Introduction to

Web Programming with RPG

Presented by

Scott Klement

http://www.scottklement.com

Session 11C

"There are 10 types of people in the world.
Those who understand binary, and those who don't."

Why Why Why WWW?

Why is it important to learn about Web programming?

- Users are demanding graphical applications.
- Client/server applications are complex and expensive to maintain.
- Web applications are graphical, yet relatively simple to build and maintain.
- Nothing to install on the PC.
- Everyone already has web access from their desks.
- Easy to deploy applications to the "entire world" if needed.
- Easy to connect your applications to those of other companies.

Many people don't even know that you can write Web applications in RPG!

Why RPG? Isn't Java or PHP Better?

- In many System i shops, there's a lot of RPG talent, and most of the existing business rules are written in RPG.
- Evolution, not revolution! It's expensive and time consuming to learn an entirely new language and way of thinking.
- Java, especially when used through WebSphere requires more hardware resources than RPG does.
- Many shops, especially small ones, do not need the added features of WebSphere/PHP, and it's not worth the added complexity.
- It's easy to get started with Web programming in RPG. If you find that you need more, go right ahead and upgrade. In that case, this'll just be a stepping stone to the future.

Two Aspects of Web Programming

Web programming has two uses:

- Providing web pages for a user to display with a browser.

  We're all familiar with this, it's what we see every day when we're out surfing the web.

- A means of communication between applications.

  Companies can work together to integrate their services into each other's applications.

HTML Overview

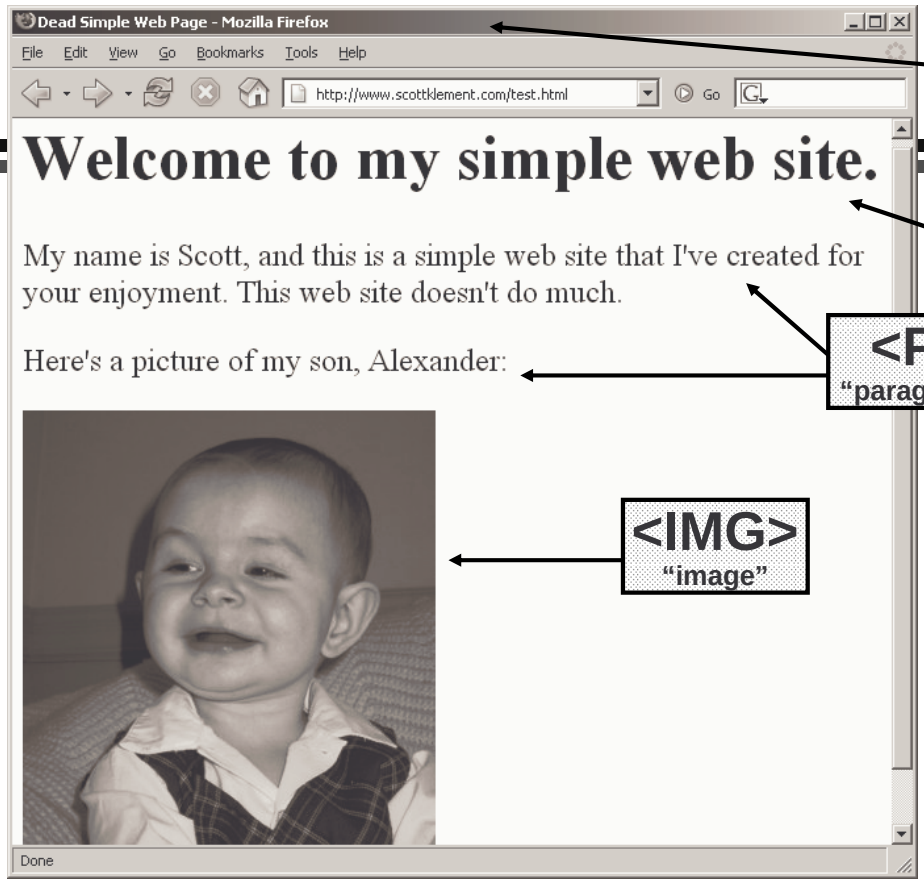
This presentation does not intend to teach HTML in it's entirety, only to give you a basic introduction to it.

- Simple text data written to a file.
- Special "tags" modify the way the data is displayed (as a title, heading, paragraph, etc.)

```
<html>
   <head>
      <title>Dead Simple Web Page</title>
   </head>
   <body>
       <h1>Welcome to my simple web site.</h1>

       <p>My name is Scott, and this is a simple web site
        that I've created for your enjoyment.  This web
        site doesn't do much.</p>

       <p>Here's a picture of my son, Alexander:</p>

       <img src="http://www.scottklement.com/AlexDressUp.jpg" />
    </body>
</html>
```

Start tags look like this: <head>

End tags look like this: </head>



What Happened?

The URL in the browser's "address" field told it which document you wanted:

`http://www.scottklement.com/test.html`

http:	//www.scottklement.com	/test.html
The protocol of the web.	Server to connect to.	Document to download.

The browser took these steps:

- Connect to the HTTP server (port 80) on www.scottklement.com
- Ask the server for "/test.html"
- The server's reply contained the HTML document.
- Browser renders the HTML document on the screen.
- In doing that, it sees the request for another URL.

`http://www.scottklement.com/AlexDressUp.jpg`

The process is repeated to get this picture. Since it this one is a picture, it displays it where the <IMG> tag was.

What About On-The-Fly Data?

The last example dealt with data that's always the same. The HTML document and picture are created once, and when the browser wants them, they're downloaded.

But, what if you have data that's not always the same? Perhaps you have a database that's constantly changing – and you want the user's request to show the current state of that data?

- Instead of a URL that points to a disk object to download, have it point to a program to run.
- When run, the program can perform any database access or calculations that it needs to, and then return the HTML.
- The freshly generated HTML can be sent to the browser.

Introduction to CGI

CGI = COMMON GATEWAY INTERFACE

This is a specification for how an HTTP server:

- Can run a program
- Receive input information from the HTTP server
- Write the results back to the HTTP server so the server can send them back to the browser.

