Screen-Scraping vs. CONNX
by Bill Luton, Development Manager

So, you’ve been tasked with building a system that extends your existing data to the Web, and you already have legacy applications running on a mainframe that do pretty much everything you want your new Web application to do. You’re thinking that perhaps using a screen-scraper to leverage your existing application might be the way to go.

What is a screen-scraper?
Screen-scrapers are a natural extension of terminal emulation. Their purpose is to extract data from a data stream that was intended to go to a display terminal. If you think about it, a host application is made up of a sequence of screens, with defined keystrokes that specify how to advance from one screen to the next. For example, the Enter key is often defined to tell the application that it is time to move to the next screen in a sequence. The F1 key might tell it to display Help text.

Similarly, a screen-scraper has a few basic functions it needs to be able to perform: It must be able to parse the data stream coming from the host to determine where input and output fields are; it must be able to send key codes to the application to move from one screen to the next; and it must be able to insert data into the data stream in cases where the host application requires input.

Early screen-scrapers used an interface called HLLAPI (High Level Language Application Programming Interface). HLLAPI was used in conjunction with an IBM® 3270 or 5250 terminal emulator. HLLAPI was very useful for automating tasks and extracting data from particular screens for use on the Web or Windows-based applications. Newer packages on the market have eliminated the need for HLLAPI by hiding the terminal emulator and exposing methods through Java Beans™, Web Services, or Microsoft® .NET objects. These new packages often handle navigation issues as well. So, for example, a Web programmer doesn’t have to know which key to press to get the application to a particular screen.

But even with this new packaging, the principle is still the same. The screen-scraper must start at some predefined point in the application, such as the logon screen, use inputs to navigate its way to whatever screen contains the desired data, and then use predefined position information to “scrape” the data out of the data stream. Once that’s done, it must navigate its way back to the starting point again, so the next time it is accessed, the driving program knows where it is in the host application.

Sound too good to be true?
The ability to completely reuse existing business logic does sound tempting on the surface. Companies that sell screen-scrapers often make such claims as, “Use your existing applications in new ways without making any changes,” or “Effortlessly interact with your mainframe data on the Web.” The truth is, screen-scraping is a cumbersome process at best. In order to access the data, you need a programming staff that has intimate knowledge of the host application to which the data is tied. If you are using HLLAPI, you must intimate knowledge of the host application to which the data is tied. If you are using HLLAPI, you must provide a map so it knows how to get to the screen your data is on. And what if the path to that screen is different depending on input? Then you must provide a map or write code for all possible outcomes of pressing the Enter key.

For example, suppose you have an insurance application that takes you to the customer information screen when you input the policy number if the number represents a policy holder of one type, but
takes you to a different screen if the number represents a policy holder of another type? All these permutations must be accounted for. And don’t forget those pesky host-initiated screens. What happens when your application is navigating merrily along and someone sends out a break message reminding everyone the system will go down at midnight for routine maintenance? The user it was designed for would simply dismiss the screen and continue. Let’s hope you remembered to tell your screen-scraper about it.

Continuing with this scenario, let’s say you wanted to have a user enter a policy number on a Web page and have it return personal information to the user, such as name, address, and phone number as well as specifics about their policies. The screen-scraper would start at the logon screen and log on with a user ID and password that had privileges on the mainframe to run the specified application. It would then need to navigate to the menu or screen necessary to start the application. It would then find the correct position in the data stream to insert the input value which, in this case, is the policy number, and then it would send an Enter key to the host. The host would then apply its business logic to this input data and return whatever screen it is supposed to.

Now let’s say it’s a menu with options that can be selected for the policy. The screen-scraper would then need to input the option for customer information and send that to the host. It would then wait for the screen with that data to return. Once the correct screen returns, the screen-scraper would (finally) go to the predetermined positions on the screen and extract the data it was interested in. It would then send the correct keystroke back to the host to get it back to the menu, select the menu for the policy information, and continue navigating around the host application until it had completed its task. It would then navigate back to the starting screen. (Yes, you have to teach it how to get back as well.) All this navigation can be very time-consuming. Even if the average response time between each screen was one second, and it only had to navigate ten screens, this one simple transaction would take ten seconds! How well will this scale in a large transaction-per-minute environment?

Did I just mention transactions? Not only do transactions need to happen quickly in most environments, they also need to be able to be rolled back when something goes wrong. What happens when a transaction involves inserting data into three different database tables and the third one fails? If you are using SQL, you can choose to simply rollback the first two inserts. If the inserts were done with a screen-scraper, how will you roll them back? Yes, you have access to the business rules, but you do NOT have access to the backend database. You are at the mercy of only being able to do only what the host application was designed to do.

Now suppose you have a Web application that gets data from a legacy mainframe host application. It navigates the screens and returns the data. You’ve decided that you don’t care if it trips up at times or if it doesn’t scale particularly well. You can also live with the whole transaction issue. You bundle up your shiny new application and show it to your users and, of course, their eyes light up and they start asking what else it can do? Wouldn’t it be cool if you could integrate data from Oracle? Well, yes, but you don’t have a mainframe application that exposes that data in the way the user asked for.

What about using your newly developed screen-scrapping skills to write an application that can load the data warehouse? Well, you suppose you could, but it would be slow and you could only move data that is exposed by the application.

What about changing the business logic behind the scenes to get different data? Yes, you could hire a mainframe programmer to make those changes, but then you would need to update all your slick navigation code because the screens on the host application changed. You could do that. It’s only time and money after all.

Enter CONNX

CONNX is a robust, highly scalable distributed SQL Engine that takes advantage of resources on both the backend host and the middleware server or
client PC. With CONNX, you don’t have any navigation issues to deal with because CONNX connects directly to the backend database. You have direct, heterogeneous access to several databases, including, but not limited to, VSAM, C-ISAM, RMS, DB2, and Adabas.

CONNX provides native OLE DB support as well as a .NET data provider so you can easily incorporate data from all your legacy systems into your favorite OLE DB- or .NET-aware development environment. CONNX also supports transactions with full rollback and commit capabilities, so there’s never an issue if part of a transaction fails. Furthermore, because CONNX directly accesses your legacy data, the issues that can arise if the backend application changes or if you want to do more creative things with your new Windows application disappear.

Let’s say you’ve built your new Web application based on some existing functionality that existed in a legacy mainframe application. Now that you and your staff see the benefits of your new application, your imaginations kick in and you start to think of all the other things you could do with your data. As I mentioned before, if you are using screen-scraping, you are bound to using the data in the way your legacy application exposes it. With CONNX, you can use it any way you want. You can access all your data in all your systems, not just the data exposed in the mainframe application.

Because CONNX allows you to heterogeneously access data from different platforms, you can actually perform joins across different data types. For example, you can do a join on a table from DB2 and another table from VSAM! Try that with a screen scraper.

With CONNX, you can use its SQL interface and seamlessly plug into any application, report writer, or development tool that is ODBC-, JDBC-, OLE DB- or .NET-compliant. This means you have direct access to your data from your favorite reporting or development tool. Direct access enables you to have the freedom to build your new application with modern Web development tools, such as WebLogic, WebSphere or .NET. And you no longer have to worry about security issues: With CONNX, you are in control of who has access to the data. Highly scalable, direct access to your data vs. time-consuming screen-scraping and navigation that only enables you to access data exposed by an application that was never meant to do what you are trying to do? Which method would you choose?

CONNX is the clear choice.