The Anatomy of a Web App Makeover

The BASIS Product Suite Download page, written years ago with a mixture of Perl, HTML, JavaScript, and SQL, was in great need of a makeover to address a growing list of enhancement requests from our community. As a testament to the complexity of this download page, its Perl code alone relied on a long list of external libraries to integrate critical functions such as CGI, database, email, FTP, SOAP, date/time manipulation, SSL and cryptography, and MIME encoding integration. With code that spread out over multiple files and various languages, this page became increasingly difficult to maintain and add the user-desired features. The time had come to rewrite it in a simpler, consolidated, and more easily maintainable language that was capable of doing everything the previous system could do and more to accommodate the upcoming improvements. Our tool of choice – BBj®, of course! With BBj's built-in BUI functionality, it was an easy decision.

New Features

The new BUI (browser user interface) download page delivers a nice list of new features available in its first release:

- Localization for five different languages
- Locale auto-detection with the option to select a language at any time for real-time translation (see **Figure 1**)
- Build timestamp display
- Reduced amount of required contact information
- Dynamic build retrieval from an Amazon S3 bucket

Figure 1. An example of the locale auto-detection

By Nick Decker
Engineering Supervisor

The Latency Test

Concerned about performance of the download page under high latency conditions, we launched a copy of our BBj production server's Amazon EC2 instance in Ireland. Surely, a distance of about 4,700 miles ought to be significant enough to introduce some network delays from the BBj interpreter in Ireland to our browsers in Albuquerque! The average ping time from Albuquerque to the West Coast production server was about 45ms, while the ping time to Ireland was more than five times that amount – an average of 245ms!

Latency can have a dramatic impact on any program where the client and server must pass information back and forth. In the case of the product download page, BBj was running in the BUI paradigm, but the same would hold true for a thin client deployment of the code. Even though the new download page has fewer fields to complete compared to the old page, the program running on the server still needs access to the information in all of those fields. In order to ensure that the user fills in the required fields appropriately (**Figure 2**), and to make use of the information later in the process to check export compliance, the BBj program needs to retrieve the contents of the 17 form elements from the client's browser.

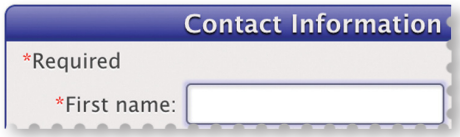

Figure 2. An example of one of the 17 required fields on the form

Making that many round trips to the client can be really time-consuming, especially as network latency increases. Our initial round of testing revealed what we had expected; validating the contents of controls on the user's browser was very time consuming.

If only there was some way to avoid asking the client 17 times for information regarding the status and contents of the controls on the form. Wouldn't it be nice if the interpreter could magically harvest all of the information at once?

Enter an Old Friend - Form Validation

It turns out that BBj has had a related paradigm already in place for a traditional thin client deployment – BBjFormValidation (see *Input Validation - Veto Power* at links.basis.com/05validation). The concept of form validation has been around for as long as the Internet itself. Because it becomes expensive to repeatedly ask the client for information regarding the contents of controls in response to user changes, web pages wait until the user completes entering all the changes before validating the fields. Once the user presses the [Submit] button, the program proceeds to validate all of the fields on the form. Beginning in version 12, BBj adds the same form validation that was previously available in the thin client to the BUI paradigm.

Adding form validation is the first of two improvements that we implemented to solve the download page's latency problem. While form validation allows the programmer to lock the form preventing the user from making any further changes until the validation has completed, it does not do anything to reduce the number of round trip communications between the server and the client. We addressed this second part of the problem with a dramatic improvement to the BBjFormValidation event itself; it now carries a payload of information with the state and contents of the controls on the form. This means that the programmer's validation routine no longer has to ask the client for the text in a BBjEditBox, or whether a BBjCheckBox was selected or not. Instead of asking the client, which causes network delays, the program simply retrieves the desired information from the form validation event itself on the server.