`http://www.scottklement.com/cgi-bin/test.pgm`

In the server config:

- You designate **`/cgi-bin`** as a "script alias". This tells the server that when a request is made to something in that directory, it's a program that should be run rather than a document to download.

Sample Apache Config

```
ScriptAlias /cgi-bin /QSYS.LIB/WEBAPP.LIB
<Directory /QSYS.LIB/WEBAPP.LIB>
   Order Allow,Deny
   Allow From all
</Directory>
```

In traditional naming, accessing a *PGM object named TEST in library WEBAPP would look like this:

`WEBAPP/TEST`

However, in IFS style naming, you access the same object with:

`/QSYS.LIB/WEBAPP.LIB/TEST.PGM`

Notes:

- This is just excerpt from a larger config file. It only depicts the settings for requests to **`/cgi-bin`**.
- **`ScriptAlias`** maps **`/cgi-bin`** to the **`WEBAPP`** library (IFS naming style)
- **`ScriptAlias`** not only maps one to the other, it also tells the server that it should CALL the object rather than download it.

Sample "Original HTTP" Config

Currently, Apache is the recommended server for the iSeries. The Original server will not run on V5R3 and later.

However, if you are still using the original server, you use the EXEC directive instead of **`ScriptAlias`**

```
Exec /cgi-bin/* /QSYS.LIB/WEBAPP.LIB/*
```

Notes:
- Same as the previous slide.
- Maps **`/cgi-bin`** to the **`WEBAPP`** library.

For Either Server

```
http://www.scottklement.com/cgi-bin/pricelist.pgm
```

Will now run the RPG program called **`WEBAPP/PRICELIST`**

Standard Output (1/2)

You now know that a request for /cgi-bin/pricelist.pgm will run a program called WEBAPP/PRICELIST. That program will read a price list database, and will use it to generate HTML code for the browser on-the-fly.

To send it to the HTTP server, an RPG writes it's output to a standard stream called "Standard Output". *(or, "stdout" for short)*

What is Standard Output?
- Commonly used in C programming, Unix programming, MS-DOS programming, Java programming. Also QSHELL and PASE on the iSeries.
- On those systems, every program has a standard output that normally writes to the screen.
- Not traditionally used in RPG, but it can be.
- The output can also be redirected to a file.
- The output can be redirected to a pipe that connects it to another program.

That's how your RPG program sends data to the HTTP server – by sending it to standard output. When the server ran your program, it connected a pipe so that it'll be able to read the standard output data as you're writing it.

Standard Output (2/2)

IBM provides the QtmhWrStout() API as a means of writing data to standard output.

This shows the parameter summary listed in the Information Center for this API:

Required Parameter Group:

1	Data variable	Input	Char(*)
2	Length of data variable	Input	Binary(4)
3	Error Code	I/O	Char(*)

The QtmhWrStout API provides the ability for CGI programs that are written in languages other than ILE C to write to stdout.

Required parameter group

Data variable

Input:CHAR(*)

The input variable containing the data to write to stdout.

Length of data variable

INPUT;BINARY(4)

Here's a matching RPG prototype:

```
D QtmhWrStout     PR                  extproc('QtmhWrStout')
D   DtaVar                      32767A   options(*varsize) const
D   DtaVarLen                     10I 0 const
D   ErrorCode                    8000A   options(*varsize)
```

PriceList Program (1/4)

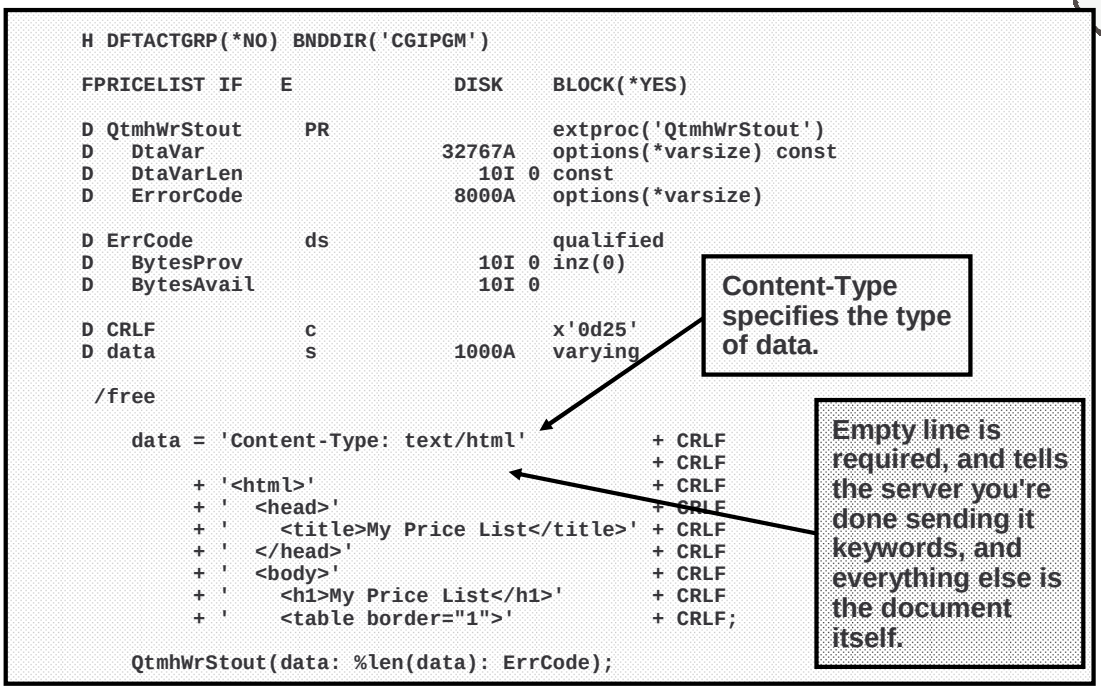
For example, you might have a price list that you want to publish to the Web. The prices are stored in the following physical file:

```
A            R PRICELISTR
A              ITEMNO         5P 0
A              DESC          25A
A              PRICE          5P 2
```

For an RPG program to process this, it'd have to:

- Tell the server what type of data it's returning (HTML in this case, and not Image, XML, Word Doc, etc.) by writing it to standard out.
- Send "header information" (HTML for the top of the page) to stdout.
- Loop through the PRICELIST file and send each price to stdout.
- Send "footer information" (HTML for the bottom of the page) to stdout.

