

Spice up PHP applications with OpenLaszlo, Part 1: Create interactive interfaces

Your first OpenLaszlo application

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)

Freelance writer

Backstop Media

14 Mar 2006

This "[Spice up PHP applications with OpenLaszlo](#)" tutorial series shows you how to use OpenLaszlo to create a more interactive interface for your PHP applications and how to use PHP to create more dynamic OpenLaszlo applications. It requires a basic understanding of -- or willingness to learn -- XML, JavaScript, and PHP. Each is well worth knowing for its own merits, plus they plug and play together nicely, since they're all based on synergistic open standards.

Section 1. Before you start

On the server side, PHP is a widely supported scripting language that produces and reads XML data, interfacing easily with relational databases and other programs. PHP is great for implementing AJAX applications with service-oriented architectures. And it's a smart choice for integrating Web services that use common Internet protocols, such as HTTP, XML-RPC, Simple Object Access Protocol (SOAP), and Representational State Transfer (REST).

On the client side, OpenLaszlo is an elegant XML-centric, JavaScript-based Web programming language. It's designed for implementing interactive, media-rich, distributed, user-friendly interfaces to XML-based Web services. For programmers, the OpenLaszlo language is fun and feature-rich. For users, OpenLaszlo applications are richer and more responsive than conventional browser-based

applications. (See [Resources](#) for available OpenLaszlo applications.)

About this series

This series chronicles the creation of a rating system for knock-knock jokes. Part 1 talks about the knock-knock protocol, and covers the creation of the basic PHP application and a simple OpenLaszlo interface, with graphics and sound, for that application. Part 2 will discuss deployment options, and show the creation of the OpenLaszlo application as a stand-alone piece that can be deployed on any Web server and communicate directly with a PHP application. Part 3 expands the application to create a system in which PHP creates the OpenLaszlo code dynamically.

About this tutorial

You will learn about OpenLaszlo, set up the development environment, and create a basic OpenLaszlo application that will serve as an interface for a joke rating service. OpenLaszlo is a free XML/JavaScript-based Web application programming language that enables you to write interactive multimedia applications that can ultimately be deployed to the browser in many ways. Currently, the target environment is a Flash movie, playable on virtually any modern browser without user intervention.

You will learn about:

- The sample knock-knock protocol
- What OpenLaszlo is and what it's for
- How to set up an OpenLaszlo development environment
- The basics of OpenLaszlo programming
- Running an OpenLaszlo application
- Adding interactivity, graphics, and sound to the interface

Prerequisites

You will need to install OpenLaszlo on your development machine (and optionally on your server), and PHP and Apache on your server (and optionally on your development machine). We will walk through installing OpenLaszlo, PHP, and Apache on your development machine:

- The [Apache Web server](#) is maintained and developed by the Apache

Software Foundation.

- You can download PHP for free from [PHP.net](#). Instructions here assume you are installing on Windows®, but the site has instructions for other platforms, as well.
 - Download [OpenLaszlo from Laszlo Systems](#). The basic instructions will be covered here..
-

Section 2. Getting ready

Before doing anything else, you need to get the development environment in place. If you already have Apache and PHP installed, feel free to skip ahead to [Configuring Apache](#).

Installing PHP

Start by installing PHP, including the free mod_php module, which extends the Apache Web server so it can execute PHP scripts efficiently. The following steps assume you are installing on Windows, but the process is essentially the same for other platforms:

1. Download the zip file distribution (see [Prerequisites](#)). This tutorial was tested with php-5.1.2-Win32.zip.
2. Extract the zip file into your destination directory, such as C:\php5. (Feel free to choose another directory, but just make sure it doesn't have spaces in it and modify the following directions accordingly.)
3. Make a copy of C:\php5\php.ini-dist and rename it C:\php5\php.ini.
4. Edit C:\php5\php.ini. Search for doc_root and change it to "C:\public_html". (Again, you may substitute accordingly.) Don't forget the quotes.
5. Create a directory for your HTML files called C:\public_html, where you will put the files you want to publish on the Web.

Next, install the Web server.