Retrieving information from the Form Validation Event was so useful and revolutionary, that BASIS made similar changes to many other events unrelated to form validation. For example, we augmented all of the BBjList events with information about the control's state. Previously, the BBj program registered for the BBjListSelectEvent so that it could react to any changes in the selection of a BBjList control. When the user selected an item in the list, this action notified the program and would then execute the callback routine or method associated with that event. In most cases, however, the program would immediately turn around and ask the control for the item in the list that the user selected. After all, if the program cared that the user selected an item in the list, chances are pretty good that it also cared which item in the list the user selected. Therefore, whenever the user selected an item in the list, the event would result in another round trip question/answer from the server to the client in order to find out the new selection in the list control.

To eliminate that round trip, BBj 12 augments the BBjList events with three new events:

1. getSelectedIndex() - returns the currently selected item in the list
2. getSelectedIndices() - returns a vector of all currently selected items in the list
3. getSelectedItem() - returns the text of the currently selected item in the list

These new events allow the BBj program to respond to a selection event in the list control and have all of the information it will need ahead of time from the event itself. BASIS also added these same improvements to other events, such as BBjLostFocusEvent and BBjEditModifyEvent. It is worth mentioning once more that these new methods are available in both SYSGUI and BUI, so they will improve the speed of many of your existing GUI applications. **Figure 3** shows form validation in action.

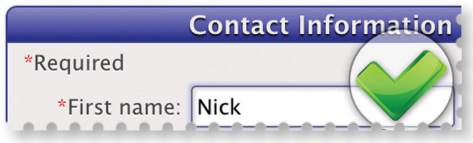

Figure 3. Form validation makes it easy to ensure that the user provides required information

Why BBj BUI Made Life Easier

Consolidating all of the underlying code for the download page into BBj greatly simplified things. Some of the more difficult aspects of the old download page, such as the export compliance check, were much easier to implement in BBj and required far less code. The old page built a SOAP message from the ground up, creating headers, adding data to the message body, and using specially coded routines to handle more complex data types. In contrast, the new BBj version of the download page utilizes a Web Service to accomplish the same task. The code is simple and straightforward; add a couple of jars to your classpath in Enterprise Manager (EM), instantiate the Web Service client, put some properties in a HashMap, and execute the `runTransaction()` method. The BBj code (**Figure 4**) is succinct and provides a high level of readability and maintainability.

```
method protected BBjNumber checkExportCompliance()
    declare java.util.HashMap request!
    declare java.util.HashMap reply!

    REM Create a new HashMap
    request! = new java.util.HashMap()

    rem Add all of the user information
    request!.put("billTo_firstName", #savedStateFirstName$)
    request!.put("billTo_lastName", #savedStateLastName$)
    ...

    rem Run the transaction
    reply! = Client.runTransaction(request!, props!)
```

Figure 4. Excerpt of the export compliance BBj code

Several other routines were also significantly easier in BBj compared to the previous version including validation, user notification LightBox, and cookie management. One of the best simplifications to the new program was to access BASIS data files and databases natively. The old version, which relied on Perl for the backend CGI language, used the DBI (database independent interface) library for database access. This required the DBD::JDBC module that works in conjunction with a server written in Java to provide a DBI front end to a JDBC driver, acting as a bridge between Perl and a BASIS JDBC database. The end result is that we were able to eliminate more libraries, remove the dependency on required run-time processes, improve compatibility, and speed up data access in the process.

Having all of the code in a single language and in a single program not only simplifies the process, but makes the coding/debugging/maintaining much easier as well. The old version retrieved values from the browser via JavaScript, validated

them; sent them back to the Perl CGI program on the server, which manipulated them in order to pass them on to the export compliance services in a SOAP message, then inserted them into a database. Not only do different languages like JavaScript, Perl, and SQL each have a vastly different syntax, but they also have different data types that can complicate the passing of variables from a program in one language to another program in a different language. Even comparing data in JavaScript and Perl is very different; JavaScript uses the `!=` operator to test for inequality, while Perl only uses `!=` when comparing numbers and instead uses the **ne** comparison operator when the value is a string. Having all of the code and data in a single language reduces the demands and requirements for the programmer and eliminates the need to keep track of all of the special rules for various languages as illustrated in **Figure 5**. The end result is a program that is much more robust, less prone to breakage, and far easier to debug.

Figure 5. Consolidating everything into a single language - BBj

Testing was Easier Too!