PriceList Program (2/4)

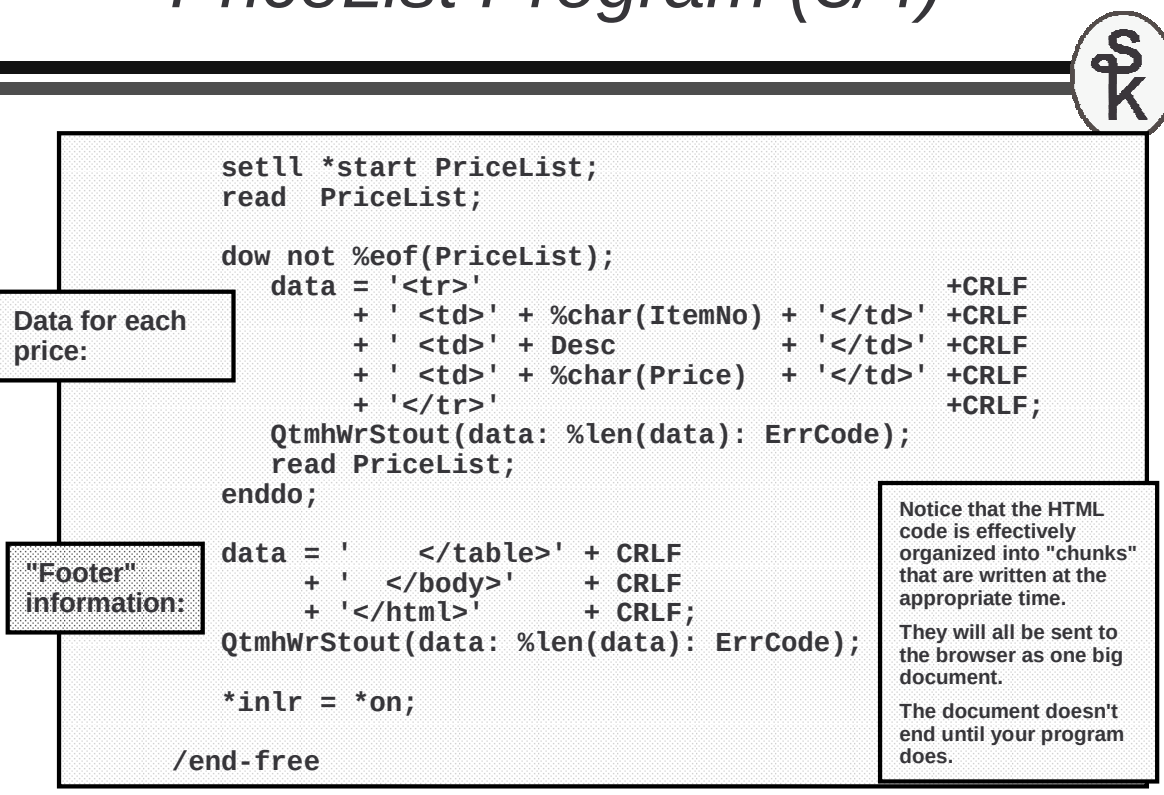


```
H DFTACTGRP(*NO) BNDDIR('CGIPGM')

FPRICELIST IF   E             DISK    BLOCK(*YES)

D QtmhWrStout     PR                  extproc('QtmhWrStout')
D   DtaVar                    32767A  options(*varsize) const
D   DtaVarLen                   10I 0 const
D   ErrorCode                 8000A   options(*varsize)

D ErrCode         ds                  qualified
D   BytesProv                   10I 0 inz(0)
D   BytesAvail                  10I 0

D CRLF            c                   x'0d25'
D data            s             1000A varying

 /free

    data = 'Content-Type: text/html'          + CRLF
                                              + CRLF
         + '<html>'                           + CRLF
         + '  <head>'                         + CRLF
         + '    <title>My Price List</title>' + CRLF
         + '  </head>'                        + CRLF
         + '  <body>'                         + CRLF
         + '    <h1>My Price List</h1>'       + CRLF
         + '    <table border="1">'           + CRLF;

    QtmhWrStout(data: %len(data): ErrCode);
```

Content-Type specifies the type of data.

Empty line is required, and tells the server you're done sending it keywords, and everything else is the document itself.

PriceList Program (3/4)



```
        setll *start PriceList;
        read  PriceList;

        dow not %eof(PriceList);
           data = '<tr>'                                +CRLF
                + ' <td>' + %char(ItemNo) + '</td>' +CRLF
                + ' <td>' + Desc          + '</td>' +CRLF
                + ' <td>' + %char(Price)  + '</td>' +CRLF
                + '</tr>'                               +CRLF;
           QtmhWrStout(data: %len(data): ErrCode);
           read PriceList;
        enddo;

        data = '    </table>' + CRLF
             + '  </body>'    + CRLF
             + '</html>'      + CRLF;
        QtmhWrStout(data: %len(data): ErrCode);

        *inlr = *on;

      /end-free
```

Data for each price:

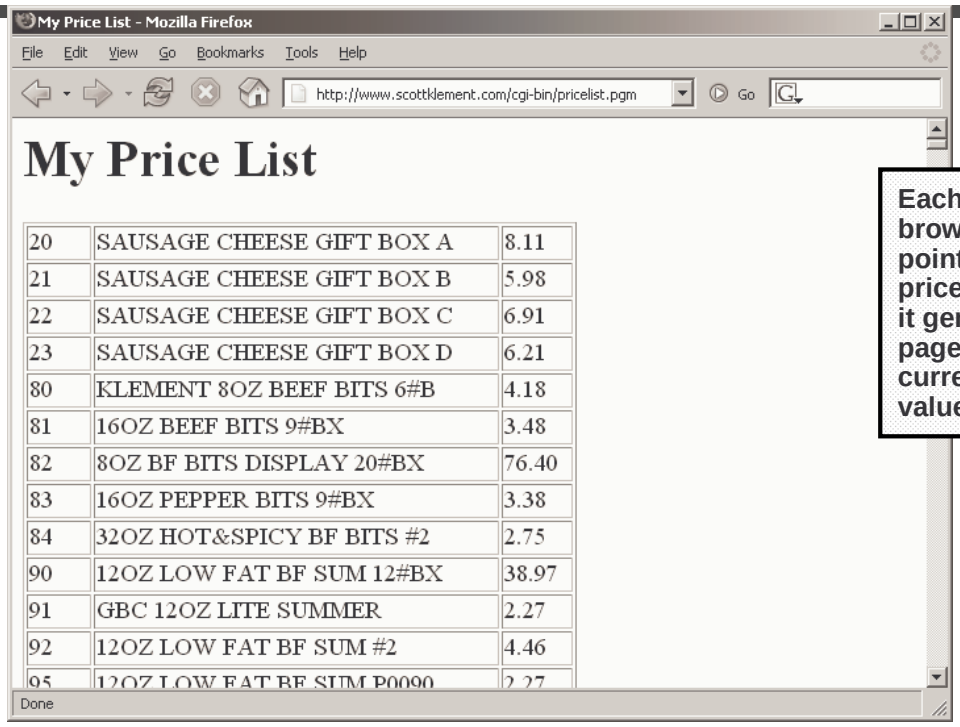
"Footer" information:

Notice that the HTML code is effectively organized into "chunks" that are written at the appropriate time.

They will all be sent to the browser as one big document.

The document doesn't end until your program does.

PriceList Program (4/4)



Each time the browser is pointed at the pricelist program, it generates this page with the current database values.

CGIDEV2

CGIDEV2 is a *FREE* tool from IBM

- Originally written by Mel Rothman (ex-IBMer)
- Written entirely in RPG.
- Includes source code and lots of examples
- Now supported (billable) from IBM's Client Technology Center (CTC)

CGIDEV2 can be downloaded from the following link:
http://www-03.ibm.com/systems/services/labservices/library.html

CGIDEV2 provides tools to simplify writing CGI programs:

- Take the HTML out of the RPG code, and put it in a separate member.
- Divide HTML into chunks (or "sections")
- Provide strings that are replaced with data from a program (or "HTML variables")

This means that you can:

- Develop your HTML in a separate (HTML design) tool
- Or just type them in Notepad or EDTF!
- Focus on "how things look" separately from focusing on "business logic".
- Get a college "whiz kid" or "web designer" to do the design while you focus on the business rules.

Sample Template File

```
/$Header
Content-Type: text/html

<html>
  <head>
    <title>My Price List</title>
  </head>
  <body>
    <h1>My Price List</h1>
    <table border="1">

/$EachPrice
  <tr>
  <td>/%ItemNo%/</td>
  <td>/%Desc%/</td>
  <td>/%Price%/</td>
  </tr>

/$Footer
    </table>
  </body>
</html>
```

This file will be entered into a PC tool like Notepad, then copied to the IFS of my iSeries.

The following are "section dividers" that separate the different chunks of HTML:

/$Header
/$EachPrice
/$Footer

The following are "variables" that will have data supplied by the RPG program:

/%ItemNo%/
/%Desc%/
/%Price%/

If you're able to have someone else do the design work, you'd simply take their HTML, slice it into sections, and insert the variables.

Then you could use their HTML w/CGIDEV2

Price List w/CGIDEV2

```
H DFTACTGRP(*NO)
H/copy hspecsbnd
FPRICELIST IF   E           K DISK
D/copy prototypeb

 /free
   SetNoDebug(*OFF);
   gethtmlIFSMult('/scotts_templates/pricelist.tmpl' );

   wrtsection('Header');

   setll *start PriceList;
   read PriceList;

   dow not %eof(PriceList);
       updHtmlVar('ItemNo': %char(ItemNo));
       updHtmlVar('Desc'  : Desc );
       updHtmlVar('Price' : %char(Price));
       wrtsection('EachItem');
       read PriceList;
   enddo;

   wrtsection('footer');
   wrtsection('*fini');
   *INLR = *ON;
 /end-free
```

/copy members provided by CGIDEV2.

Load the HTML template.

Write sections and update variables as required.

When done, write the special *FINI section. This tells CGIDEV2 that you're done.

Input From the Browser

So far, all of the examples have focused on writing output from your program to the Web. For obvious reasons, you sometimes want to get input from the user sitting at the browser.

The way you do this is with the <form> and <input> HTML tags. These create "blanks" on the screen where the user can type.

The <form> tag shows where the start & end of the form is, as well as telling the browser where to send the input. The <input> tag represents an input field or graphical device that gets input from the user. (text=field to type text into, submit=button for submitting form.

Input Example HTML

```
<html>
<body text="black" link="blue" bgcolor="white">
  <form action="/cgi-bin/custords.pgm" method="post">
    <h1>Show Customer's Orders</h1>
    <table border="0">
    <tr>
      <td align="right">Customer number:</td>
      <td align="left"><input type="text" name="custno" maxlength="8" /></td>
    </tr>

    <tr>
      <td align="right">Start Date:</td>
      <td align="left"><input type="text" name="start_date" maxlength="10" /></td>
    </tr>
    <tr>
      <td valign="top" align="right">Status:</td>
      <td align="left">
          <input type="radio" name="status" value="O" checked />Open<br />
          <input type="radio" name="status" value="R" />Processing<br />
          <input type="radio" name="status" value="I" />Invoiced<br />
          <input type="radio" name="status" value="D" />Delivered<br />
          <input type="radio" name="status" value="P" />Paid
      </td>
    </tr>
    </table>
    <input type="submit" value=" Ok ">
   </form>
</body>
</html>
```

