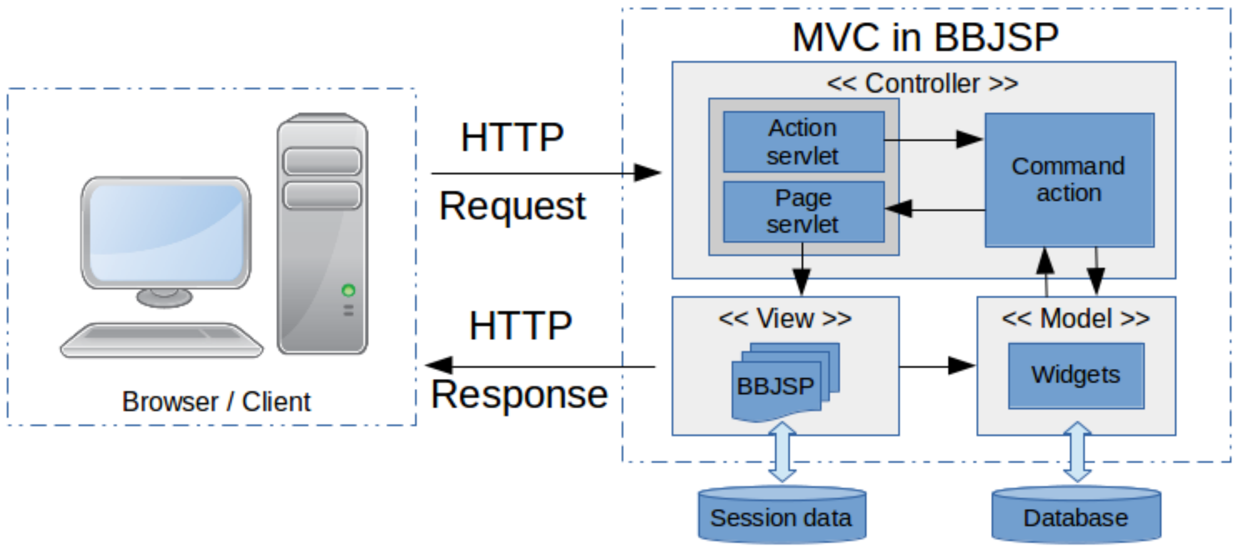


Getting Started with BBJSP

What is BBJSP and How to Use It

by Richard Stollar

You've probably heard of JavaServer Pages (JSP), a standard in today's Java stack for generating rich content-driven web pages. But did you know that we introduced a new feature called BBJSP to BBj® 16 which is much like JSP but designed for the BBj® programming language? This article will take you through the first steps with BBJSP and point you in the right direction for the configuration and setup process.

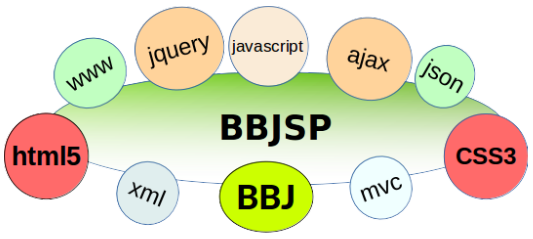
This is the first in a series of articles that will look at the BBJSP framework, with each building on what you have already learned. Through this process we will be building a full create, read, update and delete (CRUD) application that demonstrates the power behind BBJSP.

Overview

BBJSP closely follows the syntax of JSP. That way any developer with reasonable knowledge of the latter can quickly move into the former, and your organization can take advantage of a widely understood framework. Many of the concepts will be familiar to PHP and ASP developers too.

At its crudest level, BBJSP is a code generator that transforms HTML markup into executable BBj programs that run on the server-side to render content to the HTML client. The model-view-controller (MVC) architecture behind BBJSP allows for complete separation of the business logic from the presentation layer. Following this paradigm, you can divide the project into distinct areas allowing for more efficient use of developer resources within a wider team of designers and coders.

When you begin building your BBJSP application you'll probably want to add a new Context to the Jetty server as this will make laying out your pages much easier. If you're not that familiar with how contexts work then you may find one of the other articles in this series, *Contexts and BBJSP*, helpful. For the remainder of this article, we will assume that you are familiar with the contexts and have it all working.



