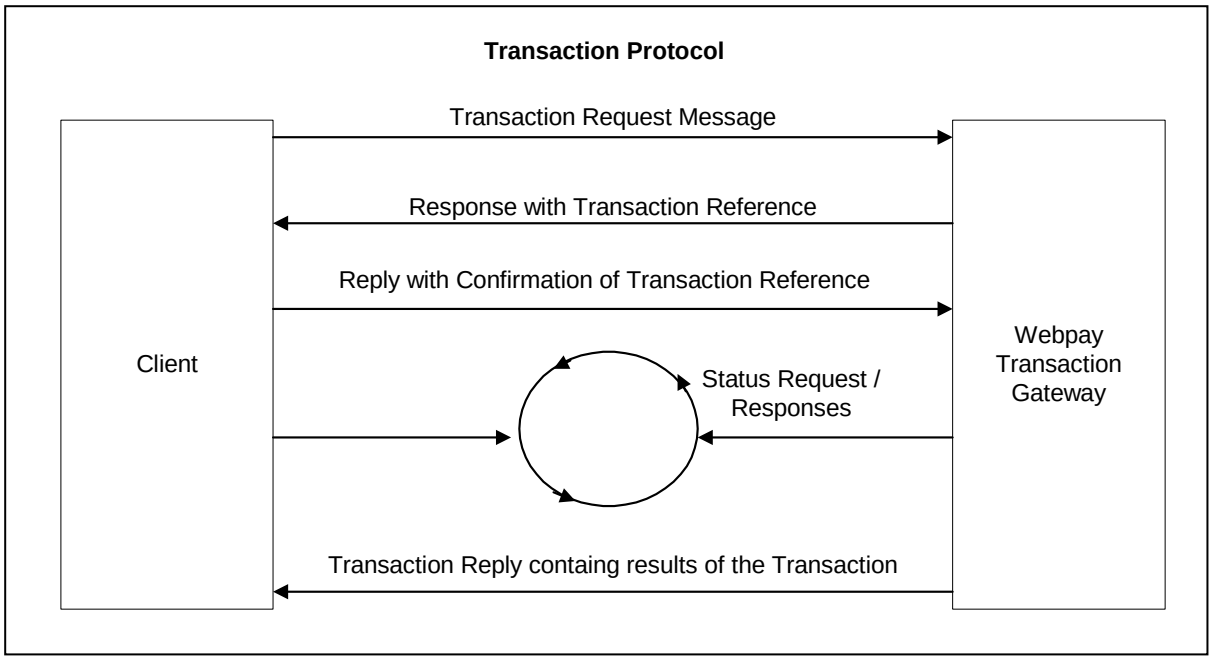
The St.George APIs handles all errors internally. Any errors encountered during processing, cause the transaction to fail. Error messages are viewed by investigating the ERROR field.

If the response code is set to a value greater than -1 it means that the St.George server has rejected the transaction. If the result is a negative number then the chances are that the error occurred while trying to contact the St.George server. With either error users should check the ERROR or RESPONSETEXT to further help remedy the error.

The St.George Libraries may log additional errors to stderr. Check your webserver error log for output.

If a transaction attempt fails and a Transaction Reference has been returned by the St.George Servers, the executeTransaction method will automatically query 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 enabled to query the status of such a transaction, and be able to do so at any time.

The following diagram depicts the transaction protocol implemented.



7 Technical Support

7.1 IPG Website

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

7.2 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. – Linux + PHP API)
- Basic Hardware description
- 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

8 Appendix

8.1 Appendix A – Files included in this kit

Documentation

File	Description
St.George SWIG API Developer Guide	This document
README	Quick reference document. May contain latest configuration information.

Certificate file

File	Description
<certificate file> for example: webpay.p12	The certificate file for the secure transaction **NOTE: For security purposes, this file will be supplied separately**

Base libraries

File	Description
libwebpayclient.Y.Y.so.i386	The 32 bit version of the Core Webpay Library (where Y.Y is current version of the Core Webpay Library such as libwebpayclient.1.16.so.i386)
libwebpayclient.Y.Y.so .x86_64	The 64 bit version of the Core Webpay Library (where Y.Y is current version of the Core Webpay Library such as libwebpayclient.1.16.so .x86_64)
libwebpayclient.so	This file is automatically created by the relevant Makefile. It is linked to the appropriate 32bit or 64bit library described above.