Because we developed the new download page inside the BASIS IDE, testing this page was also very easy. Simply tapping the [F5] key or clicking the [BB Execute] button instantly launched the program in thin client mode. Adding a BUI definition for the app in EM only took a minute, so testing it in BUI mode in a browser was also straightforward. EM provides a link to the BUI app so sending off a quick email to the BASIS QA department with the link greatly facilitated testing efforts. Because a BBj installation includes a fully-configured Web Server that is available when starting BBj Services, the QA department could also check out the BBj program and resource for the download page from our SVN source code repository and then host and test the new page locally.

By contrast, testing the old download page was extraordinarily difficult. We could only run the old page on a system pre-configured with Apache Web Server, Perl interpreter, dozens of external libraries and modules built from source code, runtime Perl/JDBC bridge server, and more. Because setting up and maintaining this server was so extensive and time consuming, it was just impractical and extremely expensive to have more than one development machine.

Comparing the Old and New Download Pages

One goal of BUI-izing the product download page was to reduce the size and complexity of the form. Comparing the old and the new pages side-by-side in **Figure 6** shows our success; the new

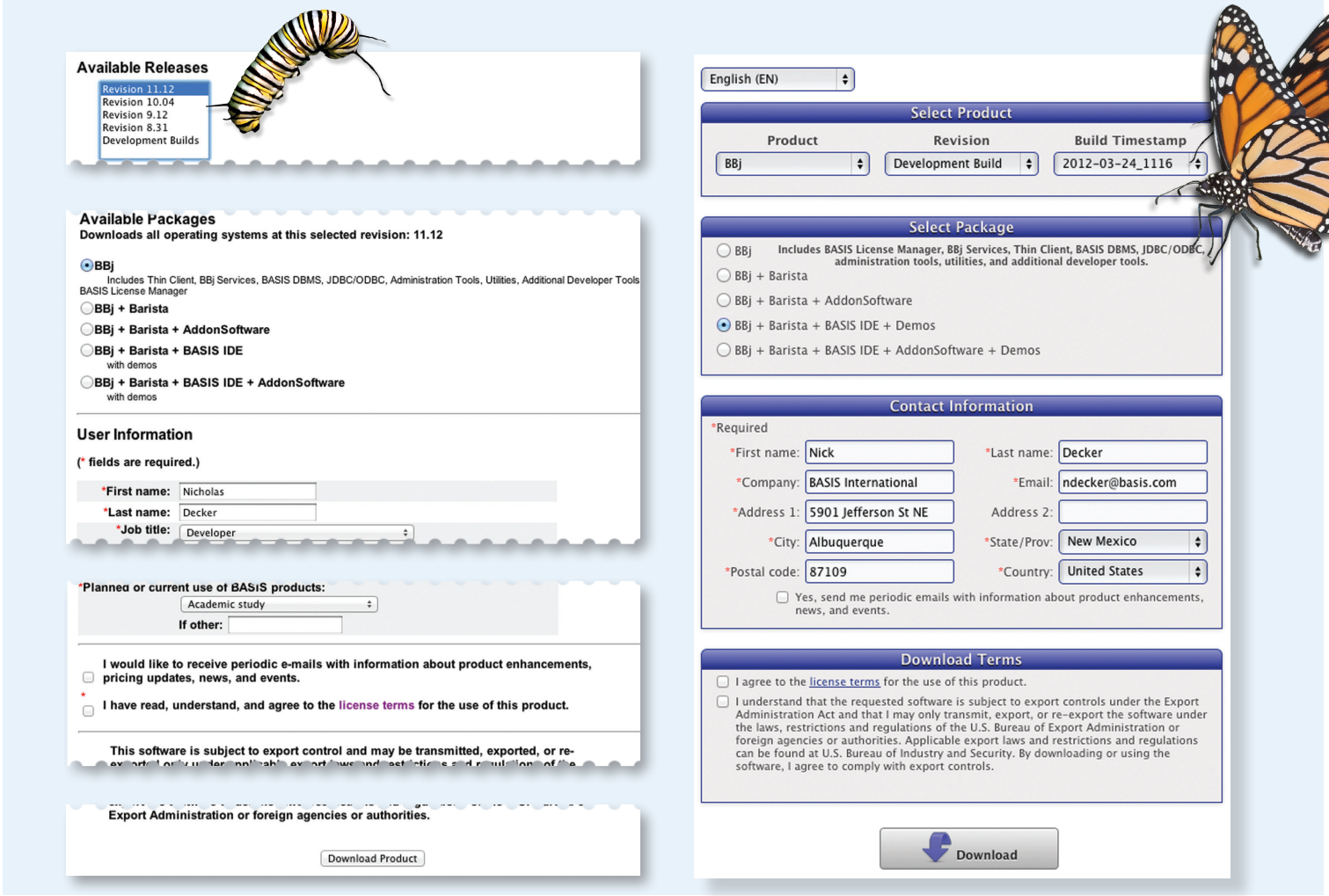

Figure 6. Comparing the old (left) and new (right) download page

page is smaller, more compact, and eliminates unnecessary data such as “Fax Number” and unwieldy supporting controls like the “US/Canada” and “Other” radio buttons and the various “If Other” text boxes. BUI’s integration of Cascading Style Sheets also assists with making the form more visually attractive while extending the BASIS website theme, including the same fonts, colors, and gradients, where appropriate.

New Features

One new feature of the BASIS Product Suite Download page is localization into five different languages. While the BUI program uses the BBTranslator Utility to translate the controls in real time, BASIS used the new BASIS Resource Bundle Editor (links.basis.com/rbe) to create and maintain all of the text in specialized resource bundles. When the program starts, it first checks to see if the URL specifies a locale for the BUI application. If there is no locale in the URL, the program checks for a past locale setting that it saved in a cookie. If that is not available either, then the program retrieves the locale from the client’s machine.

