



Client Framework 4.4.7

Beginner Tutorial

Table of Contents

Chapter 1. Introduction	3
1.1 About the Beginner Tutorial	3
1.2 Intended Audience	3
Chapter 2. Understanding the Basics	4
2.1 Backbase Tag Libraries and Markup Languages	4
2.2 The MPI/SPI Model	4
2.3 HTML and XHTML	5
Chapter 3. Enabling the Framework	6
3.1 Building Blocks for Enabling the Framework	6
3.2 Basic Startup Page	6
Chapter 4. Ajaxifying an Application	9
4.1 Introducing the Example Application	9
4.2 Partial Page Reload	11
4.3 Optimizing Client Runtime Processing	15
4.4 Conclusion	17
Chapter 5. Enhancing a Form	18
5.1 Enhancing the Form with a Spinner	19
5.2 Enhancing the Form with a Slider	19
5.3 Changing the Image Size	20
5.4 Validating Input Fields	21
5.5 Processing the Form Input	22
5.6 Conclusion	23
Chapter 6. Contact & Support	24
6.1 Support	24
6.2 Training	24
6.3 Sales	24

Introduction

About the Beginner Tutorial

This **Beginner Tutorial** will help you understand the components of the Backbase *Client Runtime*, its tag libraries, and markup languages. Basic concepts of *Asynchronous JavaScript and XML (AJAX)*, *Single Page Interface (SPI)*, the *Backbase Tag Library (BTL)*, and the *XML Execution Language (XEL)* are discussed in this guide, as well as a demonstration on how to create a simple *Rich Internet Application (RIA)*.

The goal of this Beginner Tutorial is to create a fully functional application with a server-side response. In case you do not have the necessary software to run one of the server-side scripts provided, you can still follow along using the mock-up *HTML* response.

The tutorial is divided into three parts:

1. The first part shows you how to create an application startup file using *BTL*, which you can use in future projects.
2. The second part introduces an order application page with a standard three-column layout.
3. The third part shows you how to replace some of the input widgets in the form with cleaner and clearer *BTL* widgets, and how to add client-side behavior, such as validation of a form field.

Before beginning the tutorial, refer to the **Technical Overview** to become familiar with the concepts behind the *Backbase Enterprise Ajax 4.4.7 - Client Framework* product.


Intended Audience

This **Beginner Tutorial** is intended for application developers starting to create *RIAs* using the *Client Framework*.

 Refer to the *Prerequisites* section in the **Application Development Guide** to find pointers to the software you may use, the skills you should have, and other developer resources.

Understanding the Basics

With the *Client Framework*, Backbase provides you with an easy method to use *Asynchronous JavaScript and XML (AJAX)* technologies and create applications based on the concept of a *Single Page Interface (SPI)*. You can find a product overview in the **Technical Overview**.

AJAX is a web development technique for creating interactive web applications using a combination of technologies. Wikipedia provides a detailed overview on *AJAX* (<http://en.wikipedia.org/wiki/AJAX> .



Backbase Tag Libraries and Markup Languages

The key insight into understanding how to develop Backbase applications is the understanding that you will be developing *HTML* documents in the same way as you did before. However, some parts will be Backbase-enabled using standard *XML* technology. There are several tag libraries and markup languages that you can use, which are processed by the *Client Runtime*:

- The Backbase Tag Library (BTL) provides you with reusable, extensible, out-of-the-box UI widgets that take away all the hassle of cross-browser problems and facilitate Rapid Application Development (RAD). These widgets use a declarative model familiar to anyone who knows standard HTML. The UI widgets are supplemented by behaviors, such as dragAndDrop and resize, which can be applied to different markup languages.
- The declarative XML Execution Language (XEL) provides you with an application-level alternative to JavaScript, and manages asynchronous operations that are difficult to program and manage using JavaScript. Within XEL, you can also use declarative Command Functions that facilitate complex application functionality.
- The binding Tag Definition Language (TDL) lets you extend BTL UI widgets, add skins and new language implementations, and create your own widgets or XML languages.

Namespaces are used to separate multiple syntax specifications. In most cases, a *Uniform Resource Identifier (URI)* is used as the namespace. Sometimes opening this *URI* in a browser will direct you to an online syntax specification. Proper namespace declaration incorporates a document type definition, which will allow editors and browsers to validate *XML* source code.

When using the Backbase implementation of the *XHTML* namespace, many *XHTML* elements and corresponding attributes are allowed within your *XML* file. Refer to the appendix of the **Application Development Guide** for an overview of supported elements and attributes.

 For more details about the architecture of the Backbase *Client Runtime*, refer to the **Technical Overview** and the **Application Development Guide**. An overview of the various technologies and elements is available at bdn.backbase.com/client .

The MPI/SPI Model


In a classical web application, any user action that requires communication with a server (for example, to store or retrieve data from a database) would result in a complete page refresh in the browser. This is known as the *Multi-Page Interface (MPI)* application model.

Using *AJAX* technologies, it is possible to load the page layout and basic look only once, and then refresh only those parts of an application that need to be updated as result of a user action. This is known as the *SPI* application model.

You can use the *Client Framework* in both models. The greatest benefit of employing the *Client Framework* will be when you transform an existing *MPI* application to a *Rich Internet Application (RIA)* using the *SPI* model.