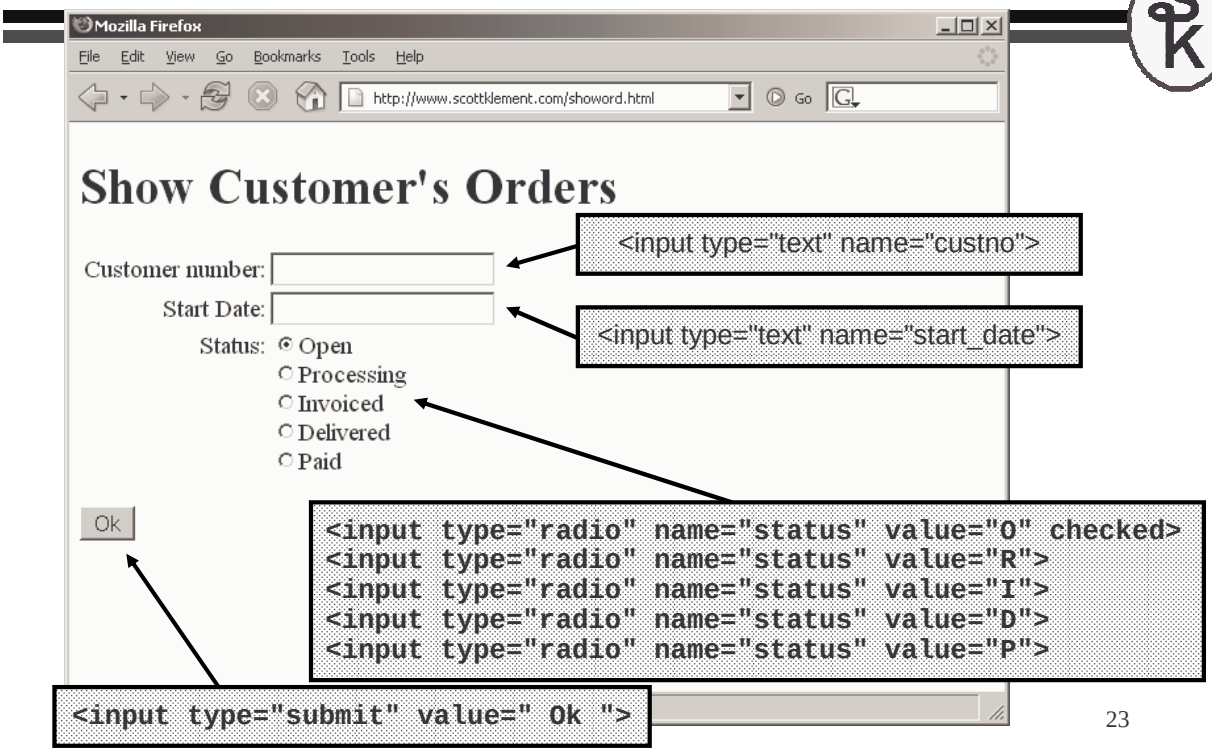
The browser will send the form's output to **/cgi-bin/custords.pgm**

The input type="text" tags are blanks for the user to type into.

Type="radio" declares a radio button.

Type="submit" declares button to click to submit the form.

What the Form Looks Like



What the Form Submits

When the browser sends form data to an HTTP server, it encodes it. The data that's actually submitted by the form looks like this:

```
custno=12345678&start_date=01%2F01%2F2006&status=P
```

- Each variable submitted is separated from the others by the & symbol.
- Each variable is separated from it's value with the = symbol
- Any spaces are converted to + symbols
- Any characters that would have special meanings (such as spaces, &, +, or =) are encoded as % followed by the hex ASCII code for the character.

If you wanted to handle these variables in your code, you'd have to write a routine that converted it back to normal. Or, you'd need to call an API that does that for you.

Fortunately, CGIDEV2 makes it easy.

- When you call the zhbGetInput() routine, it reads all the variable info from the browser, and parses it for you.
- You can then call the zhbGetVar() API each time you want to know a variable's value.

Order List Example (1/4)

```
/$Header
Content-Type: text/html

<html>
  <head>
    <title>List of Orders</title>
  </head>
  <body>
    <h1>List of Orders</h1>
    <table border="1">
 <tr>
  <td><b><i>Order Number</i></b></td>
  <td><b><i>Amount</i></b></td>
  <td><b><i>Delivery Date</i></b></td>
 </tr>

/$Order
  <tr>
  <td align="left">/%OrdNo%/</td>
  <td align="right">/%Amount%/</td>
  <td align="right">/%Date%/</td>
  </tr>

/$Footer
    </table>
  </body>
</html>
```

```
/$Error
Content-Type: text/html

<html>
  <body>
     <font color="red">
        /%ErrMsg%/
     </font>
  </body>
</html>
```