Once the program has a valid locale setting, it calls a method to translate all of the controls. If the locale is not one of the five supported languages, then it uses the default language, kept in sync with English by the BASIS Resource Bundle Editor. The user may also change the language at any time by selecting a pre-populated option from the BBjListButton. The callback routine for this list selection event retrieves the selected locale from the BBjListSelectEvent to avoid a trip to the client, then calls the method to translate the controls using the new locale. **Figure 7** shows a portion of the page translated into Italian.

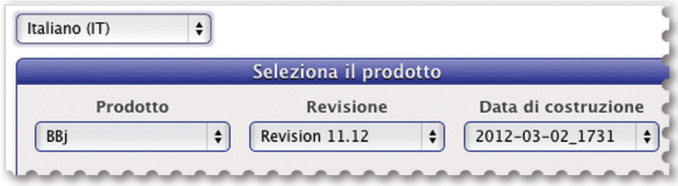

Figure 7. The download page translated into Italian

BUI and GUI Mortgage Demo Optimizations

To illustrate how you might modify your GUI or BUI application to take advantage of the new form-validation’s payload, consider how we applied the changes to the BUI Mortgage demo. The original version of the code calculated the payment information when the user pressed the [Calculate] button. The shift to form validation does not change much in this part of the code, just the event type for which the button is registered. Instead of reacting to the ON_BUTTON_PUSH event, we now trigger form validation from the button by registering for the ON_FORM_VALIDATION event as shown in **Figure 8**.

```
102  REM Register for events
103      myWindow!.setCallback(mySysGUI!.ON_CLOSE,"WIN_CLOSE")
104      myWindow!.setCallback(mySysGUI!.ON_SCREEN_RESIZE,"SCREEN_RESIZE")
105      myExitButton!.setCallback(myExitButton!.ON_BUTTON_PUSH,"WIN_CLOSE")
106      myCalcButton!.setCallback(myCalcButton!.ON_BUTTON_PUSH,"CALCULATE_PAYMENT_INFO")
107      myPrintButton!.setCallback(myPrintButton!.ON_BUTTON_PUSH,"PRINT_SCHEDULE")
```

```
REM Register for events                                                          101
    myWindow!.setCallback(mySysGUI!.ON_CLOSE,"WIN_CLOSE")                        102
    myWindow!.setCallback(mySysGUI!.ON_SCREEN_RESIZE,"SCREEN_RESIZE")            103
    myExitButton!.setCallback(myExitButton!.ON_BUTTON_PUSH,"WIN_CLOSE")          104
    myCalcButton!.setCallback(myCalcButton!.ON_FORM_VALIDATION,"CALCULATE_PAYMENT_INFO")  105
    myPrintButton!.setCallback(myPrintButton!.ON_BUTTON_PUSH,"PRINT_SCHEDULE")   106
```

