Web 2.0 Applications with EGL Rich UI

March 18, 2009

Note

The information contained in these materials is provided for informational purposes only and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to actual or potential IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Current planning about actual or potential product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

Before using this information and the product it supports, read the additional information in "Notices" at the end of this document.

Contents

Web 2.0 Applications with EGL Rich UI

This whitepaper briefly introduces *EGL*, a business language that both protects a company's investment in software and facilitates a developer's use of new technologies. Our main focus is on the language's support for *Web 2.0*, which is a set of innovative techniques for reaching users who rely on browsers, personal digital assistants, and cell phones.

We describe Web 2.0 and show how it relates to enterprise development. We then demonstrate how to use EGL to create a responsive user interface. Last, we tell how to access Web services, which may be deployed in techical environments as diverse as the Windows platforms, Linux, IBM i, and CICS® for z/OS®.

Introduction to EGL and EGL Rich UI

EGL is a high-level language that lets developers create business software without requiring that they have a detailed knowledge of runtime technologies or that they be familiar with object-oriented programming. The language is architected to reflect patterns that are common to different kinds of business software, and the language hides many details that are platform specific.

EGL helps a company retain developers who are knowledgeable in business processes, even if those developers lack the time needed to stay current with technical change. And the relative simplicity of the language helps traditional developers become accustomed to the latest technologies.

The language feature that supports Web 2.0 is called *Rich UI*. Working with this feature, you can create innovative user interfaces, which in turn can access back-end services that process enterprise data. The services may be developed by your company—written in EGL or another language—or may be available on the Web.

Of particular interest is this: you can use EGL logic—often, just a few lines of code—to exchange data between a Web 2.0 application and a long-standing

in-house application; for example, a COBOL application that runs on IBM i or System z. In this way, EGL help you to retain your software inventory as technologies change.

EGL also protects your company against the effects of future change. The language's support of multiple platforms means that migration of EGL code will be relatively easy.

In short, use of EGL is prudent. The language conserves developer time, training cost, and software inventory and is a hedge against future change.

EGL is provided in each of the following products:

- IBM Rational Business Developer
- IBM Rational Developer for i for SOA Construction
- IBM Rational Developer for System z with EGL

A free trial of Rational Business Developer is available at the following site:

http://www.ibm.com/rational/cafe/community/egl

Web 2.0

The next sections outline the general benefits of developing Web 2.0 applications.

Quick Application Development from Multiple Sources

Web 2.0 lets your company quickly develop *mashups*, each of which is a combination of software capability from different sources. For example, you might be developing an application to let travelers reserve rooms in one or another city. With Web 2.0, the application can provide access to a Google™ map for each hotel and can include, within the map, a weather forecast for the city. Neither the mapping software nor the forecast software was created with the other in mind, yet the two kinds of software are usefully integrated in a creative way.

Web 2.0 makes a huge amount of Internet-based data available to your company. The effect is to increase convenience for customers and internal users and to reduce development expense.

Greater Application Flexibility and Attractiveness

Web 2.0 interfaces are flexible and have a dynamic quality. The potential variations in style and content allow for a visual quality that is far removed from a traditional business application. As before, the developer's job is to create an interaction that gives users a set of choices appropriate to the business need. An additional goal is to let users be productive in their individual ways while protecting them from distractions and from being overwhelmed by unwanted choice.

Consider an interface that lets users find a real-estate property of interest (Figure 1). The user can search for a specific location, price range, type of property, and so on.

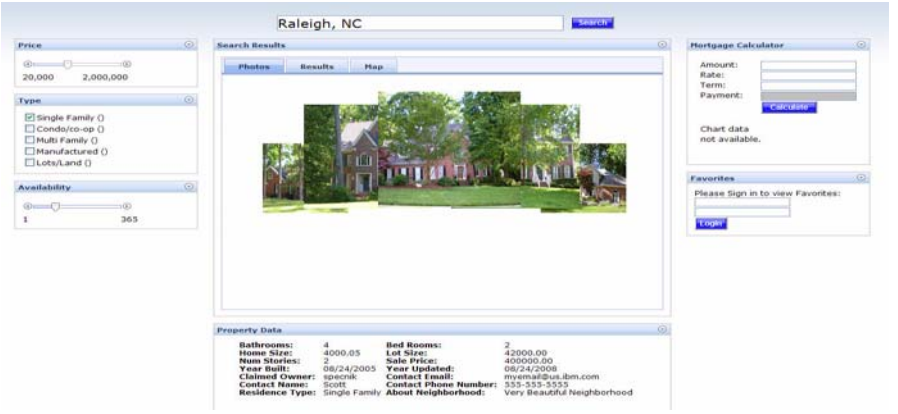

Figure 1: Example Interface for a Web 2.0 Application

The user changes entries at the left of the screen to display a set of pictures that reflect the search criteria. The user then uses a mouse to move a cursor over the set of pictures and in this way scrolls the images to the right or left. The bottom area of the screen shows details of whatever house is displayed at the front and center. The user can then access a mortgage calculator to quickly learn the monthly payment needed to pay back a loan of a given size and term for the selected property.

Later, we consider some of the technological advances that make Web 2.0 applications like this one possible; in particular, JavaScript™, AJAX, asynchronous service access, and cascading style sheets.

Increased Market Potential

Modern Web-based applications let you attend to the preferences of individual users. The value of that capability is highest for Web-based sales applications, for the following reason: attention to the changing preferences of a large number of users means that your company can benefit from niche markets, as noted in *The Long Tail: Why the Future of Business is Selling Less of More*, by Chris Andersen (Hyperion, 2006).

Here's an example. Amazon.com books that rank low in sales provide a relatively large percentage of overall sales compared to the benefit of such books in a brick-and-mortar store, where shelf space is a critical factor in attracting business. The increased sales of lower-ranked products is a consequence of two factors: the sheer numbers of potential customers on the Web, and the individual attention paid to those people. The consequence of increased interest in lower-ranked products is an increase in overall revenue.