HTML and XHTML

The Backbase-enabled sections must contain valid *XHTML*. If your application contains legacy *HTML* that would not be valid as *XHTML*, you can still enhance your application with *BTL* widgets and migrate it to an *SPI*. This enhancement can be done in two ways:

1. Encapsulate the sections where you would like to use BTL widgets, or where you would like to use dynamic and partial page refresh, within `script` tags, as shown in the next chapter.
2. Dynamically enhance existing HTML elements using the TDL. This is a more complicated process, but it has the advantage that original code need not be touched. An example of this process can be found in the *Progressive Enhancement Demo*, which is available at bdn.backbase.com/client .

Part 2 and part 3 of this tutorial show an example of the first method of enhancement.

Enabling the Framework

This chapter explains what you need to include in your code to make it Backbase-enabled. You can make a start-up page that can be used as a skeleton for your own development. All elements from the Client Framework are explained briefly to place the functionality in context. For more details, refer to the **Application Development Guide** and the **API Reference**.

If you are interested in making a *JSP* project, start a new Backbase project. The files you create should be placed in the `WebContent` folder. For other kinds of projects, use the Eclipse facilities for that environment. For example, if you are using *PHP*, install one of the *PHP* plugins and use the Client Framework within that context.

Building Blocks for Enabling the Framework

To include a *BTL* widget, or to use any of the dynamic Client Framework facilities, you must start the Backbase *Client Runtime* by simply adding this line to the `head` section of your *HTML* page:

Example 1 – Include Client Engine

```
<script type="text/javascript" src="../backbase/4_4_7/engine/boot.js">
</script>
```

The `boot.js` file is the *JavaScript* library that loads the *Client Runtime*. It must be loaded once for an *HTML* page. The path to the `boot.js` file is relative to the startup page. Ensure that you change the path to reflect the file structure on your web server.

The Backbase-enabled sections in your page must be enclosed in `script` tags.

Example 2 – Include Backbase-Enabled Section

```
<script xmlns="http://www.w3.org/1999/xhtml" type="application/backbase+xml">
  <!--
    Everything between these tags will be processed by the engine
  -->
</script>
```

The *Client Runtime* processes all `script` blocks in the page that have a `type` attribute set to `application/backbase+xml`. All code within these script blocks will be parsed by the *Client Framework* engine. This is where you will include all Backbase-enabled code. The *XML* fragments that you enclose in these `script` tags must be well-formed *XML*. Other content in your page is processed by the browser in a regular manner.

In the first `script` section in your page, you must include the Client Framework bindings.

Example 3 – Include Bindings

```
<xi:include href="../backbase/4_4_7/bindings/config.xml" />
```

The implementation file defines the elements and interfaces in the *XHTML* namespace. By including this file, the *Client Framework* engine can recognize these elements and process them appropriately.

Basic Startup Page

The following snippet contains a complete page that implements a click event handler. This page can be used to verify that your Backbase installation was successful.

Create a new file in your *IDE* or in a simple text editor. A typical name for this file is `index.html`, but any valid name is acceptable, as long as the extension is `html` or `htm`. Copy the snippet below and paste it into the document.