With BBJSP you can embrace a wide range of open-stack technologies which have the best support for web application developers and open the door to new possibilities. As you gain experience with BBJSP you will find that integrating your existing code into a BBJSP application is a breeze.

Figure 1: Common technologies that you can leverage with BBJSP

Let's get going then!

Building a CRUD application

We're going to build a CRUD application that will demonstrate how you can use the different components of the BBJSP framework to update data.

Our CRUD application needs three main components: a list of records, a form for editing those records, and the business logic for interacting with the database. These three components fit together as shown in **Figure 2**.

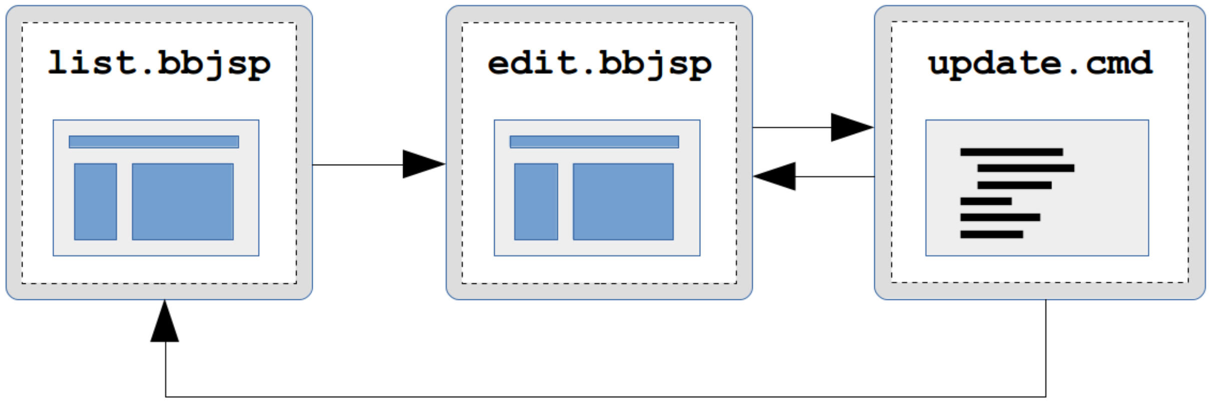


Figure 2: An overview diagram of the components of our CRUD application

You'll notice that the update component has two possible exits. A successful execution will return the user to the list page component, and any failure will go back to the edit page component. This allows for the update component to validate data before updating the database.

This article covers the first component in that process, a list page that displays a table of records from a database. We'll use the SALESREP table in the ChileCompany database for records since you probably have that installed. In a follow-up article we'll look at the steps involved in creating, updating and deleting records and take you through the full CRUD application which will add the final components as follows:

- A web page that shows a single record for editing
- A BBj® program that will handle updating the database entries

Presenting a Table of Records

Let's first create a basic page which has the table with a header row and one empty data row that we'll populate with records later. We added some CSS to embellish the table as shown in **Figure 3**, but this is not required.

```
<!DOCTYPE html/>
<html>
  <head>
    <style>
      table {
        border:2px solid black;
        text-indent: 4px;padding:0px;
        border-spacing: 0px;
      }
      th{
        border:0px;
        border-bottom:2px solid black;
        padding:4px;
        background-color:ABFBFF;
      }
      td{
        border:0px;
        padding:1px;
        background-color:E3FEFF;
      }
      .lb {
        border-left:1px solid black;
      }
    </style>
  </head>
  <body>
    <table width='100%'>
      <tr>
        <th align='left'>Code</th>
        <th align='left' class='lb'>Name</th>
        <th align='left' class='lb' colspan='4'>Address</th>
        <th align='left' class='lb'>Phone</th>
        <th align='left' class='lb'></th>

      </tr>
      <tr>
        <td></td>
        <td class='lb'></td>
        <td class='lb'></td>
        <td></td>
        <td></td>
        <td></td>
        <td class='lb'></td>
        <td class='lb'></td>
      </tr>
    </table>
  </body>
</html>
```

Figure 3: The HTML template for our list page with CSS code

Now we need to connect that simple HTML table with the database and populate the rows.