Figure 8. Changing the calculation button’s callback to trigger a Form Validation Event

The next change took place in the subroutine that executes when the user presses the [Calculate] button. To start with, we retrieved the Form Validation Event into an object, highlighted in green in **Figure 9**. Because the event object contains all of the information about all of the controls on the form, the next few changes deal with getting the contents of the various controls. This is the most important part of the change since it is where we eliminate all of the latency overhead of making round trips asking the client for information. This change is pretty simple, too. Instead of getting the value of the **`myPrincipal!`** InputN control directly, we modified the code shown in **Figure 9** to get the value from the form validation event by referencing the **`Principal!`** InputN control.

```
                                                    REM Get the form validation event                    213
                                                    event! = mySysGUI!.getLastEvent()                    214
                                                                                                         215
REM Determine principal                             REM Determine principal                              216
pOriginal = myPrincipal!.getValue()                 pOriginal = event!.getValue(myPrincipal!)            217
pctDown = myPctDown!.getValue()/100                 pctDown = event!.getValue(myPctDown!)/100            218
pctDownDollars = pOriginal * pctDown                pctDownDollars = pOriginal * pctDown                 219
principal = pOriginal - pctDownDollars              principal = pOriginal - pctDownDollars               220
```

Figure 9. Retrieving a field value from the validation event

That sort of change continues on for the remainder of the input controls. Once the routine has processed all of the input values, it updates the screen with the results. One last change is necessary, as form validation requires the program to accept or deny the validation event. This is due to form validation locking the top-level window from user changes while the program processes the form. In order for BBj to unlock the window and allow additional changes to the form, the program must call the **`accept()`** method as shown in **Figure 10.**

```
347  DONE_CALCULATING:                              DONE_CALCULATING:
348      REM Make everything visible and back to normal     REM Make everything visible and back to normal
349      myBusyIndicator!.setName("busyIndicatorInvisible")  myBusyIndicator!.setName("busyIndicatorInvisible")
350      myPrintButton!.setEnabled(1)               myPrintButton!.setEnabled(1)
351      myWindow!.setCursor(0)                     myWindow!.setCursor(0)
352      myBusyIndicator!.setVisible(0)             myBusyIndicator!.setVisible(0)
353                                                 myCalcButton!.accept(1)
354      return                                     return
```

Figure 10. Calling the accept() method to complete form validation

After making the changes in the program from querying the controls directly to using form validation, we ran comparison tests to

a remote server to see the effect. Modifying the program to use form validation turned out to be a winner, making the program more than 11 times faster when retrieving the contents of the controls. That speed improvement resulted from a test using a fairly low-latency connection with a server that was approximately 1,000 miles away with an average of 45ms ping times. In higher latency scenarios, the speed improvement was even more significant.

Optimizing Layout - Screen Detection for Mobile Browsers

Initially, we embedded the new download page in an IFrame on the BASIS website, which we built with a Drupal-based CMS system. The main page, shown inside a desktop browser in **Figure 11**, contains the menuing system as well as the left and right navigation bars. In order to maximize the space available for the BUI application when running on a mobile device, we took advantage of pre-written code available from detectmobilebrowsers.com. It offers code in more than 15 languages to detect if the web page is being loaded into a mobile browser or not. We inserted the PHP code into our Drupal page and redirected the client to the BUI-only version of the app if viewing the page in a mobile browser. This maximizes the amount of space for the application on smaller devices such as tablets and smartphones.

Although it may seem odd at first to download BBj to a phone or tablet, many mobile devices and operating systems offer the ability to save downloaded files to the cloud via applications like Dropbox. This capability allows you to download BBj from any Internet-connected device with a browser and save it to a single location in the cloud for subsequent access by numerous machines.

Mission Complete - Optimizations in Place!