Example 4 – Confirm Backbase Installation

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html>
  <head>
    <title>Backbase - Introductory Tutorial</title>
    <script type="text/javascript" src="../../backbase/4_4_7/engine/boot.js"></script>
  </head>
  <body>
    <script xmlns:e="http://www.backbase.com/2006/xel"
      xmlns="http://www.w3.org/1999/xhtml" xmlns:xi="http://www.w3.org/2001/XInclude"
      type="application/backbase+xml">
      <xi:include href="../../backbase/4_4_7/bindings/config.xml" />
      <div>
        <e:handler event="click" type="text/javascript">
          alert('This is a working Backbase installation');
        </e:handler>
        Click me
      </div>
    </script>
  </body>
</html>
```

The page is set up as a normal *XHTML* page.

- It starts with the namespace declarations on the script block. The snippet includes the W3C *XHTML* (as default) and XInclude (xi prefix) namespaces, as well as the *XML Execution Language (XEL)* (e prefix) namespace created by Backbase.

Namespace declarations are required to create valid *XML* documents, so that the engine can process the page elements properly. They indicate to which tag library an *XML* element belongs; for example, `e:handler` belongs to the `http://www.backbase.com/2006/xel` namespace, which defines *XML Execution Language*.

- The `xi:include` tag includes `backbase/4_4_7/bindings/config.xml`, which contains definitions of the tags created by Backbase.

When using elements from the *XHTML* namespace inside the script block, you must include the Application Configuration File named `config.xml`, but only once to enable usage in all script blocks on the page. However, you must declare the *XHTML* namespace every time *XHTML* elements are used to ensure the validity of the *XML* fragment.

- The `div` tag is a standard XHTML element, but it is also part of the Backbase implementation of XHTML, and it is processed by the Client Framework engine. In this way the `div` element can be extended with extra functionality.
- In the example, the `div` widget is extended by adding an event handler using the `e:handler` tag. Event handlers allow you to respond to events that occur on a page or element instance. Code within the handler is executed when the specified event type is triggered. The handler in our example listens to events that occur on the parent widget (the `div` widget). The `event` attribute specifies which event is handled (in this case, the `click` event).
- By setting the `type` attribute to `text/javascript`, you can use regular *JavaScript* statements inside the `e:handler` widget.

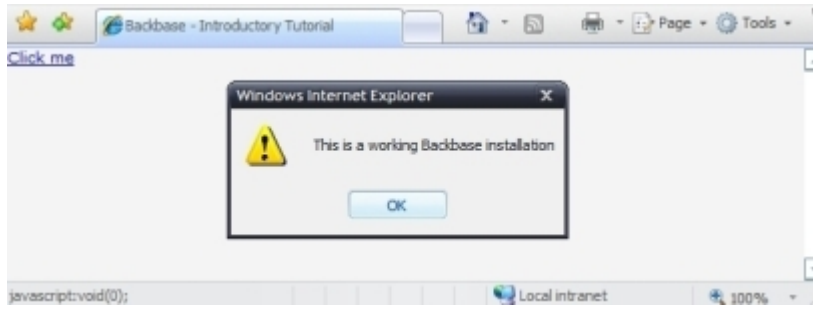
When you set the `type` attribute to `application/xml`, you can use *XML*-based execution languages like *XEL*.

- When clicked, the event handler shows an alert box with the message "This is a working Backbase installation".
- After closing the event handler, further content of the `div` widget can be added. In this case, we only add the text `Click me`.

i You must include the namespace declarations once in each *XML* document (file), preferably on the root tag of the document.

You can test this page by saving it to your local file system and opening the file in your browser. Click the text in the left corner of your browser window. If your installation was successful, an alert box appears.

Figure 1 Basic Startup File - Working Backbase Installation



The alert box signifies that your Backbase installation is working and you have successfully created a fully functional web page using the *Client Framework*.

To execute your new application in **Eclipse**, right-click the name of the file you created and choose: **Run as... -> Run On Internal Server**. Your default browser will start and display the application page as described above. Eclipse is most beneficial when you develop *JSP* or *JSF* applications. If you are using *PHP* or some other scripting language, you should install additional plugins, or run and test your application outside Eclipse. There are some hints on running *PHP* later in this tutorial.

Ajaxifying an Application

Using *AJAX* functionality as provided by the *Client Framework*, you can make *Rich Internet Applications (RIAs)*, that are more responsive and user friendly. Of course, you can start from scratch and create a completely new application; when this is appropriate, you should certainly do it. However, there are also ways that you can improve existing applications. Because the principles for both approaches are the same, we take the more adventurous and interesting route of trying to **Ajaxify** an existing application.

This chapter shows you how to make simple modifications to an existing application to achieve better responsiveness by preventing total page reloads. For more information, refer to the **Technical Overview**.

Introducing the Example Application

For the following example application, you must open the `index.html` file that is in the `tshirt-classical` folder. This example application consists of a page with a three-column layout: a menu section at the left, content in the middle, and extra information or advertisements on the right.

Figure 2 Traditional Application



The welcome page for this application is displayed in Figure 2. By clicking any of the menu items, you can navigate to other sections; for example, you can bring up an order form that can be used to order T-shirts. The form contains only ordinary *HTML*.

Figure 3 The Order Form



After you have filled in some data, click the submit button, and a response is displayed in the middle column.

Figure 4 Response after Submit



When you try out this application, you will see a page that only changes in the middle to show new content. In our case, the updates are quick, because this is a small page. Therefore, it looks as if the left and right side of the page remain static.

i A little red icon starts blinking for a few seconds every time the entire page is refreshed.

Apart from this blinking, each reload is barely noticeable for a small application. Imagine that you have a real application with many images on it: even if the browser does proper caching, a reload will visibly take some time.

You can find the code for the complete example in the `examples` folder of your *Client Framework* installation.

Before learning how to **Ajaxify** an application, you can try out the code in the `tshirt-ajaxified` folder. The functionality is the same as for the classical web application. It also looks exactly the same, except for two things:

1. The little red icon blinks just once when you load the page for the first time. The remainder of this chapter shows you how to prevent total page reloads.
2. In addition to the Single Page Interface (SPI), the enhanced form has more efficient widgets and interactive functionality, like validity checking.

Classic Example Application Source Code

Below is a snippet that contains the part of the classic *HTML* page that is enclosed in the `body` tags.

Example 5 – Classic `index.html`

```
<div id="bodydiv">
  <div id="left-float">
    <div id="middle-content">
      <h1>Welcome to our T-shirt Shop</h1>
      <!-- The welcome text goes here -->
    </div>
    <div id="side1">
      <ul>
        <li>
          <a href="index.html">Home</a>
        </li>
        <li>
          <a href="shirtForm.html">Order Form</a>
        </li>
        <li>
          <a href="about.html">About Us</a>
        </li>
        <li>
          <a href="contact.html">Contact</a>
        </li>
      </ul>
      