Installing Apache

Apache will act as a Web server for the PHP files and as a proxy for the OpenLaszlo applications. Download the Apache installer and run it (see [Prerequisites](#)).

In the Apache HTTP Server V2.0 Installation Wizard screen, set the following values:

Listing 1. Setting the values for Apache HTTP Server V2.0

```
Network Domain: localhost
Server Name: localhost
Administrator's Email Address: admin@localhost
Install Apache HTTP Server 2.0 programs and shortcuts:
for All Users, on Port 80, as a Service - Recommended.
```

Don't configure Apache to use port 8080 because that's the one OpenLaszlo is going to use. If you're running IIS or another Web server on port 80, you'll have to disable it or configure PHP to run with that Web server, instead of Apache.

Select the setup type "Typical" and install Apache HTTP Server V2.0 to the folder: C:\. Note that the installer always makes a subdirectory called Apache2, so if the installation directory is C:\, Apache will be installed in C:\Apache2. You can install it anywhere you like, but it's a good idea never to install PHP or Apache under a directory with spaces in its name (avoid C:\Program Files, for example).

Configuring Apache

Before installing OpenLaszlo, you need to make a few changes to the Apache configuration to enable it to run PHP more efficiently, as well as to act as a proxy so users who can't access the OpenLaszlo application on port 8080 can access it through Apache on port 80:

1. Make a backup copy and edit the file C:\Apache2\conf\httpd.conf.
2. Search for `DocumentRoot` and change it to the name of your HTML directory, C:\public_html.
3. Search for `LoadModule` and uncomment the following lines (by deleting the leading pound sign) to enable the following modules:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule rewrite_module modules/mod_rewrite.so
```

4. Configure the rewrite rules by adding the following lines to the end of the `httpd.conf` file:

```
RewriteEngine on
RewriteRule ^/lzxnet(.*)$ http://localhost:8080/lps-3.1.1$1 [p]
RewriteRule ^/lps-3\.1\.1(.*)$ http://localhost:8080/lps-3.1.1$1 [p]
```

The `RewriteEngine` command enables the rewrite rules. The first `RewriteRule` command proxies all requests for `localhost://lzxnet/*` to `localhost:8080//lps-3.1.1/*`. The second `RewriteRule` command proxies all requests for `localhost://lps-3.1.1/*` to `localhost:8080//lps-3.1.1/*`. These two rewrite rules are required to support OpenLaszlo development with Tomcat on port 8080, and they should be updated to reflect the version of LPS you have installed and the socket Tomcat is listening on.

5. To load and configure `mod_php`, add the following lines to the end of the `httpd.conf` file: `LoadModule php5_module "C:/php5/php5apache2.dll"`