Rewriting the BBj Download page was a smashing success by any measurement. The resultant new BBj code base was several times smaller, easier to write, test, and debug than the old multi-language code base. We eliminated several other languages, libraries, runtime servers, machine dependencies, and setup and configuration complexities. The new download page is faster, easier to navigate and use, and includes new features such as localization and real-time translations. Best of all, we accomplished all of this with our own toolset, proving that that BASIS' BUI technology is a perfect fit for your next web development project. ■

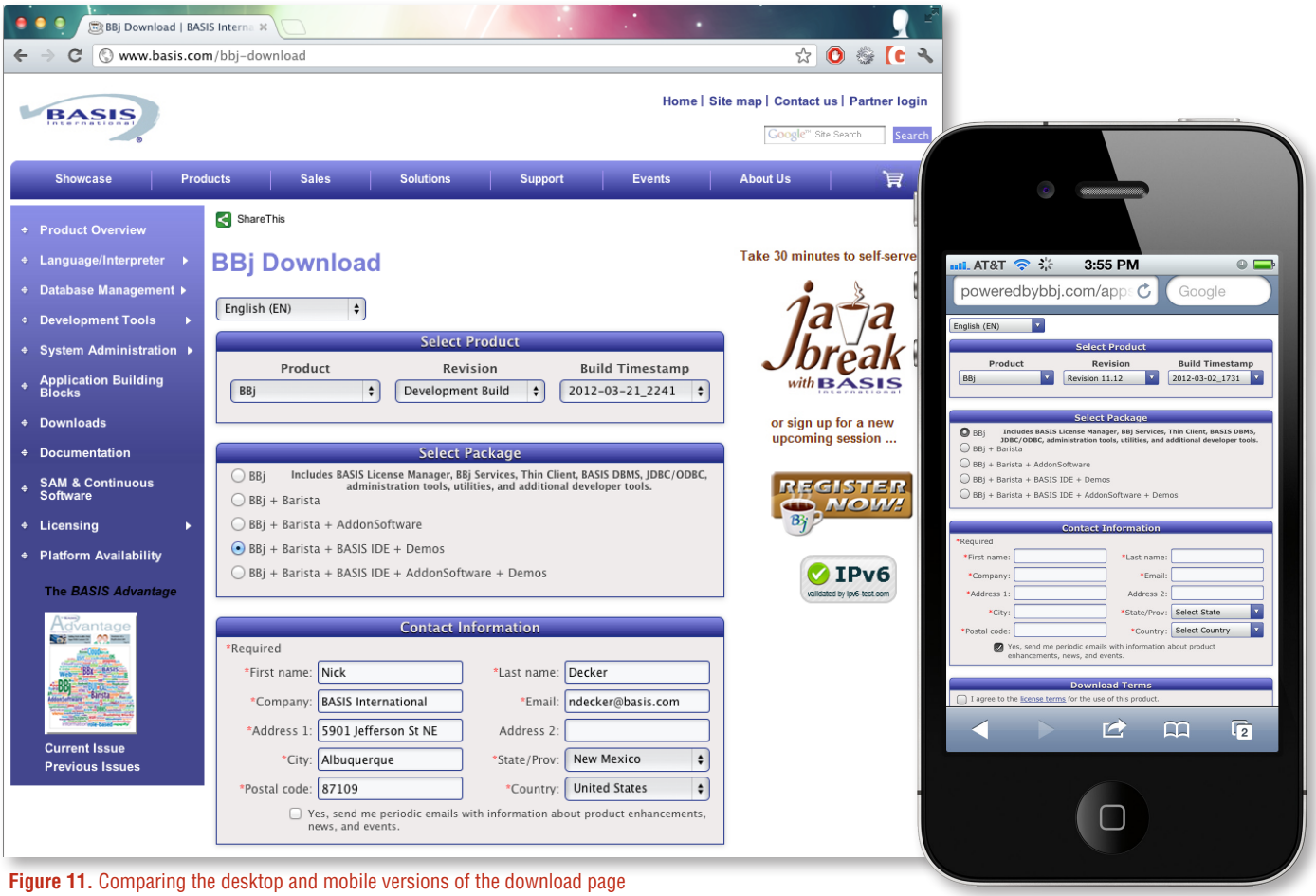

Figure 11. Comparing the desktop and mobile versions of the download page

- Read the *BASIS Advantage* article *Input Validation - Veto Power* at links.basis.com/05validation
- Learn more about the BBj methods and events mentioned in this article in the online documentation at links.basis.com/basishelp
- Optimizations do not end here - read more in this issue
  - *Looks Better, Runs Faster* at links.basis.com/12image
  - *GUI, BUI Everywhere* at links.basis.com/12gui
- The BUI Mortgage source code is installed with BBj as part of the demos package