    </div>
  </div>
  <div id="side2">
    <!--
      The extra text or advertisements go here
    -->
  </div>
</div>
```

When the page is first loaded, the `div` element with `id="middle-content"` contains welcome text (not shown here). When you click on any of the menu items, the middle section is replaced by either the form or other text. The left and right sections stay the same, because we carefully crafted the pages that are loaded this way. The `href` attributes in the `a` (anchor) tags indicate that total page reloads are done. Looking in the `tshirt-classical` folder, you will find five *HTML* files, each representing a complete page that is loaded in response to clicking a menu item or submitting a form: `index.html`, `shirtform.html`, `response.html`, `about.html`, and `contact.html`. The files are identical, except for `id="middle-content"`.

To a developer, it may seem that maintaining this code is not too difficult, especially if you are using a server-side scripting language, such as *PHP* or *JSP*, which allow modularity of the code by dynamically building the page *at the server*, and then assembling its various parts. However, from a browser's perspective, there are five different *HTML* pages, which may result in a sluggish or flickering appearance to the end user.

Furthermore, the server script that assembles the page may become very complex for large applications with many pages. If the client page controls part of the interaction, then server scripts that can respond with limited knowledge and are therefore simpler.

Partial Page Reload

The previous chapter showed you how to Backbase-enable parts of the page by loading the *JavaScript* libraries and initial bindings of the framework. The resulting **Ajaxified** form application, as applied to `index.html`, is shown below.


Example 6 – Enable the HTML Page

```
<!-- --><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
<html xmlns:e="http://www.backbase.com/2006/xel"
xmlns:c="http://www.backbase.com/2006/command" xmlns:xi="http://www.w3.org/2001/XInclude">
  <head>
    <title>Backbase - Introductory Tutorial - Using Ajax</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="css/layout.css" />
    <script type="text/javascript" src="../backbase/4_4_7/engine/boot.js">
    </script>
  </head>
  <body>
    <script type="application/backbase+xml">
      <xi:include href="../backbase/4_4_7/bindings/config.xml" />
    </script>
    <!--
      the content of the page goes here
    -->
  </body>
</html>
```

The main content of the page is not enclosed in `script` tags. It is tempting to put the `script` tag as the first and only child within the `body` tag: this would make your whole page Backbase-enabled, and it would make coding easier, since you do not have to ask yourself whether the space must be Backbase-enabled. In this example, we chose not to put all code within *one* `script` element, because processing of *XML* elements by the *Client Runtime* causes unnecessary overhead that can be avoided.

Notice also that the namespaces are declared directly in the `html` tag, on the element in the *DOM* tree nearest to the root. If you are including or loading documents dynamically, you should put the necessary namespace declarations for each document in the root-tag of that document.

You can restrict Backbase-enablement to areas where it is needed (sections that contain *BTL* elements, *XEL* functions, or *TDL* widgets), and leave other parts of your application (sections that only contain standard *HTML*) untouched, even if their content is dynamically loaded using Client Framework functionality. For our simple application, you may not see a difference in response times, but for larger applications, the performance gain can be significant.

 You can have as many Backbase-enabled sections in your page as you like.

Backbase-enabled Areas

To use a tag from the *BTL*, *XEL*, *Command Functions* or *TDL*, you must include it within `script` tags with `type="application/backbase+xml"`. With the *JavaScript API*, you can use Client Framework functionality anywhere where *JavaScript* coding is appropriate. Whether you use the declarative *XML* style of code that the Backbase markup languages offer, or the *JavaScript* style with the *JavaScript API*, is simply a matter of preference and experience. Generally, the *JavaScript API* performs better, because it requires less overhead. On the other hand, the simplicity and clarity of declarative code may help you construct applications in less time with fewer errors.

In our application, we enclose two areas in `script` tags.

1. One of the menu items (*Home*) is Backbase-enabled to show how you can use *XEL* and *Command Functions* to dynamically load content when a user clicks an item.

Example 7 – The Home Menu Item

```

<li>
  <script type="application/backbase+xml">
    <a href="javascript://">
      <e:handler event="click">
        <c:load url="welcome.xml" destination="id('middle-content')">
          mode="replaceChildren" />
        </c:load>
      </e:handler>
      Home
    </a>
  </script>
</li>

```

The `c:load` command invoked by the click event handler specifies that the `welcome.xml` file is loaded, and its content has replaced everything within the `div` element with `id="middle-content"`.

The `mode` attribute can have the following string values:

- `replace` - replaces the selected destination node.
 - `replaceChildren` - replaces its children.
 - `firstChild` - places as the first child.
 - `lastChild` or `appendChild` - appends to the selected destination node (default).
 - `insertBefore` - inserts before the selected node.
 - `insertAfter` - inserts after the selected node.
2. We need a second Backbase-enabled area to show the form, because the enhanced version of the form uses *BTL* elements. The same `div` element can contain the text that appears when the *Home*, *About Us*, or *Contact* menu items are clicked.

Example 8 – Script Tag to Enable Content

```

<script id="xml_space" type="application/backbase+xml">
  <div id="middle-content">
    <!-- Backbase enabled space. -->
  </div>
</script>

```

As long as the alternative content is *XHTML*-compliant, nothing else is required to execute this application properly. However, the *Client Runtime* causes overhead that can be avoided by optimizing this processing, which is discussed later in this chapter.

Using the JavaScript Backbase API

When the *Client Framework JavaScript* libraries are loaded, you have the complete functionality of the framework at your disposal in the same way as you can use the functionality of any *JavaScript* library. Most of the functionality is available with methods on the **bb JavaScript object**.

As an alternative to the *XEL* click event handler and the `c:load` command function, you can use the `bb` object by invoking the `command.load` method on it. Here is a code snippet for this method, which works in the same way as the one for the *Home* menu item, except that it loads the *About Us* text when the menu item is clicked.

Example 9 – The About Menu Item