Build files

These files are used to create the API libraries.

File	Description
makefilePhp5	Compile script that builds the PHP 5 libraries.
makefilePerl5	Compile script that builds the Perl 5 libraries.
makefileC++	Compile script that builds the C++ libraries.
swig\webpay.c	A c program that provides wrapper functions to the Webpay Client Library.
swig\webpay.h	Header file for webpay.c
swig\webpay.i	The SWIG interface file used by SWIG.

Test Scripts

File	Description

c++\run_base_test.sh	A simple test script that executes a transaction.
c++\run_status_test.sh	A simple test script that runs a status check on a transaction.
php5\test.php	A simple test script that can be run from the command line i.e #php test.php
php5\webpayForm.php	A more complicated test script that can be run through your browser. Please ensure that this file is in your webserver's public_html directory.
php5\webpayFormHandler.php	Used by webpayForm.php
php5\statusCheck.php	A test script that illustrates how to perform a status check on a transaction.
perl5\test.pl	A simple test script that can be run from the command line… i.e #perl test.pl
perl5\statusCheck.pl	A test script that illustrates how to perform a status check on a transaction.

Miscellaneous Files

File	Description
c++/BaseTest.cpp	The source C++ file for the base test. The sample program illustrates how to perform a transaction.
c++/StatusCheck.cpp	The source C++ file for the status Check. The sample program illustrates how to perform a status check on a transaction.
c++/webpayclient.h	The header file contains declaration for public methods.
c++/BPlatform.h	Contains the platform specific declarations and definitions.
c++/BTypes.h	The header file that contains helper declarations and definitions

8.2 Appendix B – General Makefile Properties

For each scripting language and version of that scripting language a new makefile must be created. All the makefiles follow a similar format.

Adding these lines in the makefile will ensure that the makefileBase is included in the current build which will make all their targets accessible. To customise the build for different languages the following properties are available to be modified.

Properties

Property	Value	Description
SWIG	swig	The path to the swig executable. If swig is not in the system path enter the relative or absolute path.
WEBPAY_CLIENT_LIBRARY	libwebpayclient.so	The name of the Webpay client library file. We do not recommend changing the value of this property.
CC	gcc	The value specified here is the C or C++ compiler used to compile the client code.
DEST_DIR	php5	This property specifies the directory that will be used to copy and build each language.
SWIG_OPTIONS	-php5 $(DEST_DIR)/webpay.i	The SWIG options are all the command line arguments that you want to be passed into SWIG. For more information on all the SWIG options either look at their web page (www.swig.org) or execute swig –help on the command line.
ALL_SWIG_FILES	$(DEST_DIR)/webpay_wrap.c $(DEST_DIR)/php_webpay.h $(DEST_DIR)/webpay.php	List all the files that are created by swig. The build process uses these files to clean the build directory of all the created files.
SOURCE_FILES	$(DEST_DIR)/webpay_wrap.c $(DEST_DIR)/webpay.c	List all the files require compilation by the C compiler. These files are passed into the compiler to build the so file.
WEBPAY_LIBRARY	webpay_php.so	Specify the name of the shared library file.
FILES_TO_CLEAN		List all extra files that you would like cleaned up by the build process.
CFLAGS	-fPIC -Wl,-R,'$$ORIGIN' -shared -I$(PHP_INCLUDE_DIR)main -I$(PHP_INCLUDE_DIR)Zend/	Enter all the flags required by the C compiler.

-I$(PHP_INCLUDE_DIR)

-I$(PHP_INCLUDE_DIR)TSRM/

* The PHP 5 API values have been used to illustrate the values

Targets

Property	Description
all	The all target executes the swig, compileStep and copyWebpayLib targets
swig	The swig target copies the swig related files to the DEST_DIR property and then applies the version and client type info in the code to values set by the property VERSION and DISTNAME. SWIG will then be executed to generate the wrapper files.
compileStep	The compile step compiles all the C or C++ code and creates the Webpay client API file.
clean	This target will clean all the files created by the build process.

* The PHP 5 API values have been used to illustrate the values

8.3 Appendix C – Glossary

WTS	Webpay Transaction Server