Web Commerce With Delphi: Live Transaction Processing

by Peter Hyde

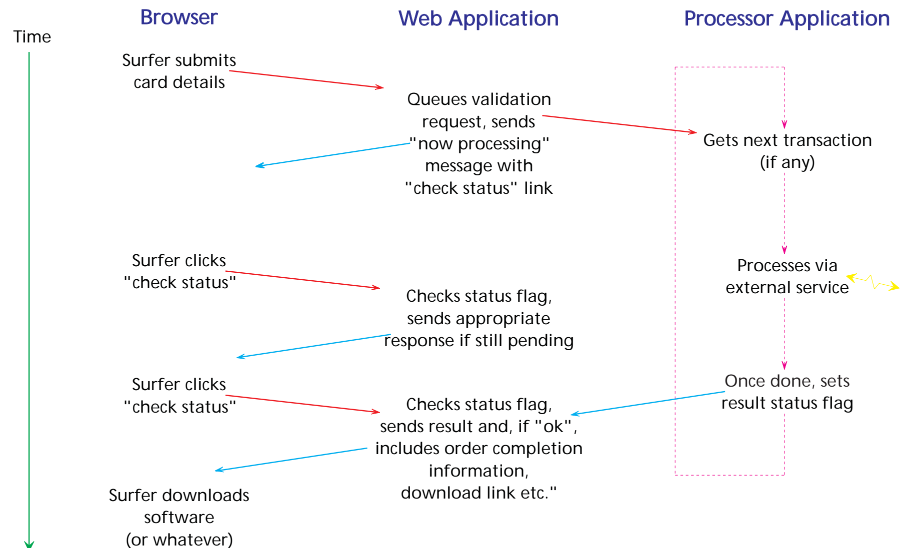
In my previous look at Web commerce (September 1997), I outlined the range of options already being used for handling transactions over the Web, and provided a working example of what is possibly the most common approach: validating credit card information on the site, but handling the actual transaction offline.

This approach is popular because it is relatively inexpensive to implement and is basically sufficient for all scenarios where there is going to be a manual 'dispatch' step in satisfying the order anyway. That manual step provides an opportunity for full bank authorisation of the credit card details, thus eliminating stolen, over-limit or spurious but numerically valid credit cards.

However, for a growing number of cases, a real-time, live authorisation and/or funds transfer step is required, either because the 'product' (such as software or information) is going to be delivered automatically and immediately from the site, or because the site vendor wants to eliminate the manual processing step.

Services that support such authorisations include ICVerify (www.icverify.com) and Cybercash (www.cybercash.com) in the US, and a range of bank-specific services elsewhere. For some countries, the service offerings are, as yet, so limited or restrictive that it may be easier to open a US merchant bank account and use one of the US-based services, in spite of the additional tax and bookkeeping complications which might be involved. An example of a service offering Merchant card accounts can be found at http://secure.href.com/merchant/.

Needless to say, the picture is changing. From the viewpoint of a software developer, this means that we should work hard to isolate the different steps involved in carrying out an online transaction so that, if the back-end service changes, the rest of our application is left relatively unchanged.



➤ *Figure 1*

Coping With Service Delays

Perhaps the most important aspect to consider when implementing support for online transactions is that 'real time' does not necessarily imply 'instant.' In any case where a third-party server or service must be involved, allowance must be made for the variation in service availability and throughput which can ensue.

For example, ICVerify, while inexpensive and robust, depends (at the time of writing) on a 1200-baud dial-up connection being made to their free phone number. This can take 15-30 seconds to complete, even assuming that the programs making and receiving the calls haven't got a queue of transactions to deal with. Even services which handle transactions via TCP/IP, like Cybercash, are subject to similar response times. Therefore, both the site's processing software and user interface should be based around an asynchronous model, effectively placing the surfer in a holding queue until the transaction is either accepted/rejected by the service or abandoned by the surfer.

Another desirable feature is to isolate the transaction processing and queue management from the dynamic Web application. This offers various advantages. Firstly, one service application can be used to handle transactions for a number of sites on the server. If traffic and server load increase, the transaction handler can be moved easily to a separate machine or suite of machines, communicating with the server via either a shared directory or more elaborate mechanisms such as DCOM. There is also no need to spawn threads in the Web application to manage the queuing and authorisation process.

Lastly, if the authorisation service used changes, only the standalone processor application needs

to change: the site software should not be affected.