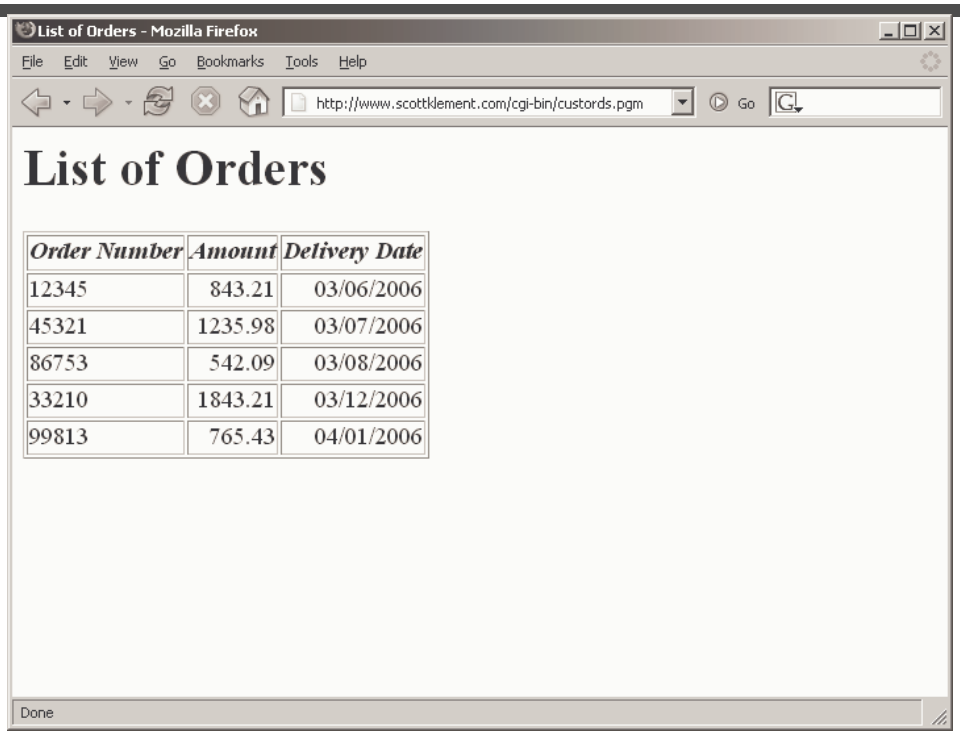

Order List Example (2/4)

```
     H DFTACTGRP(*NO)
      /copy hspecsbnd
     FORDDATA     UF   E           K DISK
      /copy prototypeb
      /copy usec

     D savedQuery      s          32767A   varying
     D custno          s             8A
     D date            s              D
     D status          s             1A

      /free

         SetNoDebug(*OFF);
         gethtmlIFSMult( '/scotts_templates/custords.tmpl' );

         qusbprv = 0;
         ZhbGetInput(savedQuery: qusec);

         custno = zhbGetVar('custno');
         status = zhbGetVar('status');

         monitor;
            date = %date(zhbGetVar('start_date'): *USA/);
         on-error;
            updHtmlVar('errmsg': 'Invalid start date!');
            wrtsection('error');
            return;
         endmon;
```

Order List Example (3/4)

```
      wrtsection('heading');

      setll (custno: status: date) ORDDATA;
      reade (custno: status) ORDDATA;

      dow not %eof(ORDDATA);
         updHtmlVar('ordno': OrderNo);
         updHtmlVar('amount': %char(OrderTot));
         updHtmlVar('date': %char(DelDate:*USA/));
         wrtsection('order');
         reade (custno: status) ORDDATA;
      enddo;

      wrtsection('footer');
      wrtsection('*fini');
      *inlr=*on;

   /end-free
```


Order List Example (4/4)



Web Applications

Closing Thoughts

Web applications represent a simple way to put a GUI face on your RPG programs. There's still the following caveats:

- Most programs need to be re-written to use this.
- If your code is modular so that the business logic is separate from the display logic, you may only have to re-write part of it.
- Programs that accept input once, and output once will convert easily. For example, reports.
- CGIDEV2 is free, and it's easy to try Web programming and experiment with it.

Web Services (1 of 2)

A Web service is a way of calling programs from other programs. It's very similar in concept to a CALL command (**`CALL PGM(GETRATE) PARM(&PARM1 &PARM2)`**) except that it makes the call over the World Wide Web.

This is different from CGI because:

- Instead of taking input from an HTML form in a browser, it accepts an XML document from another program.
- Instead of writing out HTML data to a browser, it writes out XML data for another program to read.

Imagine being able to call a program on another company's computer! Even if that company is on the other side of the world!

Think of some of the things you could do...

Web Services (2 of 2)

Imagine some scenarios:

- You're writing a program that generates price quotes. Your quotes are in US dollars. Your customer is in Germany. You can call a program that's located out on the Internet somewhere to get the current exchange rate for the Euro.
- You're accepting credit cards for payment. After your customer keys a credit card number into your application, you call a program on your bank's computer to get the purchase approved instantly.
- You've accepted an order from a customer, and want to ship the goods via UPS. You can call a program running on UPS's computer system and have it calculate the cost of the shipment while you wait.
- Later, you can track that same shipment by calling a tracking program on UPS's system. You can have up-to-the-minute information about where the package is.

These are not just dreams of the future. They are a reality today with Web services.

SOAP and XML

Although there's a few different ways of calling web services today, things are becoming more and more standardized. The industry is standardizing on a technology called SOAP.

SOAP = Simple Object Access Protocol

SOAP is an XML language that describes the parameters that you pass to the programs that you call. When calling a Web service, there are two SOAP documents -- an input document that you send to the program you're calling, and an output document that gets sent back to you.

The format of a SOAP message can be determined from another XML document called a WSDL (pronounced "wiz-dull") document.

WSDL = Web Services Description Language

A WSDL document will describe the different "programs you can call" (or "operations" you can perform), as well as the parameters that need to be passed to that operation.

Sample WSDL (bottom)

```
. . .

<portType name="CurrencyConvertorSoap">
   <operation name="ConversionRate">
      <documentation>
         Get conversion rate from one currency to another currency
      </documentation>
      <input message="tns:ConversionRateSoapIn"/>
      <output message="tns:ConversionRateSoapOut"/>
   </operation>
</portType>

<binding name="CurrencyConvertorSoap" type="tns:CurrencyConvertorSoap">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                 style="document"/>
   <operation name="ConversionRate">
      <soap:operation soapAction="http://www.webserviceX.NET/ConversionRate"
                      style="document"/>
   </operation>
</binding>

<service name="CurrencyConvertor">
  <port name="CurrencyConvertorSoap" binding="tns:CurrencyConvertorSoap">
     <soap:address location="http://www.webservicex.net/CurrencyConvertor.asmx"/>
  </port>
</service>
</definitions>
```

Note: I removed the namespace identifiers and encodings to simplify the document a little bit.

Read it from the bottom up!