Easier Application Distribution

Web 2.0 allows a quick response to changing business needs. The reason is easy application update. Developers can add capabilities without reinstalling an application on multiple workstations. A Web 2.0 application is deployed on a Web server such as WebSphere® Application Server or Apache Tomcat. Users then access the application from a Web browser that resides on a workstation or on a mobile device such as an iPhone. A user requests the most recent version of the application simply by clicking a button or typing a Web address.

Faster Collaboration in Pursuit of a Business Goal

Web 2.0 also allows a company to gain the benefits of faster collaboration. The blogs (Web logs) found throughout the Internet offer a model for handling business tasks such as consulting with experts or eliciting customer opinion. Search mechanisms are available for finding details in specific posts,

which can be categorized by keyword so that users can retrieve information as needed, avoiding the excessive information that undermines productivity.

Rich Internet Applications and EGL Rich UI

A subset of Web 2.0 applications is in the category of *Rich Internet Application* (*RIA*). An example is the real-estate application that we introduced earlier.

The processing of an RIA occurs primarily in the browser. The logic is said to be *client side* because the logic does not run on the remote machine that transmits the Web page. As a result, the response time for user interactions is unusually fast. Fewer resources are needed on the server, lessening the load on the server-side machine and reducing the need for hardware.

You write RIAs with EGL by using Rich UI.

With Rich UI, your logic is the basis of generated, client-side JavaScript. The important detail about JavaScript is that it provides greater flexibility so that the user's experience can go beyond receiving a page and submitting a form. Consider Figure 2, for example. After the user clicks a radio button, the code can respond by changing the content of a second control such as a textbox.

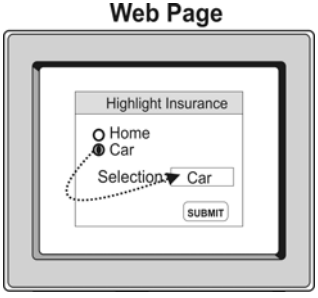

Figure 2: Example Use of JavaScript

The change occurs quickly because the logic runs locally and, in most cases, redraws only one area of the page.

An extension of client-side JavaScript is Asynchronous JavaScript and XML (AJAX), a technology that permits the runtime invocation of remote code and the subsequent update of a portion of a Web page, even as the user continues working elsewhere on the page. For example, after the user selects a purchase order from a list box, the JavaScript logic might request transmission of order-item details from the remote Web server and then place those details in a table displayed to the user. In this way, the application can access content from the server but can save time by selecting, at run time, which content is transmitted.

How a Rich UI Application Fits in the Enterprise

Figure 3 illustrates how a Rich UI application might fit into a larger technical scheme at run time.

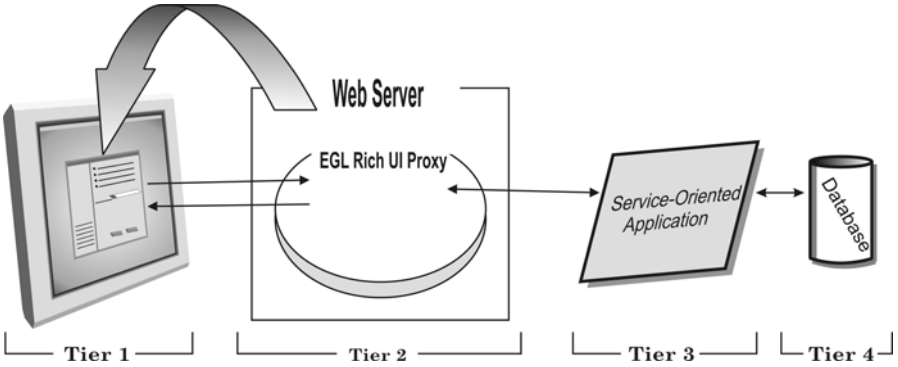

Figure 3: The Rich UI Application at Run Time

Our example scheme includes four logical *tiers*:

- Tier 1 is a Web browser where the Rich UI application runs
- Tier 2 is a Web server that transmits the Rich UI application and that handles the code's subsequent access of services (as described later)
- Tier 3 is the software that runs those services
- Tier 4 is the software that hosts a database

A tier is logical in the sense that some or all can be on the same machine. The distinction among the tiers gives you a way to think about the different kinds of processing that occurs at run time.

After a Web server transmits the Rich UI application to the user's browser, subsequent interaction with the server occurs only if the browser-based code accesses a *service*, which is a unit of logic that is more-or-less independent of any other unit of logic. A service might be half a world away from the user.

The Rich UI application can access any number of discrete services, and each might do a simple task such as provide details on a property. However, enterprise development often involves access of service-oriented *applications*. Each of these applications is composed of services that work as

a unit to fulfill a specific and complex purpose; for example, to process a mortgage request.

You automatically deploy the Rich UI application with the EGL Rich UI Proxy, which is EGL runtime code that handles the communication between the Rich UI application and the accessed services.

Support for Old and New Programming Models

A traditional COBOL, RPG, or Web application is form based. The typical steps may be quite familiar to you:

1. Transmit a form from a server, to elicit user input
2. On the client, validate the input for a given field; for example, to ensure that a numeric field contains only digits
3. Elicit the user's data submission, which occurs when the user takes an action such as clicking a SUBMIT button or pressing ENTER
4. On the server, validate the submitted data to ensure (for example) that one field value—such as U.S. state—is compatible with a second—such as zip code:
    - If the cross-field validation fails, redisplay the same form, including an error message and the user's input
    - If the cross-field validation succeeds, store the data in a database and, if necessary, transmit a different form to the user

Whatever the variations in these steps, the form-based processing model has widespread and continued value. When you use EGL Rich UI, you can organize data fields into a set of forms, can enforce a pre-specified ordering of user tasks, and can perform cross-field validation on a form-by-form or multiform basis. The new EGL technology lets you simulate use of the old.