```
AddType application/x-httpd-php .php
PHPIniDir "C:/php5"
```

6. The `LoadModule` command loads `mod_php`. The `AddType` command configures files ending with ".php" to be handled by `mod_php`. The `PHPIniDir` command tells `mod_php` where to find its configuration file. Save the file.
7. Restart Apache. Look for the Apache Service Monitor icon on the right side of the taskbar (it looks like a red feather). If it's not there, you can start it with the **Start menu > All Programs > Apache HTTP Server > Configure Apache Server > Monitor Apache Servers**. Open the Apache Monitor window and click **Restart** to restart Apache with the new configuration.

Now let's test the PHP integration.

Test the Apache installation

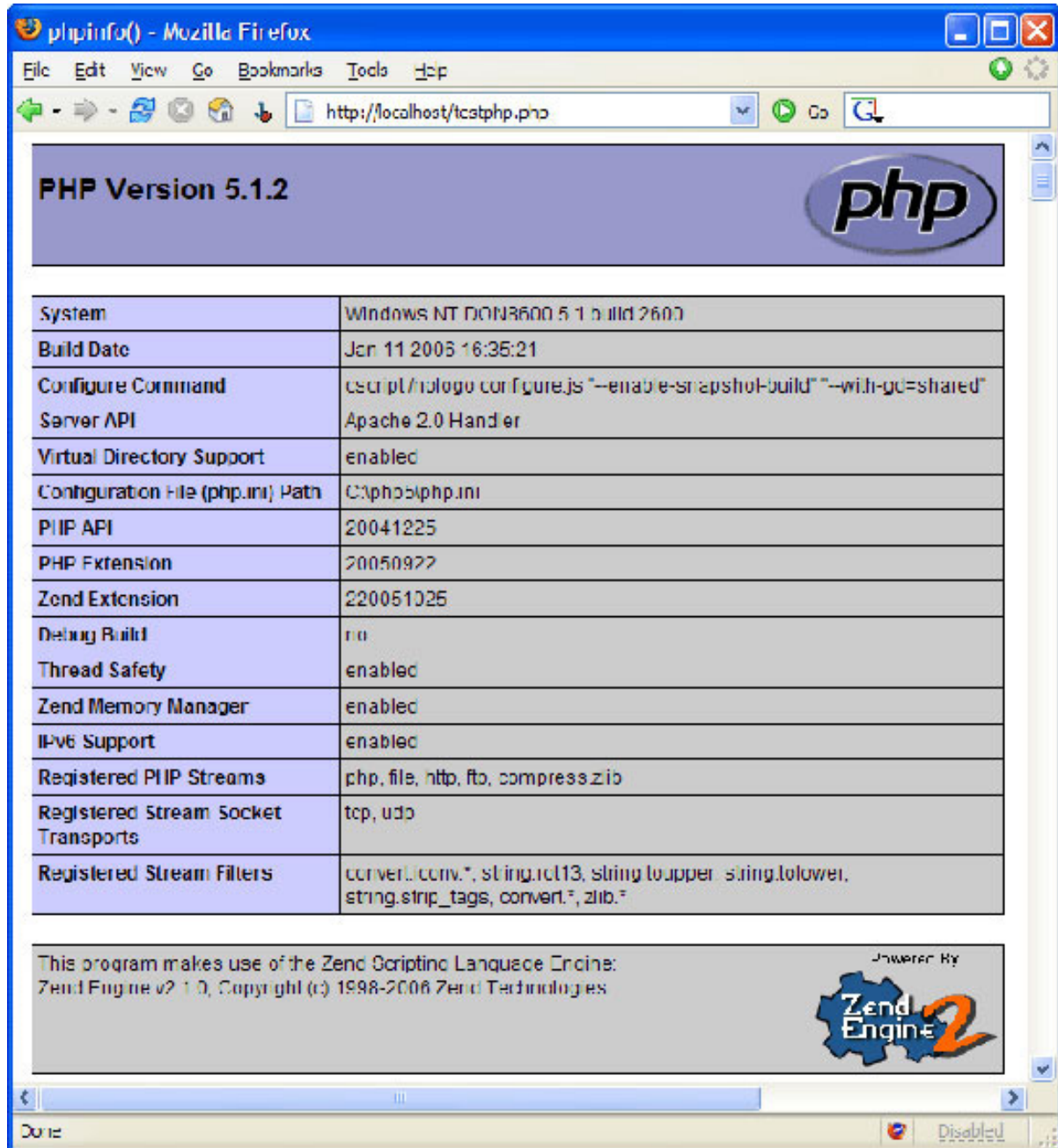
To test the actual Apache configuration, open your browser to `http://localhost`. Depending on how you have security configured and what files are present, you will see the Test Page for Apache Installation or a Forbidden 403 error. Don't panic because it means the server is running.

To test PHP, create a file, save it as `C:\public_html\testphp.php`, and add the following line:

```
<? print phpinfo(); ?>
```

Open the URL <http://localhost/testphp.php>. It should display the PHP information page, describing the PHP version and configuration.

Figure 1. PHP test page



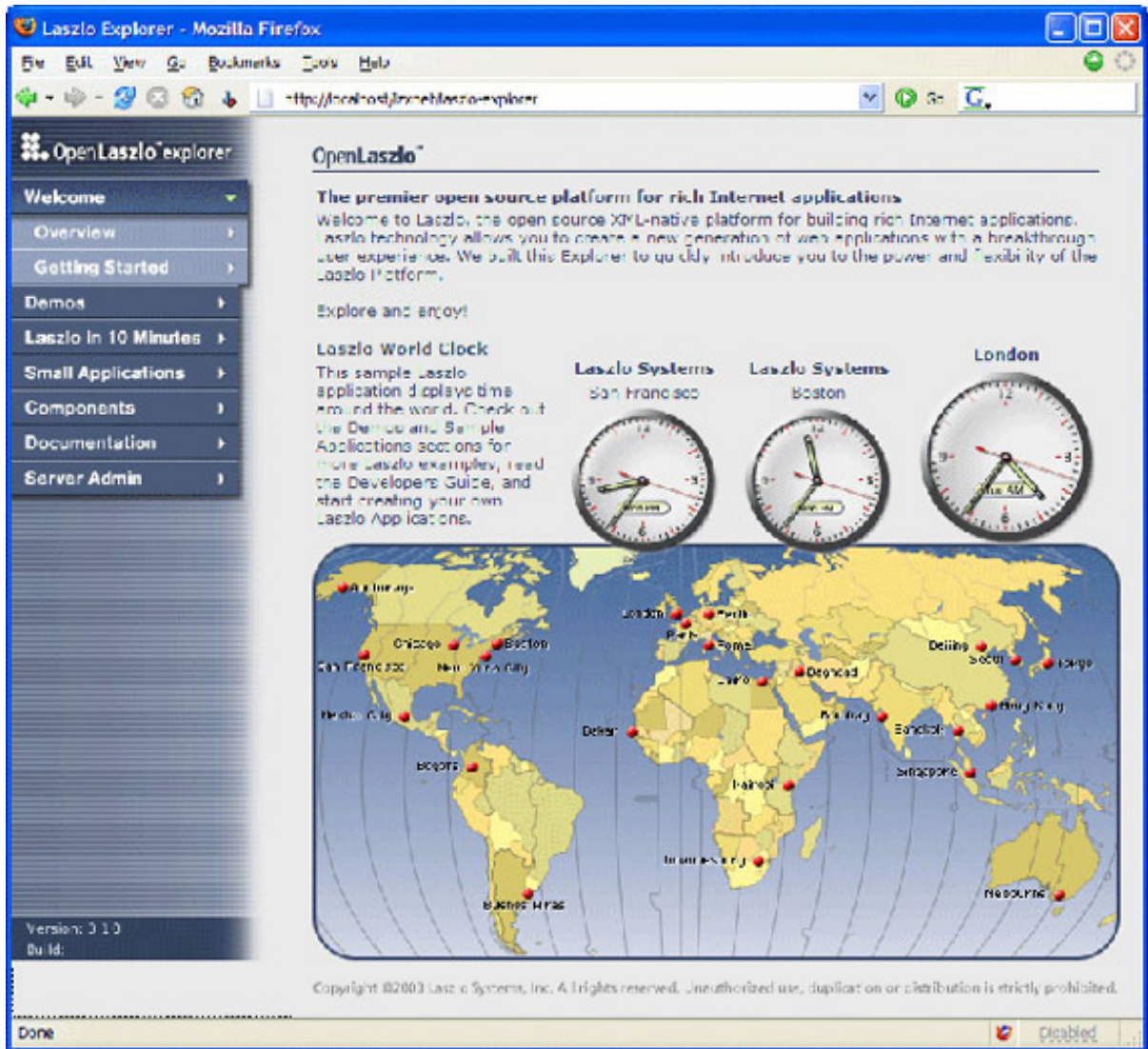
Installing OpenLaszlo

OpenLaszlo is available as a binary installer, as a source code archive, or from the Subversion repository. You will use the binary installer, which includes everything you need, including demos, documentation, examples, libraries, and the Tomcat Web server, which hosts the OpenLaszlo server.

Double-click the installer to start it and install OpenLaszlo into C:\openlaszlo. After the installer finishes, it will start the OpenLaszlo server, which you'll see in a command window. You can start and stop the OpenLaszlo server later with **Start menu > All Programs > OpenLaszlo Server > Start OpenLaszlo Server** and **Stop OpenLaszlo Server**.

The installer should also open your browser to a demo page. If it doesn't, point your browser to <http://localhost/lzxnet> to browse the index of Laszlo documentation and demos. The Laszlo Explorer is an interactive introduction that demonstrates what OpenLaszlo can do, including a hands-on tutorial called "Laszlo in 10 Minutes." You can run it by opening <http://localhost/lzxnet/laszlo-explorer>.

Figure 2. OpenLaszlo Explorer



Section 3. Overview

This section introduces the example application and explains how it will be implemented.

What will be accomplished

The goal here is to demonstrate a reasonably interesting application with a nontrivial protocol, no matter how corny. And there's nothing more corny than knock-knock

jokes (except for elephant jokes, but they're not as richly interactive). So you'll make a knock-knock joke server you can extend with your own favorites. Then you'll make a nice OpenLaszlo user interface with a wooden door background, animated brass knocker, sound effects, text fields, and buttons.

First, you'll implement a text-only server-side PHP HTML-based example; then you'll implement a rich client-side OpenLaszlo-based example. In Part 2, you'll see how to split up the work by implementing a distributed client-server OpenLaszlo/PHP example.

Figure 3. OpenLaszlo knock-knock application



The knock-knock protocol

Let's look at the logic you're going to implement with the PHP and OpenLaszlo versions of this application.

Knock-knock joke protocol for "Knock Knock Ha! Or Not!" example:

1. User clicks on the shiny brass knocker, and the client plays a knocking sound. In this HTML version, you'll simply display a form with text and a button, without any graphics or sounds, by opening `http://localhost/KnockKnockHTML.php?request=Start`.
2. Server responds with the message `<KnockKnock/>` (or an HTML form, in this first example).
3. Client displays the message "Knock Knock!" and a button asking "Who's There?"
4. User wonders who's there and presses "Who's There?" button to find out.
5. Client sends a "Who's There?" message to server, by requesting `http://localhost/KnockKnockHTML.php?request=WhosThere`.
6. Server computes its identity (using the "Deep Who" algorithm developed at IBM) and responds with the message `<WhosThere who="A little old lady/>`.
7. Client displays the message "A little old lady" and "A little old lady who?" button.
8. User contemplates the question, desires more details, then presses the "A little old lady who?" button to be enlightened.
9. Client sends a message `Who(who='A little old lady')` by requesting `http://localhost/KnockKnockHTML.php?request=Who&who=A+little+old+lady`.
10. Server ponders its true identity, based on the token provided in the `who` parameter, decides upon an appropriate answer, and responds with the message `<Who who="A little old lady">I didn't know you could yodel.</Who>`.
11. Client displays the answer "I didn't know you could yodel," a button "Ha!" and a button "Not."

12. User rolls on the floor laughing, then gleefully presses "Ha!"
13. Client sends a "Ha(who='A little old lady') " message to the server, by requesting `http://localhost/KnockKnockHTML.php?request=Ha&who=A+little+old+lady.`
14. Server records the user's emotional response and replies with the message `<ThankYou/>`.