```

<li>
  <a href="javascript://" onclick="bb.command.load('about.xml', 'GET', null, null,
    bb.document.getElementById('middle-content'),'replaceChildren');"> About us </a>
</li>

```

Straightforward *JavaScript* is used, so that you do not need to enclose the code in `script` tags. On the other hand, it is less clear what the code does. Refer to the **API Reference** for more details.

For the other menu items, we show some variations on this theme. With two ordinary *JavaScript* functions directly below the script that loads the *Client Runtime*, the code becomes slightly more readable, which will ease further customization. The first function (`load_xml`) loads content in Backbase-enabled space. This function's body contains the code for the `onClick` event of the *About* menu item. The second function (`load_html`) loads content into any space that is not Backbase-enabled. The first function uses the `bb` object to address the Backbase-enabled space, while the second function finds the element in the *DOM* tree that any browser can recognize.

Example 10 – Load Helper Functions

```
<script type="application/javascript">
  function load_xml(sUrl, sTarget){
    bb.command.load(sUrl, 'GET', null, null,
      bb.document.getElementById(sTarget), 'replaceChildren');
  }
  function load_html(sUrl, sTarget){
    // generate the html at the target location.
    bb.command.load(sUrl, 'GET', null, null,
      document.getElementById(sTarget), 'replaceChildren');
  }
</script>
```

These functions are called by clicking a menu item. The code for the *Contact* menu item used the `load_xml` function.

Example 11 – The Contact Menu Item

```
<li>
  <a href="javascript://" onclick="load_xml('contact.xml', 'middle-content')"> Contact
</a>
</li>
```

The code for the *More...* menu item uses the `load_html` function. It loads content into the right side of the page, which was considered static.

Example 12 – The More Menu Item

```
<li>
  <a href="javascript://" onclick="load_html('more.xml', 'more')"> More... </a>
</li>
```

Ajaxifying the Response Files

When you click a menu item and an *AJAX* request is sent to the server, you must simplify the *HTML* response file by removing everything around the middle content.

Example 13 – Response for Welcome Text

```
<div>
  <h1>Welcome to our T-shirt Shop</h1>
  <!-- more text here -->
</div>
```

Ensure that the response is interpreted as *XML* by the browser. If you are returning static text, and the file extension is `.xml`, then the browser will assume that it is *XML*. If the response is the result of server-side script processing, the script needs to return the **appropriate headers** with `Content-type` set to `application/xml`. The scripts provided with the code in this tutorial illustrate this point.

Ajaxifying the Form

This section shows you how the response received after a form submit can update only part of the page, instead of refreshing the complete page. To accomplish this, add a `bf:destination` and a `bf:mode` attribute.

Example 14 – Set Destination and Mode on the Form

```
<form class="demo_form" action="response.xml" bf:destination="id('middle-content')"
bf:mode="replaceChildren" method="post" title="Form" id="form">
  <!--
    the contents of the form go here
  -->
</form>
```

i The Client Framework `form` implementation requires its own namespace. You must add `xmlns:bf="http://www.backbase.com/2007/forms"` to the namespace declarations in the *XML* file where the `form` is used.

The `bf:destination` attribute specifies where the *Client Runtime* will place the response received from the server. In our example, this is `id('middle-content')`, where most of the dynamic content is located. The value of the `bf:mode` attribute is `replaceChildren`.

If you replace this value with another (such as `lastChild`), the response text will appear below the form. Each time you click the submit button, the response text will be repeated. If you try the `replace` value, the whole node will be replaced. Seemingly, this will work, but since the `div` with `id('middle-content')` is removed, an error will occur when you click one of the menu options.

Optimizing Client Runtime Processing

If the `script` tag in the middle section encloses the bulk of the application code, it may seem like the performance gain will be minimal. There are two ways to minimize this overhead:

1. Use special tags to indicate that a section of *HTML* need not be processed by the *Client Runtime*. This is used in our **Ajaxified** application.
2. Use `b:xhtml` escape tags to make your application run even faster. This requires more advanced knowledge of the Client Framework.

Using b:xhtml Escape Tags

Most of the *Client Runtime* processing overhead can be avoided for tags that do not need it. The following snippet shows the content of the `welcome.xml` file.

Example 15 – XML Containing Welcome Text

```
<b:xhtml>
  <div>
    <h1>Welcome to our T-shirt Shop</h1>
    <!-- more text here -->
  </div>
</b:xhtml>
```

The static *HTML* text is enclosed with `b:xhtml` tags. This signals the *Client Runtime* to leave the children of this tag alone and allow the browser to process it. If there are some Client Framework features within this area that you would like to use, enclose them in `b:xml` tags.

Example 16 – Jump Back to XML Space