However, even if you simulate use of the old technology, Rich UI offers benefits that are specific to Web 2.0. First, the cross-field validations can occur in logic that runs in the browser, so you can move from form to form quickly, without contacting the server. Second, any contact with the server is *asynchronous*, which means that the user does not wait for a response, but

keeps interacting with the application. Later, we show how a new variation in the EGL call statement makes this behavior possible.

Beyond leading the user through a sequence of forms, you can devise an application flow that updates different areas of the screen from within the application; for example, in response to the user's choice or to some combination of application data. Again, changes to the displayed Web page reflect statements that run in the client-side code.

End-to-end processing with EGL

Rich UI is only one of the EGL facilities that are available when you work with one of the products listed on page 2. You can always do the following tasks:

- Create an EGL Web project. Working in that project, you write, generate, and debug services and specify details on how a given service will be deployed.

- Create an EGL Rich UI project. Working in that project, you write a Rich UI application that interacts with users and that accesses services. The EGL debugger lets you step through the application, step into an invoked service that you generated previously, and step through the rest of the application. The Rich UI deployment wizard lets you organize the generated output for placement on one or another type of Web server.

- Organize your generated outputs for installation on the same Web server or on different ones.

The available tools include a visual editor for Rich UI, as well as a source-code editor and enterprise tooling that help you to handle many aspects of development and code preparation.

We use the phrase *end-to-end processing* to indicate that both the front- and back-end logic can be written in EGL. In response to a simple requirement, you can code all aspects of end-to-end processing. More important for enterprise development, you can share expertise with colleagues who are handling different aspects of a complex requirement and who share both the development environment and the coding language.

The ability to communicate easily—a result of shared knowledge and experience—offers a productivity boost beyond the boost offered by the tools and by the structure of EGL itself.

Deployment

A Rich UI application that is deployed to WebSphere Application Server or Apache Tomcat is accompanied by the *EGL Rich UI Proxy*, which is runtime software that oversees communication between the application and each service that is accessed by the application.

A Rich UI application that is deployed to a simple HTTP server—for example, to the Apache HTTP server—has the interface capabilities of a Rich Internet Application but does not support service access.

The DOM Tree

We now describe how browsers handle a Rich Internet Application at run time. The purpose is twofold:

- To help you learn the Rich UI technology faster, as is possible by clarifying the runtime effect of what you code at development time
- To make it easy for you to respond to advanced technical requirements

When a user enters a Web address into a browser, the browser transmits a request to a Web server, which is usually on a second machine. The address identifies a specific server and indicates what content is to be returned to the browser. For example, if you enter the address *http://www.ibm.com*, a server replies with a message that the browser uses to display the IBM home page. The question that is of interest now is, how does the browser use the message?

The browser brings portions of the message into an internal set of data areas. The browser then uses the values in those data areas to display on-screen controls, which are commonly called *widgets*. Example widgets are buttons and text fields.

Consider the next Web page (Figure 4).

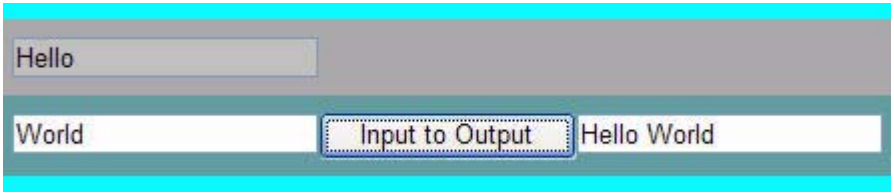

Figure 4: Example Web Page

Seven widgets are displayed:

- The enclosing box is **myBox**
- The upper box within myBox is **myBox02** and includes the text field **myHelloField**
- The lower box within **myBox** is **myBox03** and includes the text field **myInTextField**, the button **myButton**, and the textField **myOutTextField**

The internal data areas used by the browser are represented by an inverted tree (Figure 5).

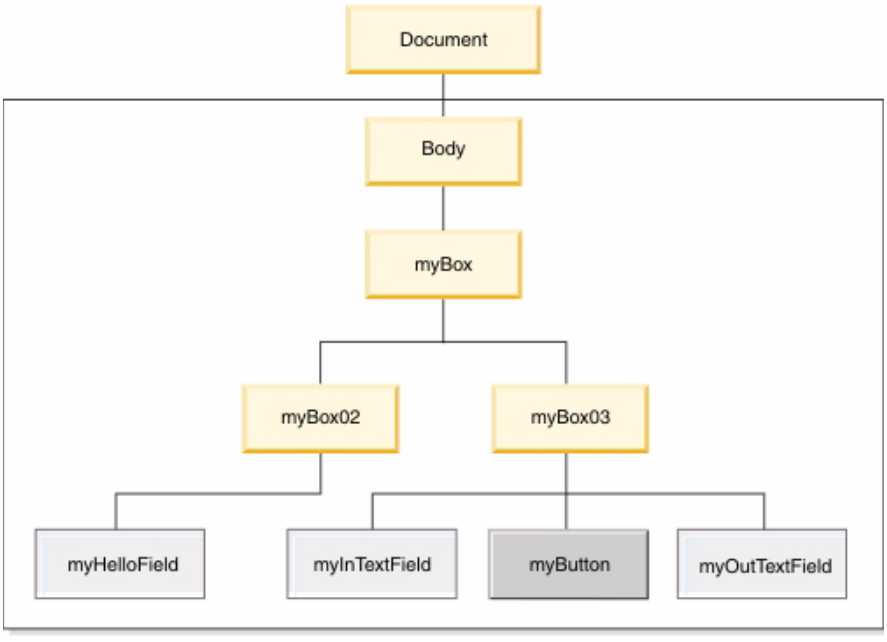

Figure 5: Example DOM Tree

The tree is composed of a root—named *document*—and a set of *elements*, which are units of information. The topmost element that is available to you is named *body*. The elements subordinate to body are specific to your application.