15. Client accepts the server's sincere gratitude, resets the user interface, and loops back to the beginning.

Extra-credit homework

Of course, if you really want to make this a viable Web 2.0 application to secure venture capital, you'd need a few more gratuitous features, such as:

- Client side -- Add front-end client support for funny sliders, such as a "Forgot to Laugh" checkbox, cut-up and paste, laugh and drop, direct emotulation, applause and laughter signs, Laugh-O-Meter widget with microphone input, and keyboard-accessible WYSIWYG laugh editor for the humor-impaired.
- Server side -- Implement back-end database support for laugh-tracking, personalized tastes, public ha-ha lists, community joke reviews, P2P joke sharing, setup tagging, punch-line folksonomies, targeted joke placement, adaptive humor, genetic timing algorithms, artificial stupidity, virtual irony, funny business logic, streaming sarcasm, and other advanced joke selection and rendering algorithms.

Section 4. What is OpenLaszlo?

OpenLaszlo is an open source platform for developing user-friendly Web-based applications that work identically across all popular browsers and platforms (Windows, Mac, Linux®, IE, Firefox, Safari, etc.). It supports a rich graphics model with scalable vectors, bitmaps, movies, animation, transparency, fonts, audio, streaming media, reusable components, user-interface widgets, control panels, property sheets, keyboard navigation, browser Back-button navigation, as well as advanced WYSIWYG text and graphical editing tools. In other words, OpenLaszlo is the velvet glove for the iron fist of PHP.

It does all this by taking an XML and JavaScript-based application, and compiling it into a form that can be readily understood by the browser. At the time of this writing, that form is a Flash movie -- without the need for the actual Flash application.

OpenLaszlo has been used for such applications as LaszloMail (a full-service e-mail client deployed by Earthlink), the Pandora music discovery service (an interface for listening to your personalized Internet radio stations), and reusable components, such as Pie menus (directional menu selection of pie slice-shaped targets). See [Resources](#) for more information about these applications.

Running OpenLaszlo applications

OpenLaszlo is designed to use open standards, and is built with open source tools and technologies. It's a high-level architecture designed to target multiple rendering environments, and the first one it currently supports is Flash. The Flash player itself is not open source, but it's freely available, often preinstalled, and widely deployed. Its advantage over DHTML is that it's perfectly consistent across all platforms and supports many advanced multimedia features seamlessly.

Flash has some serious limitations, such as the lack of a JavaScript source code compiler or interpreter, no support for XML standards like XPath, and cross-domain issues. However, PHP, and the OpenLaszlo runtime and server address many of these problems.

Looking forward, OpenLaszlo's platform-independent architecture means that in the future, it can target other runtime environments, such as DHTML, Java, Mozilla XUL, Avalon XAML, SVG, open source Flash players, cell phones, and other embedded devices, as they become more powerful and mature. The OpenLaszlo Platform Roadmap (see [Resources](#)) describes the development goals and progress toward the next major release, which will include support for multiple runtimes and many architectural improvements.

OpenLaszlo concepts and server features

OpenLaszlo application source code is written in the LZX language, which is XML with embedded JavaScript code. Laszlo programs can also include reusable component libraries and resources in various formats, such as audio, images, SWF vector graphics, animations, movies, fonts, rich text, and XML data. The OpenLaszlo server provides a compiler that translates text LZX programs, libraries, and binary resources into binary SWF files the Flash player can execute directly.

The OpenLaszlo server has many features, including the abilities to:

- Dynamically compile, debug, and optimize OpenLaszlo programs