Web App Requirements

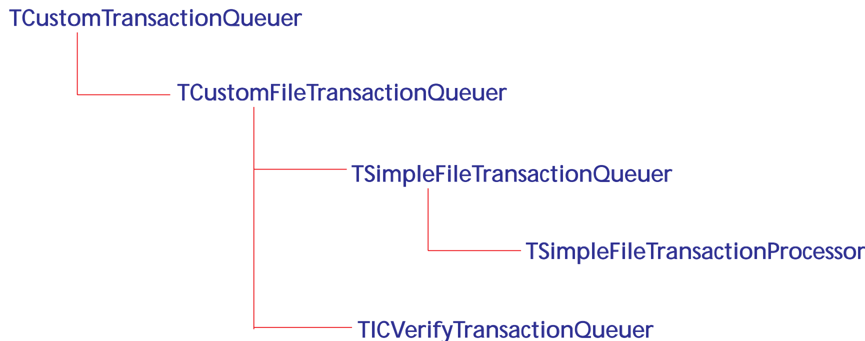
Figure 1 illustrates this kind of asynchronous transaction processing system. By offloading the transaction processing to the Processor Application, we've limited the Web application's automation requirements to four.

The first is to numerically validate the card number and expiry date (as done in the example application in my last article: there's no point making the surfer wait 30 seconds while you call the processing service to find out that the card number has been mistyped). The second is to queue a 'process this transaction' request and tell the surfer what is happening.

The third requirement is to respond sensibly to 'check processing status' requests from the surfer while the request is in the queue or still being processed. By using a `META REFRESH` tag in the HTML you can have the browser automatically 'poll' your Web application every few seconds if you like.

Lastly, when the transaction processing has finished, and has set a status flag which records the result, the app must use that flag to decide what page to display when the surfer next checks the status. Usually it would be an 'Order accepted' page, which may contain links to download the product or information ordered.

In no cases need the Web application hang around waiting for things to happen, it can immediately respond to surfer requests with either a 'queued', 'rejected' or 'completed' page.



Components in the WebTrans.pas Unit

➤ *Figure 2*

Queuing Components

We'll start our design with `TCustomTransactionQueuer`, an abstract class which defines handling which should apply regardless of what Web automation or transaction processing solution you intend to use.

Its properties and data types are shown in Listing 1 and the full component is in the WebTrans.pas file on this Issue's disk. Figure 2 shows an outline of the component hierarchy for the `WebTrans` unit.

The `TTransactionData` type encapsulates all the information which might be required to carry out a particular transaction. Other properties, such as `QueueByFile`, control the way the component itself behaves in the current application. A particular implementation derived from this component will use those fields or properties which are important for a given transaction-processing service.

`TCustomTransactionQueuer` does not actually queue anything anywhere, it leaves that work to derived components so that they can manage their own specific implementations.

➤ *Listing 1*

```
type
  TTransactionStatus = ( tsInvalid, tsQueue, tsCancel,
    tsProcessing, tsTimeOut, tsAccept, tsReject);
  TTransactionData = record
    // a unique identifier, eg surfer name, session ID:
    TransactionID: string;
    { May be used, eg if doing authorisations (not confirmed
      sales) with settlement happening LATER after shipping
      is confirmed: }
    TransactionType: string;
    // only needed if back end is multi-merchant enabled:
    MerchantID: string;
    // as above, assuming the back end requires a password:
    MerchantPassword: string;
    // available with some services, appears on form:
    Comment: string;
    Clerk: string;       // ditto
    CardNumber: string;  // no spaces or punctuation
    ExpiryMonth: string; // 01..12
    ExpiryYear: string;  // 98, 99, 2000, 2001...
    // total amount to be authorised or transferred:
    TransactionAmount: string;
    { Time when transaction was first queued, set by Queue
      not caller: }
    Age: TDatetime;
    // set by Queuer or returned from Processor:
    TransactionStatus: TTransactionStatus;
  end;
  // Abstract class which implements the most common logic:
  TCustomTransactionQueuer = class(TComponent)
  private
    fTransactionData: TTransactionData;
    // File/Memory require implementation in derived classes:
    fQueueByFile: Boolean;
  protected
    // if encryption is implemented:
    fEncryptionPassword: string;
    function LoadTransaction: Boolean;
    { derived classes must implement these methods to
      support File, Memory or both. The functions should
      return False if the Transaction is not present.
      The 'Save' procedures should recognise a tsCancel
      status and handle it accordingly, in most cases just
      by deleting the entry. }
    procedure SaveTransactionToFile; virtual; abstract;
    function LoadTransactionFromFile: Boolean;
       virtual; abstract;
    procedure DeleteTransactionFromFile; virtual; abstract;
    procedure SaveTransactionToMemory; virtual; abstract;
    function LoadTransactionFromMemory: Boolean;
      virtual; abstract;
    procedure DeleteTransactionFromMemory;
      virtual; abstract;
  public
    procedure QueueTransaction;
    procedure CheckTransaction;
    procedure CancelTransaction;
    procedure DeleteTransaction;
    // only publically used for Processor components:
    procedure SaveTransaction;
    function GetStatusMessage: String; virtual;
  published
    property TransactionData: TTransactionData
      read fTransactionData write fTransactionData;
    property StatusMessage:string read GetStatusMessage;
  end;
```