A set of rules describe both the tree and how to access the data that the tree represents. That set of rules is called the *Document Object Model* (*DOM*). We refer to the tree as the *DOM tree*; and we refer to the relationships among the DOM elements by using terms of family relationships:

- **myBox03** and **myInTextField** are *parent* and *child*
- **myBox** and **myButton** are *ancestor* and *descendant*
- **myInTextField**, **myButton**, and **myOutTextField** are *siblings*

In the simplest case (as in our example), a widget reflects the information in a single DOM element. In other cases, a widget reflects the information in a subtree of several elements. But in all cases, the spacial relationship among the displayed widgets reflects the DOM-tree organization, at least to some extent. The following rules describe the default behavior:

- A widget that reflects a child element is displayed within the widget that reflects a parent element.
- A widget that reflects a sibling element is displayed below or to the right of a widget that reflects the immediately previous sibling element

We often use a technical shorthand that communicates the main idea without distinguishing between the displayed widgets and the DOM elements. Instead of the previous list, we might say, “A widget is contained within its parent, and a sibling is displayed below or to the right of an earlier sibling.”

The DOM tree organization does not completely describe how the widgets are arranged. A parent element may include detail that causes the child widgets to be arranged in one of two ways: one sibling below the next or one sibling to the right of the next. The display also may be affected by the specifics of a given browser; for example, by the browser-window size, which the user can update at run time in most cases. Last, the display may be affected by settings in one or more *cascading style sheets*, which are files that set display characteristics for an individual widget or for all widgets of a given type.

When you develop a Web page with Rich UI, you declare widgets much as you declare integers. However, the widgets are displayable only if your code also adds those widgets to the DOM tree. Your code can also update the tree—adding, changing, and removing widgets—in response to runtime events such as a user's clicking a button. The central point is this: *Your main task in Web-page development is to create and update a DOM tree*.

Some of the tasks needed for initial DOM-tree creation are handled for you automatically when you work with the Rich UI editor to fulfill a drag-and-drop operation. Alternatively, you can create a DOM tree by writing EGL definitions in source code and can even reference DOM elements explicitly.

In general terms, you create and update a DOM tree in three steps:

1. Declare widgets of specific types—Button for buttons, TextField for text fields, and so forth—and customize the widget properties. For example, you might set the text of a button to “Input to Output,” as in our example. Here is EGL code to set the text:

```
myButton Button {text = "Input to Output"};
```

2. Add widgets to the initial DOM tree. Here is EGL code to relate one widget to the next:

```
handler MyHandlerPart
   type RUIhandler { initialUI =[myBox] }

   myBox Box{ children = [myBox02, myBox03] };

   myBox02 Box{ children = [myHelloField] };

   myBox03 Box{ children = [myInTextField,
                myButton, myOutTextField] };

   myHelloField TextField { text = "Hello" };

   myInTextField TextField{};

   myButton Button { text = "Input to Output" };

   myOutTextField TextField{};
end
```

3. Alter the DOM tree by adding, changing, and removing widgets at those points in your code when you want the changes to be displayable. Here is EGL code that resets **myBox03** so that the only child of that box is **myOutTextField**:

```
myBox03.children = [myOutTextField];
```

Given the Web page in Figure 4 as a starting point, the revised Web page is as follows:

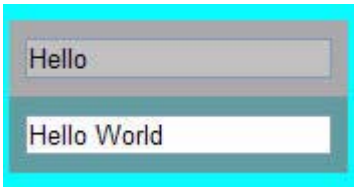

Figure 6: Revised Web Page Example

We say that a widget or its changes are “displayable” rather than “displayed” because a widget in a DOM tree can be hidden from view.

The Rich UI Handler Part

To create a Rich UI application, you code a Rich UI Handler part. The Handler part holds the EGL logic to add widgets to an initial DOM tree and to respond to events such as a user's click of a button.

Let’s consider our example Web page again (Figure 7).

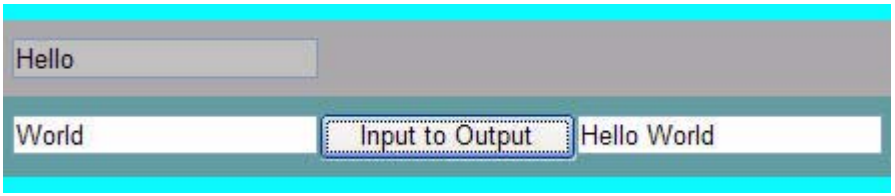

Figure 7: Example Web Page

The content shown there is displayed after the user types *World* in the field **myInputTextField** and clicks the button.

Here is the Rich UI handler part that provides the logic for displaying that Web page.

```
handler MyHandlerPart
   type RUIhandler {initialUI =[myBox]}

   myBox Box{padding = 8,
             children =[myBox02, myBox03],
             columns = 1,
             backgroundColor = "Aqua"};

   myBox02 Box{padding = 8, columns = 2,
               children =[myHelloField],
               backgroundColor = "DarkGray"};

   myBox03 Box{padding = 8, columns = 3,
      children =[myInTextField, myButton,
                 myOutTextField],
      backgroundColor = "CadetBlue"};

   myHelloField TextField
      {readOnly = true, text = "Hello"};

   myInTextField TextField{};

   myButton Button
      {text = "Input to Output", onClick ::= click};

   myOutTextField TextField{};

   function click(e EVENT in)
      myOutTextField.text =
         myHelloField.text + " " + myInTextField.text;
   end
end
```

Listing 1: Code for the Example Web Page

Consider the following components of the part **MyHandlerPart**:

- The definition statement at the beginning of the part has the identifier **RUIHandler**, which indicates that the handler logic will be transformed into client-side JavaScript. The part also has a set of handler properties with which you customize runtime behavior.