- Act as a proxy for OpenLaszlo applications, so the Flash player can communicate with other servers
- Perform XML-RPC and SOAP calls
- Send and receive messages over persistent connections with other clients
- Transform XML into an efficient binary representation and transcode various file formats into SWF files that the Flash player can digest directly

SOLO mode

Networked OpenLaszlo applications can take advantage of the OpenLaszlo server by running in *proxied mode*. You can also develop applications that run in *Standalone OpenLaszlo Output (SOLO)* mode independently. SOLO applications talk to your PHP server directly or to other XML Web services such as Flickr, Google Maps, etc. Self-contained OpenLaszlo programs can run without any network server at all. They can even be deployed as desktop applications.

XML data

OpenLaszlo uses XML data for structuring programs, reusable component libraries, class definitions, resources, declaring JavaScript classes, methods and attributes, creating and configuring objects, constraints and data bindings. XML data sets can be precompiled efficiently into OpenLaszlo programs or asynchronously downloaded from Web services, then bound to JavaScript objects automatically.

JavaScript code

OpenLaszlo uses JavaScript code for defining scripts and method bodies, event handlers, attribute expressions (automatically updated constraints), as well as representing objects, XML structures and graphical views as JavaScript dictionaries. Laszlo represents and manipulates XML as JavaScript object dictionaries and has a sophisticated asynchronous XML data binding system that uses two-way constraints to tie the properties of JavaScript objects together with XML data, and drive the creation and replication of intelligent data-bound Laszlo views.

Declarative programming

Declarative programming is an elegant way of writing code that concisely describes

what to do, instead of how to do it. OpenLaszlo supports declarative programming in many ways: using XML to declare JavaScript classes, create object instances, configure them with automatic constraints, and bind them to XML data sets. Declarative programming dovetails and synergizes with other important OpenLaszlo techniques, including objects, prototypes, events, constraints, data binding, and instance-first development.

Constraints

Constraints in OpenLaszlo are object attributes whose values are defined as functions of other object or XML data attributes. They're used to configure the relationships between objects, their attributes, and XML data. Like a spreadsheet, Laszlo conveniently keeps track of all dependencies and automatically updates constraints when the values they depend on change. Constraints can refer to object attributes with JavaScript expressions and XML data with XPath expressions.

Automatic constraints are wonderful, for the same reason that automatic garbage collection is great. They save a huge amount of tedious error-prone effort and repetitive coding by automatically and reliably performing intricate housekeeping tasks. They also make the code much more readable and concise by letting you express what to do, instead of how to do it (the essence of declarative programming).

XML data binding

XML data bindings are XPath-based constraints that create two-way connections between object attributes and XML data. When an XML data-bound attribute is changed, the XML data and all constraints that depend on it are updated, so it's easy to implement automatic updating of multiple XML data views and editors.

XML data bindings can also cause the replication of exemplary Laszlo objects (prototypes), so you can declare how one example object will look in terms of XML elements and attributes, and it will replicate a separate instance automatically for each XML element in the data set.

Section 5. The PHP application

Overall, you are talking about an OpenLaszlo interface for the PHP application, so let's look at the general logic of the PHP application.

The knock-knock jokes

Let's start by looking at the structure of the data we're working with. The data is contained in the KnockKnockJokes.xml file, with each joke in its own KnockKnockJokes element (see Listing 2).

Listing 2. The KnockKnockJokes.xml file