The next layer of our onion is the TCustomFileTransactionQueuer, a class which adds queue-by-file functionality to the mix. This is also an abstract class, leaving two methods which actually specify how the contents of the queue file are organised to be implemented by a derived class.

The TSimpleFileTransactionQueuer is the culmination of all this, albeit a rather simple one. It simply stores and retrieves the most essential information in comma-delimited form in the appropriate directory with a filename which ends in .TRN.

The Queuing Components

After placing TSimpleFileTransactionQueuer on our Web application's form, we can use it when responding to three possible events from the surfer: queuing, checking and (optionally) cancelling a transaction request. Listing 2 shows pseudocode for queuing a request.

Checking a request which has already been queued is exactly as for queuing, except that possible return status values (and hence page handling) should also include 'Processing', 'Accepted' or 'Rejected'. Note that the CheckTransaction method just calls QueueTransaction anyway, because our implementation includes a failsafe mechanism whereby, if a transaction is lost from the queue for any reason, the surfer's next check will simply re-queue it. An ideal implementation, such as the example application introduced later, should be able to handle stopping and restarting of both the Web application and the Processing application without loss of 'state'.

Listing 3 shows the pseudocode for cancelling a request.

The logic for assigning literals and sending the appropriate response page for each situation depends on what Web development components you are using. However, the rest should be entirely generic, perhaps only varying if you switch card processing services and find that the new one requires an additional TransactionData field to be supplied (eg MerchantID).

```
with TransactionQueuer, TransactionQueuer.TransactionData do
begin
  AssignTransactionDataFromFormLiterals;
  // need at least TransactionID, CardNumber, ExpiryMonth,
  // ExpiryYear,  TransactionAmount
  QueueTransaction;
  SendPageBasedOnStatus(TransactionStatus); // "Queued" or "Problem"
  if TransactionStatus in ([tsTimeout, tsInvalid, tsCancel, tsAccept, tsReject])
    then DeleteTransaction;
end;
```

➤ *Listing 2*

```
with TransactionQueuer, TransactionQueuer.TransactionData do begin
  TransactionID := value from literals // (eg session ID, surfer name)
  CancelTransaction; // could fail if, for example, it has just
    // entered "processing" mode
  SendPageBasedOnStatus(TransactionStatus); // all possibilities are open
    // since all possibilities are open, must clean up if necessary,
    // even though a Cancel would normally delete the transaction anyway:
  if TransactionStatus in ([tsTimeOut, tsInvalid, tsCancel, tsAccept, tsReject])
    then DeleteTransaction; // harmless if already done
end;
```

➤ *Listing 3*

The TRNQUEUE.DPR project is a non-Web demonstration of transaction queuing which uses this component. Be sure to install the components in WEBTRANS.PAS onto your component palette before loading it.

The Processor Application

For some services, a processor (back end) element might not even be necessary. For example, if you purchase and run the standard ICVerify program for Windows, it will look for transaction files created in its documented ASCII format and then carry out the transaction. Thus, in effect, the processor application has already been made for you: just use the TICVerifyTransactionQueuer component which you will find at the end of the WEBTRANS.PAS unit (the TrnQueue project has a Use ICVerify checkbox which makes it do just that). However, if you want to use a local processing service, Cybercash, or even ICVerify via its DLL instead of its standalone Windows product, you'll very likely be making your own processor application to act as an interface between your queuing mechanism and the processing service's API.

To make this task easy, we've derived a model TSimpleFileTransactionProcessor component from our Queueing component. The extra logic included is mainly the GetNextTransaction function which looks for any pending transactions in the queue (ignoring those which have already been processed but not yet deleted).

A sample application which uses the processor component is provided in TRNPROC.DPR. A TTimer fires every five seconds to check and, if necessary, clear the queue. This sample project substitutes user confirmation for the calls to a processing service which you'd add when you take it 'live'. Leave it running while you experiment with the TrnQueue project mentioned above, and you'll have all the elements of the queue/process logic ready for testing.

A Live Web Application

If you're in a hurry to get your live Web commerce solution online, I've revised the sample shopping cart application from my September 1997 article to include transaction queueing using the TICVerifyTransactionQueuer component. Because this application is built with the WebHub component framework for Delphi, you'll need to download a trial version of WebHub from www.href.com.