- The part includes a set of widget declarations, Those declarations—starting with the one for **myHelloField**—are ordered so that the later declarations such as for **myBox02** can access the earlier ones.

- The part includes an *on-construction function*, which is a function that runs when the handler is first invoked by the browser. In this example, the on-construction function is named **initialization** and does nothing. Later, we give example logic for the function.

- The part includes an *event handler*, which is a function that you write to respond to an event, In this example, the function **click** runs as soon as the user clicks the button **myButton**.

  You can also configure your code so that an event handler responds to an event that is internal to the code. Such an event might be receipt of a message that was returned from a service.

You can see how to create an initial DOM tree by reviewing the part named **MyHandlerPart**:

- To specify which widgets are the initial children of the body element, use the **initialUI** property of the Rich UI handler. In our example, the only child of the body element is **myTopBox**, which is a widget of type **Box**. This widget is a rectangular control that can contain other widgets.

- To specify which other widgets are descendents of the body element, use the **children** property of a container widget. In our example, **myTopBox** is the parent of **myBox02** and **myBox03**. The code also shows that **myBox02** is the parent of **myHelloField** and that **myBox03** is the parent of **myInTextField**, **myButton**, and **myOutTextField**. We illustrated these relationships earlier, in Figure 5.

Here's a revised example of the on-construction function, followed by a revised display (Figure 8):

```
function initialization()
   myButton.text = "Click here!";
   initialUI = [myBox03];
end
```

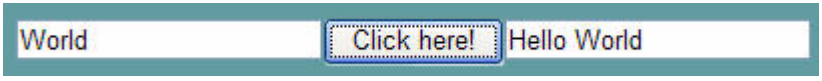

Figure 8: Revised Web Page

Introduction to the Rich UI Editor

You can use the EGL Rich UI editor to modify a Rich UI handler and to preview the handler's runtime behavior.

The Rich UI editor includes three views:

- The *Design surface* is a rectangular area that shows the displayable content of the Rich UI handler. You can drag-and-drop widgets from the palette into the Design surface and then customize those widgets in the Properties view.

- The *Source view* provides an embedded version of the EGL editor, where you update logic and add or update widgets. The Design view and Source view are integrated: changes to the Design view are reflected in the Source view; and, if possible, changes to the Source view are reflected in the Design view.

- The *Preview view* is a browser, internal to the Workbench, where you can run your logic. You can easily switch to an external browser if you prefer.

Figure 9 gives an example of a file open in the Rich UI editor and shows the Design surface, along with the palette at the right and, at the bottom left, a Properties view.

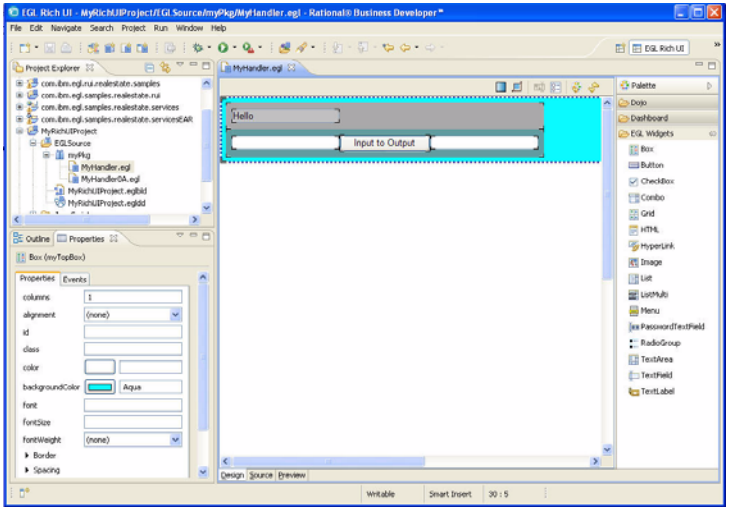

Figure 9: EGL Rich UI Editor

When you drag a widget from the palette to the Design surface, the areas that can receive the widget are called *potential drop locations*, and the color of those areas is yellow by default. When you hover over a potential drop location, the area is called a *selected drop location*, and the color of that area is green by default. You can customize those colors.

When you first drag a widget to the Design surface, the entire surface is a selected drop location, and the effect of the drop is to declare the widget and to identify it as the first element in the Rich UI handler's **initialUI** property.

When you drag another widget to the Design surface, you have the following choices:

- You can place the widget adjacent to the initially placed widget. The effect on your source code is to declare the second widget and to identify it as another element in the **initialUI** array. Your placement of the new widget is either before or after the first widget and indicates where the widget is placed in the array at development time. The ultimate effect is to specify another child of the DOM tree body element at run time.

- If the initially placed widget was a container—for example, a box—you can place the second widget inside the first. The effect on your source code is to add an element to the **children** property of the

container. The ultimate effect is to specify a child of the container element at run time.

Your subsequent work continues to build the DOM tree. You can repeatedly fulfill drag-and-drop operations, with the placement of a widget determining what array is affected and where the widget is placed in the array. The drag-and-drop operation is an alternative to writing a widget declaration and array assignment in the code itself.

Embedded Handlers

You can use multiple Rich UI Handler parts to compose a single application. However, by saying "embedded handlers" we do not mean to say that you physically embed one Handler part in another. Instead, one Handler part—a part that presents the user interface—declares a variable used to access the functions and widgets in a second Handler part. For example, the following statement declares a variable that provides access to the Handler part **secondHandlerPart**:

```
myVariable secondHandlerPart{};
```

A reasonable practice is to use embedded handlers for service invocation and for other back-end business processing.