```
<b:xhtml>
  <!-- text here -->
  <b:xml>
    <b:calendar />
  </b:xml>
  <!-- more text here -->
</b:xhtml>
```

Within a `b:xhtml` element, a `b:xml` element will force processing by the *Client Runtime*. The code that is enclosed with `b:xhtml` tags must be *XHTML* compliant. If this is not the case for your application, there are advanced features to deal with this compliancy, such as the `enhanceHTML` and related elements provided with the *BTL* language. You can target elements outside Backbase-enabled space using the function that handles the *More...* menu item. This space can contain any non-compliant *HTML*, as long as the browser understands it.

Optimizing Client Runtime Processing Alternative

This optimization method is a more involved way of separating the Backbase-enabled parts from the rest of your application. The middle section can contain Backbase-enabled *XML* when the form is displayed, or just *HTML* when any other menu option is clicked. This example shows two separate areas where each type of code can be loaded.

Example 17 – Separate XML and HTML Sections

```
<div id="middle-content">
  <div id="html_space"> </div>
  <script id="xml_space" type="application/backbase+xml">
    <!--
      Backbase enables the use of true xml. This is a space to put it.
    -->
  </script>
</div>
```

The functions used to load content into these pages are similar to the previously used `load_xml` and `load_html`. You must add extra functionality so that when content is loaded into one area, the other should be hidden.

Example 18 – Functions That Can Separate HTML and XML Space

```
<script type="application/javascript">
  function load_html(sUrl){
    document.getElementById('html_space').style.display = '';
    bb.document.getElementById('xml_space').viewNode.style.display = 'none';

    // When using command.load with an html destination, the responseText is used to
    // generate the html at the target location.
    bb.command.load(sUrl, 'GET', null, null,
    document.getElementById('html_space'),'replaceChildren');
  }
  function load_xml(sUrl){
    document.getElementById('html_space').style.display = 'none';
    bb.document.getElementById('xml_space').viewNode.style.display = '';
    bb.command.load(sUrl, 'GET', null, null,
    bb.document.getElementById('xml_space'),'replaceChildren');
  }
</script>
```

You can try this code by replacing the relevant parts in the `index.html` file, and by changing the menu items as follows.

Example 19 – Updated Menu Code

```
<ul>
  <li>
    <a href="javascript://" onclick="load_html('welcome.xml')"> Home </a>
  </li>
  <li>
    <a href="javascript://" onclick="load_xml('shirtForm.xml')"> Order Form </a>
  </li>
  <li>
    <a href="javascript://" onclick="load_html('about.xml')"> About us </a>
  </li>
  <li>
    <a href="javascript://" onclick="load_html('contact.xml')"> Contact </a>
  </li>
</ul>
```

You must remove the `b:xhtml` tags in the `.xml` files that contain the content that is loaded when you click any menu item other than the order form item. You must also change the value of the `bf:destination` attribute to `id('xml_space')`. This causes the response for the form submit to be written to the Backbase-enabled

`id('xml_space')` area. If the response does not need this functionality (as in our example), it is possible to submit the form to an *HTML* destination. This requires some more *JavaScript*, which falls outside the scope of this tutorial.

Initial Loading of Content

The `div` with `id="middle-content"` should initially display the welcome text. This text cannot be static, because it will be replaced by dynamic content when you click a menu item. The `Home` menu item should display the welcome text again. A script is included at the end of the `body` to initially load the welcome text.

Example 20 – Display Initial Welcome Text

```
<script type="application/backbase+xml" e:onload="load_xml('welcome.xml',  
'middle-content')"> </script>
```

The `e:onload` is used to trigger the execution of the `load_xml` function.

Conclusion

In this chapter, we discussed how you can **Ajaxify** an existing application. We have shown various ways that you can *optimize* this **Ajaxified** application by localizing *Client Runtime* processing to areas where it is needed.

Enhancing a Form

This chapter uses some of the *Client Framework* widgets to make an existing *HTML* form more user friendly. We will enhance the `form` that was part of the application described in the previous chapter. The code for this form is shown below.

Example 21 – Part of Form to be Enhanced

```
<form class="demo_form" action="response.xml" bf:destination="id('middle-content')"
bf:mode="replaceChildren" method="post" title="Form" id="form">
  <div id="form-fields">
    <fieldset title="T-shirt Options" id="tshirtOptions">
      <legend>T-shirt Options</legend>
      <div class="row rowOdd">
        <div class="lspan">
          <label for="amount">Amount:</label>
        </div>
        <input class="inputText" type="text" name="amount" id="amount" size="1"
maxlength="1" value="1" />
      </div>
      <div class="row">
        <div class="lspan">
          <label for="size">Size:</label>
        </div>
        <select class="inputText" name="size" id="size">
          <option value="small">S</option>
          <option value="medium">M</option>
          <option value="large">L</option>
          <option value="extra large">XL</option>
          <option value="extra extra large">XXL</option>
        </select>
      </div>
    </fieldset>
    <fieldset title="Check Out" id="checkOut">
      <legend>Check Out</legend>
      <div class="row">
        <div class="lspan">
          <label for="name">Name:</label>
        </div>
        <input class="inputText" type="text" name="name" id="name" />
        <span> * </span>
      </div>
      <div class="row rowOdd">
        <div class="lspan"> </div>
        <button type="submit">Order the T-shirt</button>
      </div>
    </fieldset>
    <p>Fields marked with * are required.</p>
  </div>
  <!--
  the code that shows the T-shirt goes here
  -->
</form>
```

The form looks as the designer intended it by using *CSS* styling; however, it is static and not very intuitive. To enhance it and make it more interactive, you must customize it as follows:

- Replace the **Amount** field with a *spinner*;
- Replace the **Size** field with a *slider*, which will change the size of the T-shirt image when you move it;
- Specify the **Name** field as required, which will be checked when you try to submit the form.

The new version of the form has the same functionality as the old one. The next section describes what the *spinner* contributes to the user experience, followed by the *slider*, and then the client-side form validation. The result of these modifications is illustrated below.

Figure 5 Form with Spinner and Slider



Enhancing the Form with a Spinner

The **Amount** `input` field is replaced by a *spinner*. The user can set the number of T-shirts to be ordered without typing a number. The minimum and maximum values ensure that the user orders at least one but no more than nine T-shirts.

Example 22 – The Spinner Widget