One of the core components of BBJSP are tags which follow similar syntax to regular HTML tags, in that they have opening and closing elements like this: `<tagname> … </tagname>`. BBJSP processes the body of the tag based upon the rules for that tag. For example the core:iterate tag will repeat everything between the opening `<c:iterate`> and closing `</c:iterate>` tags.

Some tags don't have a body and for these tags you'd use the self-closing syntax like this: `<tagname />` with one such tag being the core:out tag which writes some value to the output stream.

Tags come in libraries and you must import the library before you can use the tag in a page. To import a tag library you need to specify where the library is and how you will reference it within the page.

In our example page, we'll need to use the core tag library, which provides basic functions like iteration (equivalent to the `FOR ... NEXT` loop in BBj), and the SQL tag library for reading the database.

Tag libraries are imported at the top of the page, normally before the opening `<html>` tag. Let's add the import tag to the top of our previous example like this:

```
<!DOCTYPE html/>
<%@ taglib uri='/WEB-CFG/tld/core.tld' prefix='c' %>
<%@ taglib uri='/WEB-CFG/tld/sql.tld' prefix='s' %>
<html>
```

Figure 4: An excerpt of the HTML template with import tags added

Now we can reference any tag within the core and SQL tag libraries using the prefix we specified.

> **NOTE:** The prefix is arbitrary but it's generally a good idea to settle on a convention that you will use in all of your pages.

We'll need to query the database to get a result set which we will use to populate the HTML table. The SQL tag library provides the functionality for executing this query. As a general principle, it's a good idea to keep things together by placing the query tag close to where you will be writing the data; just before the beginning of the table will be a great place. **Figure 5** shows the code snippet that reads the database.

```
<s:query template='tpl' var='result' datasource='ChileCompany' sql='SELECT * FROM SALESREP' />
```

Figure 5: The sql tag responsible for querying the database

NOTE: We used the '`s`' prefix that we specified in **Figure 4** for the SQL library as this tells the internals of the framework how to find the tag, in this case the `query` tag, and enables validation of the attributes.

From the `query` tag, the `datasource` and `sql` attributes will be quite self-explanatory but the other two attributes are less obvious. The `var` and `template` attributes are name-bound variables (keep reading for more on that one). They tell the BBJSP framework how we will reference the result set and the template of the result respectively in the rest of the page. The data bound to `result` (the value of the `var` attribute) will contain a BBjVector of records in the templated format. The template will be in the name-bound variable `tpl` as specified in the `template` attribute.

Name-bound Variables

BBJSP uses name-binding to store data; which means that you don't create variables with names like `foo$` but just `foo` which can be referenced in expressions. Once you've specified the name of something, it's useage becomes an expression like ${something}. These variables only exist during the execution of the page.

Next, we need to iterate through the elements in the name-bound variable by using the `iterate` tag from the core tag library to fill in the row data from the entries in the result set. For that, we'll wrap the table row in an open-close iterate tag as shown in **Figure 6**.

```
<c:iterate data="${result}" id="item" >
  <tr>
    <td></td>
    <td class='lb'></td>
    <td class='lb'></td>
    <td></td>
    <td></td>
    <td></td>
    <td class='lb'></td>
    <td class='lb'></td>
  </tr>
</c:iterate>
```

Figure 6: The HTML table row wrapped in a core iterate tag

That's still not exactly what we want because all that will produce is several empty rows since we haven't filled the table cells with anything. But before we do that, let's think about the purpose of the iterate tag. The `data` attribute tells the tag what data we'll be processing and the `id` attribute says how we want to reference the current element while processing.

In our example, we're telling the framework to send whatever data has already been bound to the name `result` (identified by the `${result}` expression) to the `data` field of the iterate tag. The `iterate` tag will process all the elements in the data one-by-one, binding each element to the name `item` as it goes through the resultset.