If the embedded handler (**secondHandlerPart**) has an on-construction function, the function runs when the related variable (**myVariable**) is declared.

You use a dot syntax to access the widgets and functions in an embedded handler. In the following outline, the handler **secondHandlerPart** is assumed to have declared a button named **itsButton**, which is attached to the DOM tree only after the button is included in the embedding handler:

```
handler SimpleHandler type RUIHandler
   { initialUI = [ myVariable.itsButton ] }
   myVariable secondHandlerPart{};
end
```

You can access a function or property in an embedded widget by extending the dot syntax. For example, the following statement retrieves the displayed text of the embedded Button **itsButton**:

```
myString STRING = myVariable.itsButton.text;
```

The **initialUI** array of the embedded handler has no effect at run time.

Overview of Service Access

We turn now to the back-end processing, returning later to a review of the application structure.

Styles of Service Access

A service accessed from a Rich UI application can represent either of two styles:

- Traditional Web services fulfill a Remote Procedure Call (RPC) style. In this case, you pass a business-specific operation name (for example, GetEmployeeData) and a set of arguments. With this style, invoking a service is similar to invoking a function.
- In contrast, REST services conform to the rules of Representational State Transfer (REST). The *RESTful* style is based on the transfer of (at most) a single unit of business data; for example, a set of values that include an employee number, title, and salary. The invoked service fulfills one of four preset operations:
    - GET, for reading data; for example, to retrieve details on the specified employee
    - POST, for creating data
    - PUT, for updating data
    - DELETE, for deleting data

When you access a service that fully conforms with the principles of REST, you do not need to include a business-specific operation name such as GetEmployeeData.

A service provider indicates whether a service is deployed as a traditional Web service or as a REST service. An important difference between the two kinds of service technologies is their relative complexity:

- A traditional Web service receives and returns data that is in a format called SOAP. which once stood for Simple Object Access Protocol. The long name was dropped.

  The complexity of a traditional Web service is related not only to the message format, but to the required presence of a configuration file at deployment time. That configuration file is called a Web Services Description Language (WSDL) file.

  The overall complexity has a benefit for enterprise development, allowing for advanced capabilities related (for example) to security and to coordination among services. In any case, the task of working with the WSDL file is handled largely by automated tools.

- A traditional REST service exchanges messages but does not require the added complexity of SOAP. Access to this kind of service usually does not involve a configuration file.

Service access in Rich UI

Service invocation in Rich UI is always *asynchronous*, which means that the requester—the Rich UI application—continues running without waiting for a response from the service. The user can still interact with the user interface while the Rich UI application is waiting for the service to respond. However, you may want to indicate that an activity is occurring “behind the scenes” and must complete before the user continues working. For example, you can disable some functionality and present a simple animation until the service responds.

A Rich UI application can access a SOAP (Web) service or a REST (Web) service, and either kind of service may be written in EGL.

Access of a REST service involves a different way of thinking about service construction and interaction, as is explained in *Overview of Service Access* in the EGL Rich UI User Manual. (The Manual is cited at the end of this document.) However, if you access a REST service written in EGL, you can ignore the issue in favor of an RPC style that is similar to a function invocation.

To access a service, you write one or two functions to handle the response that follows a given invocation:

- A *callback function* receives the business data returned from a service
- An *onException function* (if present) receives error details if the runtime technology is unable to return business data

The overall process for invoking a service from Rich UI has four steps:

1. Create an EGL Interface part, in some cases by working through an automated task.
2. Create a variable based on the Interface part.
3. Code a **call** statement that includes a reference to a callback function and (optionally) a reference to an onException function.
4. Code the functions that were referenced in the **call** statement. You can start the task with a few clicks that quickly create empty functions for subsequent customization.

Division of Labor in EGL Rich UI

Modern data-processing systems organize logic by a division of labor known as *Model, View, and Controller (MVC)*. The terms—especially *view* and *model*—are variously defined:

- The *view* is the user interface, or the logic that supports the user interface, or the business data in the user interface
- The *model* is a database (or other data storage), or the logic that accesses a database, or the data that is sent to or retrieved from a database

- The *controller* is the logic that oversees data transfer between the user interface and the database-access logic

In many cases, the acronym MVC refers to processing that involves multiple platforms. For example, a Rich UI application on a Windows platform might be considered to be the view (and to include the controller), while a service that accesses a database might be considered to be the model.

You can distinguish the view from model in the Rich UI application itself. In this case, the terms have a constant meaning:

- The view is a widget in the user interface
- The model is a data field that is internal to the application

EGL Controller and ValidatingForm

Rich UI provides two definitions to help you create an MVC division of labor. The **controller** lets you tie a specific view to a specific model. The **validating form** lets you reference a set of controllers and in this way simulate traditional form processing.

Figure 10 shows a set of widgets, including a validating form that allows input of a user name, password, and email address.