```
<div class="row rowOdd">
  <label for="amount">Amount:</label>
  <b:spinner type="text" value="1" id="amount" name="amount" class="inputText" min="1"
    max="9" />
</div>
```

Enhancing the Form with a Slider

A *slider* is used instead of the drop-down list to indicate the desired size of the T-shirt. The label on the slider indicates the size. To visualize a change in size, the image of the T-shirt is shrunk or enlarged according to the *slider* position.

Example 23 – The Slider

```
<div class="row">
  <label for="size">Size:</label>
  <b:slider id="size" name="size" class="inputText" value="large">
    <e:handler event="slide" type="text/javascript">
      var value = this.getProperty('value');
      setImageSize(value);
    </e:handler>
    <b:sliderOption value="small">S</b:sliderOption>
    <b:sliderOption value="medium">M</b:sliderOption>
    <b:sliderOption value="large">L</b:sliderOption>
    <b:sliderOption value="extra large">XL</b:sliderOption>
    <b:sliderOption value="extra extra large">XXL</b:sliderOption>
  </b:slider>
</div>
```

The *slider* code has similarities to the code for the the drop-down list, such as discrete options for the possible values that the slider can have. This is an extra feature of the *BTL* version, whereas in a general slider, you must specify a range of numeric values.

The code shows an *event handler* that fires when a *slide* event occurs. This event handler gets the value of the slider and then calls the function `setImageSize` with the value as the argument.

For more information about the *slider* and event handlers, refer to the [API Reference](#) and the [Application Development Guide](#).

Changing the Image Size

This section explains how to use the events that result from interaction with the slider. For example, you can use these values to change the image size in the form to clarify which size you are ordering.

Example 24 – Part of Form Showing the T-shirt

```
<fieldset title="My Shirt" id="tshirtPrice">
  <legend>My Shirt</legend>
  
  <p id="priceContainer">
    <span>Price: &euro; 15,-</span>
  </p>
  <p>
    Cost includes handling and shipping. International shipment available.
  </p>
</fieldset>
```

The event handler for the *slider* calls the function `setImageSize`. This function changes the image size every time the *slider* position is moved.

The simpler approach would be to include the code that changes the image size directly in the event handler. However, our approach makes the code more modular, as expected in a real application. Therefore, we made a separate function that is called in the event handler, and put the definition of the function in an external file as an example of how you could develop your own function library. The file must be specified using *XInclude* right after the inclusion of the configuration file.

```
<xi:include href="myfunctions.xml" />
```

You can use the `setImageSize` function and other ones that you may need to define in the same way as in a *JavaScript* library.

Example 25 – Changing the Size of the Shirt Image

```
<e:xel>
  <e:function name="setImageSize">
    <e:argument name="shirtValue" required="true" default="'large'" />
    <e:body type="text/javascript">
      //CDATA
      var oShirt = document.getElementById('shirt');
      var iWidth = 187;
      var iHeight = 147;
      var oSizes = new Array();
      oSizes['small'] = 0.7;
      oSizes['medium'] = 0.8;
      oSizes['large'] = 0.9;
      oSizes['extra large'] = 1.0;
      oSizes['extra extra large'] = 1.1;
      var newSize = oSizes[shirtValue];
      oShirt.style.width = iWidth * newSize + 'px';
      oShirt.style.height = iHeight * newSize + 'px';
      //CEND
    </e:body>
  </e:function>
</e:xel>
```

Except for the body, which is plain *JavaScript*, the function is coded in *XEL* and is declared using the `e:function`. The `shirtValue` argument is declared using the `e:argument` tag, and the body is enclosed with an `e:body` tag. You should be familiar enough with *JavaScript* to understand the code in the body of the function.

You could write the function and event handler without *JavaScript* by using *XEL*. Depending on your preference, you can mix and match both languages. For more information about *XEL* and functions, refer to the **Application Development Guide** and the **API Reference**.

Validating Input Fields

Anyone who fills in a form is often confronted with the frustration of making mistakes, and must deal with a server that sends unfriendly error messages, and then wipes out the input that has already been entered. You can enhance the user experience considerably by checking the values in the form fields for validity *before* the form is sent to the server. The user should also see a meaningful error message when values are entered incorrectly, or when a value that is required was not provided.

Client Framework offers several built-in facilities to validate user input. There are also many ways to extend this validation, which are provided in examples in the demos as well as the *Client Framework* package.

This section only shows a very simple validation. An error message is displayed when you try to submit the form while the `name` field is not filled in.

Example 26 – Input Validation