However, the query template bound a string containing the format of the data to the name `tpl`. We will need to convert the formatted string into a layout for our page, which we do with the `template` tag from the `core` library. **Figure 7** shows how we use it.

```
<c:template id='theRecord' template='${tpl}' data='${item}' />
```

Figure 7: The template tag that is responsible for formatting each record

This tag will take the data identified by the name-bound variable `${item}` and apply the string template identified by the name-bound variable `${tpl}` to produce a data structure which BBJSP uses to fill each HTML table cell. To make the final presentation of fields in each record we can now access them as elements in a new name-bound data element `theRecord`. **Figure 8** shows the final code block that will do that:

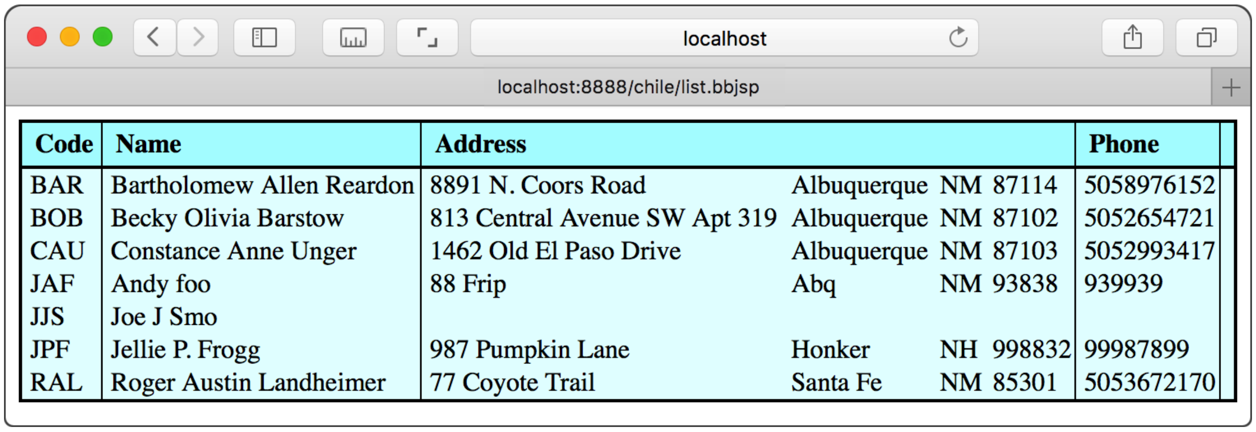
```
<c:iterate data="${result}" id="item" >
  <c:template id='theRecord' template='${tpl}' data='${item}' />
  <tr>
    <td>${theRecord['SALESPERSON']}</td>
    <td class='lb'>${theRecord['NAME']}</td>
    <td class='lb'>${theRecord['ADDRESS']} ${theRecord['ADDRESS2']}</td>
    <td>${theRecord['CITY']}</td>
    <td>${theRecord['STATE']}</td>
    <td>${theRecord['ZIP']}</td>
    <td class='lb'>${theRecord['PHONE']}</td>
    <td class='lb'></td>
  </tr>
</c:iterate>
```

Figure 8: The completed iterate block used to fill the HTML table

You'll notice that we can reference each of the fields from the result set by the column name used in the table. `${theRecord}` is actually a HashMap which stores the individual fields, making it possible to extract the phone number from the record using syntax like `${theRecord['PHONE']}`.

Summary

We've taken the first steps in creating a CRUD application but there's still plenty of work left for us to do. Currently our application will display the contents of the database in an HTML table, as shown in **Figure 9**, but this doesn't go far enough. In an upcoming article we will continue developing our CRUD application to edit these records.



Code	Name	Address				Phone
BAR	Bartholomew Allen Reardon	8891 N. Coors Road	Albuquerque	NM	87114	5058976152
BOB	Becky Olivia Barstow	813 Central Avenue SW Apt 319	Albuquerque	NM	87102	5052654721
CAU	Constance Anne Unger	1462 Old El Paso Drive	Albuquerque	NM	87103	5052993417
JAF	Andy foo	88 Frip	Abq	NM	93838	939939
JJS	Joe J Smo					
JPF	Jellie P. Frogg	987 Pumpkin Lane	Honker	NH	998832	99987899
RAL	Roger Austin Landheimer	77 Coyote Trail	Santa Fe	NM	85301	5053672170

Figure 9: The HTML browser output from our new application

Check out the example available for download at links.basis.com/16code
For more information on contexts, refer to:

- http://documentation.basis.com/advantage/v18-2014/14jetty.pdf