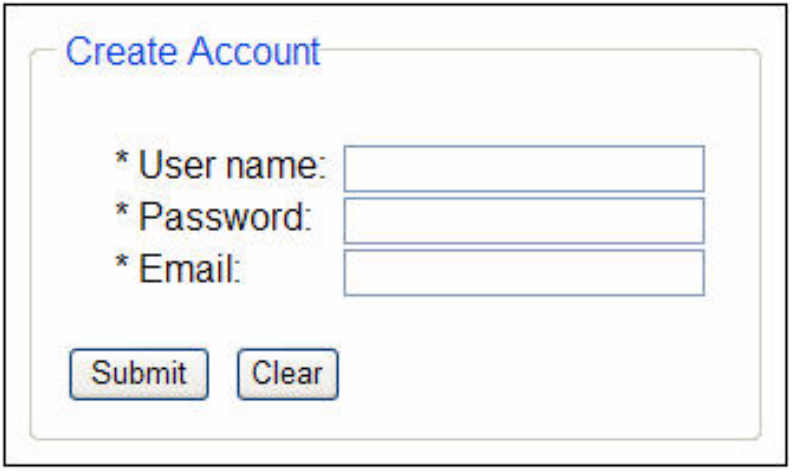

Figure 10: Traditional Form in Rich UI

The following outline indicates how you can accomplish the display:

```
handler MyHandlerPart
   type RUIhandler {initialUI =[myDiv]}

   myDiv Div {
      width = 310,
      borderStyle="solid",
      borderWidth=1,
      padding=10,
      children=[ myGrouping ]};

   myGrouping Grouping {
      text="Create Account",
      legend.font="Arial",
      children=[
         myForm,
         new Box {
            marginTop=3,
            children=[ submitButton, clearButton ]
         }
      ]};

   myForm ValidatingForm{};

   submitButton Button{ text = "Submit",
      onClick ::= submitForm };

   clearButton Button{ text = "Clear",
      onClick ::= clearForm };

   myService MyInterfacePart{@BindService{}};

   myRecord MyRecordPart{};

   // other declarations are not displayed
end
```

The topmost widget is a *div*, which is a container that offers flexibility in Web-page design. Here, the div includes a single child—a *grouping*, which is a widget collection preceded by customized text. The text in this case is *Create Account*. The grouping in turn includes a validating form—our main concern—and a box that contains two buttons.

Also present in the preceding code are two declarations:

- **myService** is a variable that is based on the Interface part **MyInterfacePart**. We'll use the variable to access a Web (SOAP) service.

- **myRecord** is a record that points to an area described by the Record part **MyRecordPart**. We'll use the record to work with **myForm**, the validating form.

Consider the full declaration of **myForm**:

```
myForm ValidatingForm {
   marginTop=20, marginLeft=20, marginBottom = 20,
   entries = [
      new FormField
         { displayName="* User name:",
           controller = usernameController },
      new FormField
         { displayName="* Password:",
           controller = passwordController },
      new FormField
         { displayName="* Email:",
           controller = emailController }
   ]
};
```

The first assignments in that declaration separate the form from other content. However, at the heart of the declaration is the **entries** field, which specifies an array of validating-form fields. Each field includes a label (for example, ** User name:*) and a controller. Here are the controllers:

```
usernameController Controller
   { @MVC { view = userNameField,
            model = myRecord.userName } };

passwordController Controller
   { @MVC { view = passwordField,
            model = myRecord.password } };

emailController Controller
   { @MVC { view = emailField,
            model = myRecord.email },
     validators ::= validateEmail };
```

Each controller includes the complex property **@MVC**, which ties a view to a model. Each view in this case is basically a text field, although the widget

passwordField additionally ensures that the user's input is displayed as a series of unreadable characters, for user security:

```
userNameField TextField{};
passwordField PasswordTextField{};
emailField TextField{};
```

Each model is a field in **myRecord**, the record that points to a data area described by the following Record part:

```
Record MyRecordPart
   userName string {
      MinimumInput=6,
      ValidationPropertiesLibrary = ValidationMessages,
      MinimumInputMsgKey = "usrMinInput" };
   password string {
      MinimumInput = 8,
      ValidationPropertiesLibrary = ValidationMessages,
      MinimumInputMsgKey = "pwMinInput" };
   email string {};
end
```

The declaration for a model such as **myRecord.userName** can include simple properties that cause the EGL runtime to validate user input. In our example, the property **MinimumInput** indicates that the minimum number of characters for the user name is 6 and for the password is 8. Two of the models also identify a Rich UI properties library and a message key. Those last details are part of a mechanism by which you assign displayable text in an external file, as needed to isolate changeable text outside of your code. One use of that mechanism is to communicate with users who read one language or another.

In addition to specifying validations as shown in the previous code, you can write *validators*, which are customized functions that also validate input. In

our example, the controller **emailController** references the validator **validateEmail**:

```
emailController Controller
   { @MVC { view = emailField,
            model = myRecord.email },
     validators ::= validateEmail };

function validateEmail(input String in) returns(String?)
   i int = StrLib.indexOf(input, "@");
   if (i > 0)
      i = StrLib.characterLen(input) - 3;
      if (i == StrLib.indexOf(input, ".", i));
         return("");
      end
   end
   return(ValidationMessages.emailError);
end
```

The example function ensures that the “at sign” (@) is in the input email address and that a period is three characters from the end of the address.

In general, you can code a validator of any complexity. A returned blank or null from the validator indicates success, and a returned string is displayed as an error message at the right of the validation-form field. We’ll explain **ValidationMessages.emailError** later, after we outline the process by which a Rich UI application has the following effect: validates user input; commits all the validated data to some backend code—in our case, to a service; and presents the returned data to the user.

Form Validation, Commit, and Publish

You handle form validation by coding an event handler that responds to the user's click of a button. In our example, the function is named **submitForm**:

```
function submitForm(event Event in)

   if (myForm.isValid())
      myForm.commit();

      call myService.register
         (myRecord.userName,
          myRecord.password,
          myRecord.email)
          returning to myCallback;
   end
end
```

When the function **myForm.isValid** runs, the validations for each controller are handled in turn, in form-field order. The function **myForm.commit** is invoked only if all validations succeed. Each controller-specific validation includes several steps; and then the function **myForm.commit** causes a second series of steps, invoking controller-specific **commit** functions to transfer data from views to models.

Two important details are not shown. First, Rich UI offers a way to minimize user inconvenience when authentication is necessary for accessing the Rich UI application, the EGL Rich UI Proxy, and individual services. For details, see the sections on security in the EGL Rich UI User Manual and, in particular, see *EGL single sign-on*.