```
<?xml version="1.0"?>
<KnockKnockJokes>
  <KnockKnockJoke who="You" Not="3" Ha="16">
    You who, is anybody in?
  </KnockKnockJoke>
  <KnockKnockJoke who="A little old lady"
Not="4" Ha="12">
    I didn't know you could yodel.
  </KnockKnockJoke>
  ...
</KnockKnockJokes>
```

You can download the full list of jokes -- such as it is -- (see [Resources](#)).

The `who` attribute acts as an identifier for the joke and as data, with the punch line as the text content of the element. The `Not` and `Ha` attributes store counts of user reactions.

Starting

Ultimately, you'll build an OpenLaszlo application that displays information for a number of states. To get a handle on that, you'll first build a simple HTML-based PHP application that models the knock-knock protocol. Start by creating the basic page and saving it, as shown in Listing 3.

Listing 3. The basic knockknock.php file

```
<html>
<head><title>Knock Knock Ha! or Not!</title></head>
<body>

<?php

if (isset($_GET["request"])) {
    $request = $_GET["request"];
} else {
    $request = "none";
}

switch ($request){
    default: {
        echo "<h1><a href=\"KnockKnockHTML.php?request=start\">Start</a></h1>";
```

```

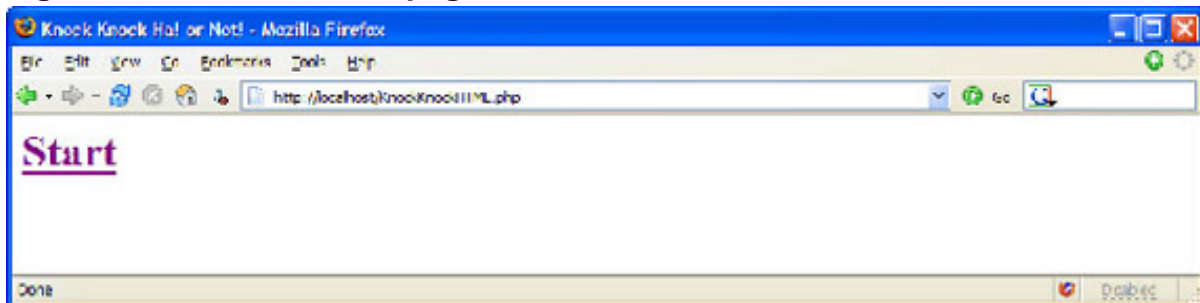
        break;
    }

}
?>
</body>
</html>

```

This is pretty straightforward, with the state represented by the `request` parameter and managed by a `switch` statement. If the parameter isn't present, you simply display a link that brings you to the start state. If you point your browser to `http://localhost/KnockKnockHTML.php`, you should see an image like Figure 4.

Figure 4. The default start page



Knocking

Now let's add the actual knocking, as shown in Listing 4.

Listing 4. Add the knocking