Sample WSDL (top)

```
<definitions targetNamespace="http://www.webserviceX.NET/">
<types>
  <schema>
    <element name="ConversionRate">
      <complexType><sequence>
          <element minOccurs="1" maxOccurs="1" name="FromCurrency" type="Currency"/>
          <element minOccurs="1" maxOccurs="1" name="ToCurrency" type="Currency"/>
      </sequence></complexType>
    </element>
    <simpleType name="Currency">
      <restriction base="string">
        <enumeration value="EUR"/>
        <enumeration value="USD"/>
      </restriction>
    </simpleType>
    <element name="ConversionRateResponse">
      <complexType><sequence>
          <element minOccurs="1" maxOccurs="1" name="ConversionRateResult" type="double"/>
        </sequence></complexType>
    </element>
    <element name="double" type="double"/>
  </schema>
</types>

<message name="ConversionRateSoapIn">
   <part name="parameters" element="ConversionRate"/>
</message>
<message name="ConversionRateSoapOut">
   <part name="parameters" element="ConversionRateResponse"/>
</message>
. . .
```

In the actual WSDL, all of the currencies of the world are listed here. I removed them to simplify the slide.

Read it from the bottom up!

Sample SOAP Documents

Again, I've removed the namespace and encoding information to keep this example clear and simple. (In a real program, you'd need those to be included as well.)

Input Message

```
<?xml version="1.0"?>
<SOAP:Envelope>
   <SOAP:Body>
     <ConversionRate>
       <FromCurrency>USD</FromCurrency>
       <ToCurrency>EUR</ToCurrency>
     </ConversionRate>
   </SOAP:Body>
</SOAP:Envelope>
```

Output Message

```
<?xml version="1.0"?>
<SOAP:Envelope>
   <SOAP:Body>
     <ConversionRateResponse>
       <ConversionRateResult>0.7207</ConversionRateResult>
     </ConversionRateResponse>
   </SOAP:Body>
</SOAP:Envelope>
```


HTTPAPI

Now that you know the XML data that needs to be sent and received, you need a method of sending that data to the server, and getting it back.

Normally when we use the Web, we use a Web browser. The browser connects to a web server, issues our request, downloads the result and displays it on the screen.

When making a program-to-program call, however, a browser isn't the right tool. Instead, you need a tool that knows how to send and receive data from a Web server that can be integrated right into your RPG programs.

That's what HTTPAPI is for!

- HTTPAPI is a free (open source) tool to act like an HTTP client (the role usually played by the browser.)
- HTTPAPI was originally written by me (Scott Klement) to assist with a project that I had back in 2001.
- Since I thought it might be useful to others, I made it free and available to everyone.

http://www.scottklement.com/httpapi/

Web Service Consumer (1/4)

```
H DFTACTGRP(*NO) BNDDIR('LIBHTTP/HTTPAPI')

D EXCHRATE        PR                  ExtPgm('EXCHRATE')
D   Country1                    3A   const
D   Country2                    3A   const
D   Amount                     15P 5 const
D EXCHRATE        PI
D   Country1                    3A   const
D   Country2                    3A   const
D   Amount                     15P 5 const

 /copy libhttp/qrpglesrc,httpapi_h

D Incoming        PR
D   rate                        8F
D   depth                      10I 0 value
D   name                     1024A   varying const
D   path                    24576A   varying const
D   value                   32767A   varying const
D   attrs                       *   dim(32767)
D                                   const options(*varsize)

D SOAP            s           32767A   varying
D rc              s              10I 0
D rate            s               8F
D Result          s              12P 2
D msg             s              50A
D wait            s               1A
```

A program that uses a Web Service is called a "Web Service Consumer".

The act of calling a Web service is referred to as "consuming a web service."

Web Service Consumer (2/4)

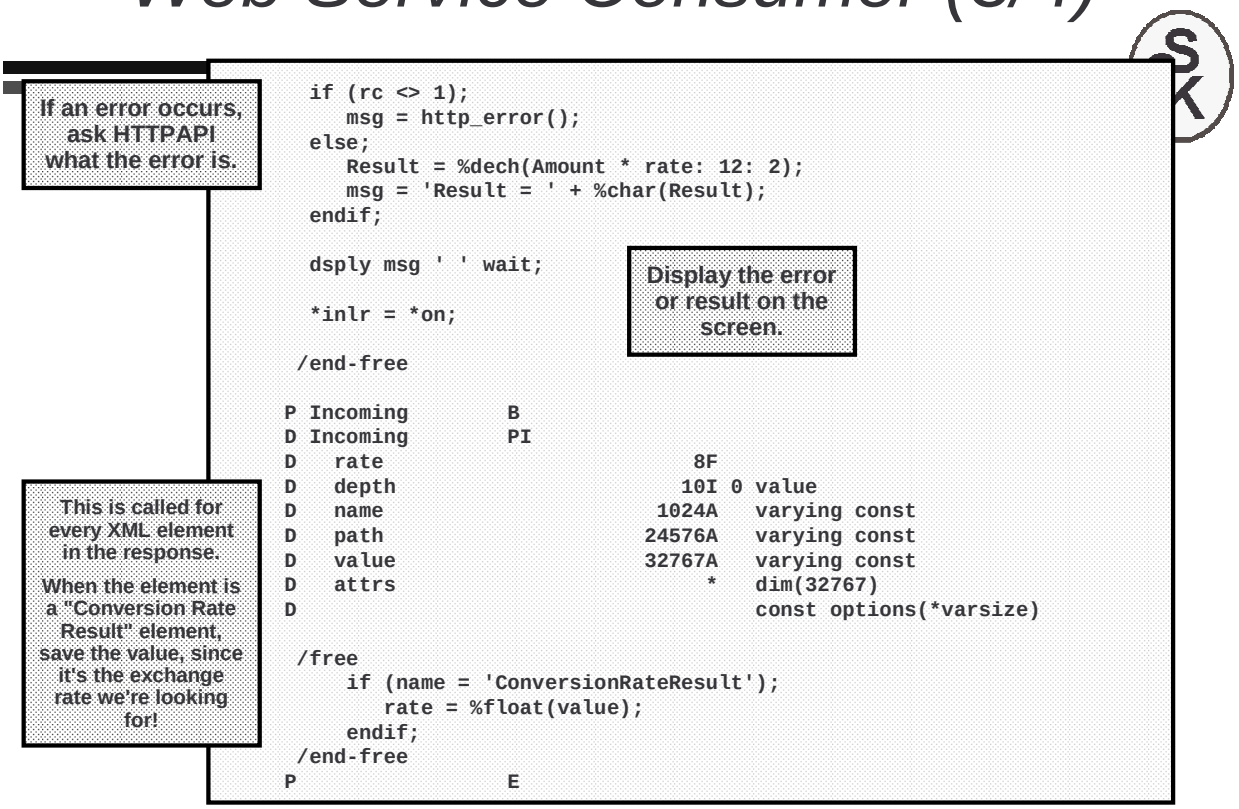
Constructing the SOAP message is done with a big EVAL statement.

```
/free
  SOAP =
   '<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>'
  +'<SOAP:Envelope'
  +'    xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"'
  +'    xmlns:tns="http://www.webserviceX.NET/">'
  +'<SOAP:Body>'
  +'  <tns:ConversionRate>'
  +'      <tns:FromCurrency>'+ %trim(Country1) +'</tns:FromCurrency>'
  +'      <tns:ToCurrency>'+ %trim(Country2) + '</tns:ToCurrency>'
  +'  </tns:ConversionRate>'
  +'</SOAP:Body>'
  +'</SOAP:Envelope>';

  rc = http_url_post_xml(
          'http://www.webservicex.net/CurrencyConvertor.asmx'
                : %addr(SOAP) + 2
                : %len(SOAP)
                : *NULL
                : %paddr(Incoming)
                : %addr(rate)
                : HTTP_TIMEOUT
                : HTTP_USERAGENT
                : 'text/xml'
                : 'http://www.webserviceX.NET/ConversionRate');
```