```
<div xmlns:c="http://www.backbase.com/2006/command"
xmlns:xi="http://www.w3.org/2001/XInclude" id="center">
  <fieldset title="Check Out" id="checkOut">
    <legend>Check Out</legend>
    <div class="row">
      <div class="lspan">
        <label for="name">Name:</label>
      </div>
      <input class="inputText" type="text" name="name" id="name" bf:required="true"
bf:messagesRef="id('required_field')"/>
      <span> * </span>
      <bf:messages id="required_field">
        <bf:message event="invalid" class="errorMessage" facet="required">
          <div>This field is required.</div>
        </bf:message>
      </bf:messages>
    </div>
    <div class="row rowOdd">
      <div class="lspan"> </div>
      <button type="submit">Order the T-shirt</button>
    </div>
  </fieldset>
  <p>Fields marked with * are required.</p>
</div>
```

Two attributes (`bf:required="true"` and `bf:messagesRef="id('required_field')"`) are added to the `input` field. These attributes are part of the **form** support that *Client Framework* offers, in addition to the attributes of the `form` tag and the *messages* container.

The `required_field` identifier refers to a *messages* container, which displays or hides the message depending on the validity of the field when the user tries to submit the form. The *messages* container can have more than one if you want; for example, one for each kind of error that you would like to catch.

When the `invalid` event occurs, the message content is made visible. In our example, red text appears to inform the user that the field is required. You can try this by pressing the submit button without entering a value in the name field.

For more information about form validation, refer to the example applications provided with *Client Framework*. For details about attributes and their values, refer to the **API Reference**.

 Validating user input on the client side of your web-application does not make validation on the server superfluous. There is always the possibility that a malicious user could have tampered with the information sent to the server. The purpose of *client-side* validation is to make your site more user friendly and more responsive when a problem occurs.

When you fill in some values in the form and press the submit button, a simple static text is displayed in the middle of the page. The next section describes server-side processing.

Processing the Form Input

The current server response is a piece of static text. In reality, you would reply with more personal information, such as displaying an invoice for the ordered T-shirts. You would also need to keep the information provided by the user, usually by storing it in a database.

The following code contains a simplified *PHP* response, followed by a snippet for a *JSP* response. You can use these example as a skeleton for your own code. If you want to try this in your environment, replace the `action` attribute in the form with the name of the script you will call.

Example 27 – PHP Response (response.php)

```
<?php
header('Content-type: application/xml');
echo '<?xml version="1.0" encoding="UTF-8"?>' ?>
?>
<b:xhtml xmlns="http://www.w3.org/1999/xhtml"
xmlns:b="http://www.backbase.com/2006/btl">

<h1>Thank you for ordering at the Backbase Demo Shop</h1>
<fieldset
title="Form Submission" id="response" >
<h3>Form submission confirmation</h3>
<?php
echo "<p>Dear <b>{$_POST['name']}</b>";
$amt = $_POST['amount'];
if ($amt == 1) $shirtTxt = 'shirt';
else $shirtTxt = 'shirts';
echo "<br/>You ordered: <b>$amt</b> $shirtTxt.";
echo "<br/>The size of the $shirtTxt you ordered is: <b>{$_POST['size']}</b>.</p>";
?>
<p>
Your order will be processed and sent to the address provided.
</p>
</fieldset>
</b:xhtml>
```

Example 28 – JSP Response

```
<?xml version="1.0"?>

<div xmlns="http://www.w3.org/1999/xhtml"
xmlns:b="http://www.backbase.com/2006/btl"
xmlns:e="http://www.backbase.com/2006/xel"
xmlns:xi="http://www.w3.org/2001/XInclude">

<%
response.setHeader("Content-Type", "application/xml");
if (request.getParameter("name") != null {
    // more code here ...
} else {
    out.println("You must enter your name.");
    out.println("Click the Load Form link to reload the form.");
}
%>
</div>
```

i Note: The header information provided should have the `Content-type` set to `application/xml`. This ensures that your response can be processed by the *Client Runtime* as *XML*. The content of your response should also be valid *XHTML*. You should know how to configure the server to enable it to process your server-side script.

Conclusion

In this chapter, we showed how an existing form can be made more enjoyable to use by adding *BTL* widgets, dynamic behavior, and form validation. We also showed important aspects of *BTL*, *XEL* and **forms support** in *Client Framework*. This should provide you with enough background information to start exploring the other documentation, demos and examples provided with *Client Framework*.

Contact & Support

Support

Backbase offers free and paid support:

- **Backbase Developer Network**—visit the developer forum on the Backbase Developer Network to find a solution to your problem or post a new forum entry. Backbase experts regularly check the forum and respond to problems posted there. You can also find new releases and product updates at the Backbase Developer Network.

To visit the Backbase Developer Network, go to bdn.backbase.com.

- **Backbase Support**—if you require direct assistance from our support experts, you can subscribe to a Support Package. Companies with a Support Package have log in credentials to the Backbase Support website. At the Backbase Support website you can contact our support experts directly. For more information on the Support Packages, see www.backbase.com/services/support.

To log in to Backbase Support, go to www.backbase.com/supportcustomer.

Training

Backbase offers a training program either at a Backbase office or onsite.

For more information, go to www.backbase.com/services/training.

Sales

For sales related inquiries, you can find relevant contact information at www.backbase.com/contact.

You can also submit questions to our general email address at contact@backbase.com.