Second, you can use a validation form to display the business data retrieved from a service. The function you use in this case is **publish**; for example, **myForm.publish**. Such a function invokes the controller-specific **publish** functions to transfer data from models to views.

Rich UI gives you much control over the runtime behavior described here, although system defaults are available. For details on controllers and form processing, see the following sections in the EGL Rich UI User Manual:

- *Rich UI validation and formatting*
- *Form processing with Rich UI*

Service Invocation and Response

Our example assumes that you are using an Interface part named **MyInterfacePart**:

```
Interface MyInterfacePart
   Function register(userName STRING in,
                     password STRING in,
                     email STRING in)
                     returns(STRING);
end
```

Our invocation of the service operation named **register** requires the presence of the callback function **myCallback**, which accepts the returned business data. That function refreshes the page with a set of widgets embedded in the original Grouping widget (Figure 10).

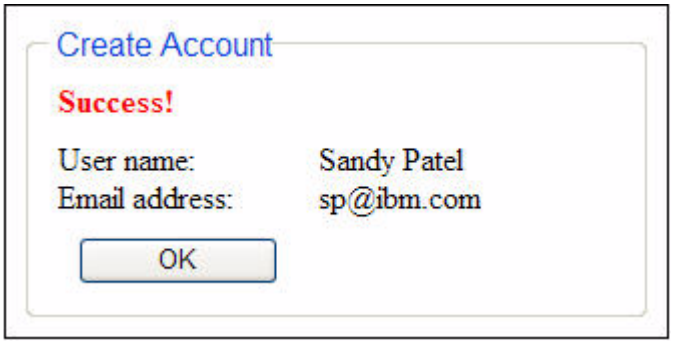

Figure 11: Display that Results from a Successful Service Access

Here is **myCallback**:

```
function myCallback(serviceResponse in)
   myGrouping.children = [
      new TextLabel {
         color="red", fontWeight="bold", margin=10,
         text = serviceResponse },
         new Box { columns=2, marginLeft=10,
            children=[
               new TextLabel { text="User name:" },
               new TextLabel { marginLeft = 10,
                               text=myRecord.username},
               new TextLabel { text="Email address:" },
               new TextLabel { marginLeft = 10,
                               text=myRecord.email},
               new Button { width = 100, margin = 10,
                            text = "OK",
                            onClick ::= reset}
            ]
      }
   ];
end
```

The user's button click invokes the function **reset**:

```
function reset( e Event in )

   myRecord = new MyRecordPart{};
   myForm.publish;

   myGrouping.children = [
      myForm,
      new Box { marginTop=3,
                children=[
                   submitButton,
                   clearButton
                ]
              }
   ];
end
```

The function begins by pointing the global record **myRecord** to a new data area of type MyRecordPart and then by publishing the empty data. The rest of the code re-creates the original display (Figure 9).

For additional details on service access, see the EGL Rich UI User Manual section *Accessing a service in Rich UI,* including the embedded sections on REST and SOAP service access.

Defining Displayable Text in an External File

Rich UI lets you define displayable text in an external file used at run time. The mechanism is useful for the following reasons:

- To override the runtime messages that are available, by default, for failed input validations or for incorrect formatting on output
- To assign text to widgets without hard-coding that text in the Rich UI application
- To display text in one or another language

The basic idea is that you set up variables in a library (type RUIPropertiesLibrary) and then provide the runtime content of those variables in a properties file that is referenced in the library. Here is an example library:

```
Library ValidationMessages type RUIPropertiesLibrary
   { propertiesFile="myFile" }
   emailError STRING;
   otherContent STRING;
end
```

The content in a properties file can include *inserts*—substitution variables—whose values are specified in your code. This feature lets you embed business data—for example, a specific employee number—in a message.

You can have several properties file for a given library, one file for each language you are supporting. The naming convention is crucial:

- You specify the root of the name as the value of **propertiesFile** (in this case, the root is **myFile**).
- When you create the properties files, you specify names that include locale information. For example, the file **myFile-en.properties** has content in English, while the file **myFile-es.properties** has content in Spanish

The invocation of the Rich UI application determines which locale to use. For example, to ensure use of Spanish files, the user might invoke an application as *www.example.com/MyApplication-es.html*. A set of rules determines which properties file to access if the invocation does not match an available locale.

For details on using properties file, see the EGL Rich UI User Manual, section *Use of properties files for displayable text*.

For Additional Information

The EGL Rich UI User Manual contains material that is also in the product help system; however, the content in the Manual is likely to be more recent:

http://www.ibm.com/rational/cafe/docs/DOC-2689

The EGL Cafe also includes material of interest:

http://www.ibm.com/software/rational/eglcafe

For a concise introduction to EGL, see *IBM Rational Business Developer with EGL* (MC Press, 2007):

http://www.mc-store.com/5087.html

For an expansion of the current essay, see *Enterprise Web 2.0 with EGL* (MC Press, expected May 2009).

For details on Web-page design, see the following Web sites:

http://www.w3schools.com/css

http://css.maxdesign.com.au

For a complete description of cascading style sheets, see *CSS: The Definitive Guide* (O'Reilly Media, Inc., November 2006).

For an overview of service interaction and some of the RPC-related technologies, see *SOA for the Business Developer* (MC Press, May 2007).

For a good introduction to REST services, see *Restful Web Services* (O'Reilly Media, Inc., May 2007).

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs

(including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
3600 Steeles Avenue East
Markham, ON Canada L3R 9Z7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- CICS
- CICS/ESA®
- ClearCase®
- DB2®
- IBM
- IMS
- Informix
- iSeries
- MQSeries®
- MVS™
- OS/400®

- RACF®
- Rational
- VisualAge
- WebSphere
- z/OS

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT® are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names, may be trademarks or service marks of others.