This routine tells HTTPAPI to send the SOAP message to a Web server, and to parse the XML response.

As HTTPAPI receives the XML document, it'll call the INCOMING subpocedure for every XML element, passing the "rate" variable as a parameter.

Web Service Consumer (3/4)



If an error occurs, ask HTTPAPI what the error is.

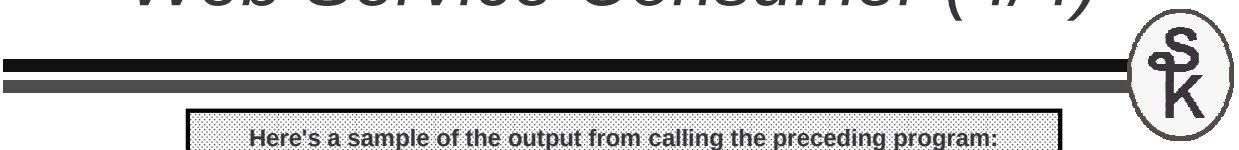
```
 if (rc <> 1);
    msg = http_error();
 else;
    Result = %dech(Amount * rate: 12: 2);
    msg = 'Result = ' + %char(Result);
 endif;

 dsply msg ' ' wait;

 *inlr = *on;

/end-free

P Incoming        B
D Incoming        PI
D   rate                          8F
D   depth                        10I 0 value
D   name                       1024A   varying const
D   path                      24576A   varying const
D   value                     32767A   varying const
D   attrs                          *   dim(32767)
D                                      const options(*varsize)

 /free
     if (name = 'ConversionRateResult');
        rate = %float(value);
     endif;
 /end-free
P                 E
```

Display the error or result on the screen.

This is called for every XML element in the response.

When the element is a "Conversion Rate Result" element, save the value, since it's the exchange rate we're looking for!

Web Service Consumer (4/4)



Here's a sample of the output from calling the preceding program:

```
                              Command Entry
                                                        Request level:   1
 Previous commands and messages:
    > call exchrate parm('USD' 'EUR' 185.50)
      DSPLY  Result = 133.69









Bottom
 Type command, press Enter.
 ===>

 F3=Exit    F4=Prompt    F9=Retrieve    F10=Include detailed messages
 F11=Display full      F12=Cancel     F13=Information Assistant  F24=More keys
```

More Information – CGIDEV2

CGIDEV2 is supported by IBM. The home page for CGIDEV2 is
http://www-03.ibm.com/systems/services/labservices/library.html

Tutorials on Web programming with CGIDEV2 are available at:
http://www.easy400.net

Scott has written several articles about CGIDEV2 for his newsletter:

- CGIDEV2 for XML
  http://www.systeminetwork.com/article.cfm?id=51276

- Web programming in RPG parts 1,2,3
  http://www.systeminetwork.com/article.cfm?id=51135
  http://www.systeminetwork.com/article.cfm?id=51145
  http://www.systeminetwork.com/article.cfm?id=51209

- CGIDEV2 for E-mail
  http://www.systeminetwork.com/article.cfm?id=51238

For More Information

You can download *HTTPAPI* from Scott's Web site:
`http://www.scottklement.com/httpapi/`

Most of the documentation for *HTTPAPI* is in the source code itself.
- Read the comments in the `HTTPAPI_H` member
- Sample programs called `EXAMPLE1` - `EXAMPLE20`

The best place to get help for *HTTPAPI* is in the mailing list. There's a link to sign up for this list on Scott's site.

Info about Web Services:
- *Web Services: The Next Big Thing* by Scott N. Gerard
  `http://www.systeminetwork.com/Article.cfm?ID=11607`
- *Will Web Services Serve You?* by Aaron Bartell
  `http://www.systeminetwork.com/Article.cfm?ID=19651`
- *W3 Consortium*
  `http://www.w3.org` and `http://www.w3schools.com`

For More Information

Web Service info, continued…

- ***RPG as a Web Service Consumer*** **by Scott Klement**
  `http://www.systeminetwork.com/article.cfm?id=52099`
- **SOAP Message Generator (automatically converts WSDL to SOAP):**
  http://www.soapclient.com/soapmsg.html
- **WebServiceX.net (Many, many useful web services)**
  http://www.WebServiceX.net
- **XMethods.net (More useful web services)**
  `http://www.xmethods.net`
- **UPS OnLine Tools**
  `http://www.ups.com/content/us/en/bussol/offering/technology/automated_shipping/online_tools.html`

This Presentation

You can download a PDF copy of this presentation from:

http://www.scottklement.com/presentations/

Thank you!