After installing WebHub, you'll find its original "SHOP1" application in the \ht\htdemos\codedemo\shop\dpr directory. Make sure that you can load, compile and run this sample application successfully with WebHub, then, from this Issue's disk, copy

HTSHOP*.* into the above directory. Also copy SHOP1.TXT into your Web server's htdocs\htdemo\WhubHtml\shop1 directory. Reload the HTSHOP1.DPR application in Delphi and set the `QueueDirectory` property for `ICVerifyTransactionQueuer` to point wherever ICVerify is expecting it to be (we're assuming you have ICVerify installed, but it isn't compulsory if you're just testing the Web logic). Now compile and run it as you did before, noting the new credit card entry, checking and queuing logic.

If you wish, you could readily replace the ICV queuing component with a `TSimpleFileTransactionQueuer` component and use the TRNPROC.DPR project to simulate the service responses instead. The logic at the Web application's end would be unchanged, just as the `TrnQueue` project can switch what queuing component it uses with a simple runtime checkbox.

To extend or apply this sample application, you'll want to review the comments in HTSHOPC.PAS. If you plan on using ICVerify as a processing service, pay close attention to the comments about it in WebTrans.pas.

Further Reading

Start by reading the *Banking Terms And Caveats* box, as it covers some of the teminology and methods used in online banking. While this information is based on the approach generally used in the US, it is likely to apply no matter where you are.

For more information on Cybercash, visit www.cybercash.com. Currently, there is no merchant signup fee and the per-transaction fee is 10 cents (US), included in the card processing fees you pay to your bank. Some pre-rolled WebHub code for calling a Cybercash Perl script to process transactions can be found at

```
www.href.com/scripts/
  runisa.dll?TN:Detail::194
```

You could perhaps use the code as a basis for a `TCyberCashTransactionQueuer` component, with the Perl script acting as the processor application.

For more information on ICVerify, visit www.icverify.com. Setup costs are about $US400 for the Windows version for a single merchant, but the $US750 SDK/partnership program is highly recommended, due to the additional technical information and bulletins. There is no per-transaction fee with ICVerify.

Where To From Here?

What you do with the components from here on will be tightly guided by the implementation requirements of the card processing service you use. Issues such as field validation, which fields to use, whether to log transactions for later settlement and so on cannot be explored in detail here because we would rapidly head down a path which is too tightly tied to one service. As a look at the WEBTRANS.PAS source will reveal, even the `TICVerifyTransactionQueuer` component is only using a narrow range of the overall ICVerify service capabilities, and for particular client (or multi-client) requirements, you may want to extend its handling significantly.

However, with a defined, generic transaction queuing approach, you can play about with implementation details pretty much to your heart's content and your Web applications will keep on working.

Peter Hyde is the author of the TCompress and TCompLHA component sets for Delphi and C++Builder, and Development Director of South Pacific Information Services Ltd, which specialises in dynamic Web site development and automation. Peter can be contacted at peter@spis.co.nz or via http://www.spis.co.nz. Thanks to Ann Lynnworth of HREF Tools Corp for her assistance in the preparation of this article.

Banking Terms And Caveats

Complete Sale. You carry out this kind of transaction when you know you will ship within 24 hours (in the US it's illegal to charge a person's card if the product is not shipped that day).

Book. You do this when you want to hold the customer's funds and plan to ship later. The customer's funds are held for about 10 business days. You must save the approval code in order to follow up with the next step.

Ship. You do this when the goods have shipped, meaning that the customer funds can then be released to you.

Regardless of whether you do the above in one step (Complete Sale) or two (Book, then Ship), you *must* run a settlement transaction if you want to get your funds, at least for most US merchant account types. If you do not settle within 10 days, you have to start all over again and see whether the customer still has funds with which to pay you.

Note that there are a *lot* of different types of card processing networks. The ICVerify product lists over 20. Each one has different protocols. If you are only going to support a single merchant, you have a lot of flexibility in choosing your back-end service provider, because you are just checking to be sure they support one processor.

However, if you are more like an ISP, you need to pick someone (like ICVerify) that has support for 99% of the processing networks, all encapsulated into one transaction interface. They have done an enormous amount of work to make that true: they have 90% of the card processing business in the US and they have been around for over a decade, whereas some of the other processing companies may be a little shaky as the market and technology settles.

Finally, look before you leap. Financial transactions are a non-trivial issue in any language or locale, and you should not assume you have all the information you need on the basis of having read these articles. By all means start from here, but do your homework too. See you on the Web!