```

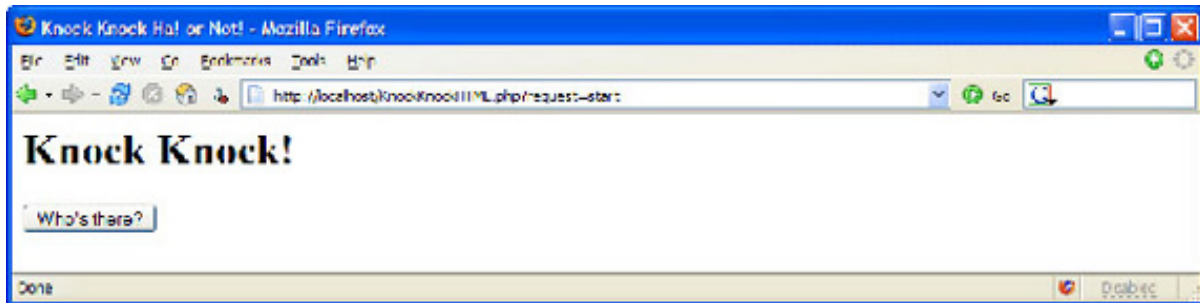
...
switch ($request){
    case "start" : {
        echo "<h1>Knock Knock!</h1>";
        echo "<form action=\"KnockKnockHTML.php\" method=\"get\">";
        echo "<input type=\"hidden\" name=\"request\" \
value=\"WhosThere\" />";
        echo "<input type=\"submit\" value=\"Who's there?\" />";
        echo "</form>";

        break;
    }
    default: {
        ...
    }
}

```

You're in the start state, so you'll display a form that gives the user a "Who's there?" button, as shown in Figure 5.

Figure 5. Adding the "Knock Knock!" form



Clicking the "Who's there?" button puts you in the `WhosThere` state, shown in Listing 5.

Listing 5. The WhosThere state

```

...
    break;
}

case "WhosThere" : {

    $whosThere = "Little old lady";

    echo "<h1>" . $whosThere . "</h1>";
    echo "<form action=\"KnockKnockHTML.php\" method=\"get\">";
    echo "    <input type=\"hidden\" name=\"request\" value=\"Who\" />";
    echo "    <input type=\"hidden\" name=\"who\" \
value=\"\".$whosThere.\"\" />";
    echo "    <input type=\"submit\" value=\"\".$whosThere.\" who?\" />";
    echo "</form>";

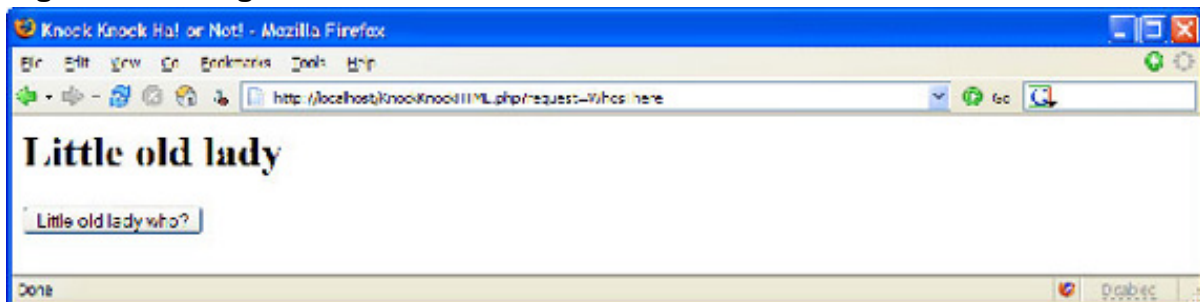
    break;
}

default: {
...

```

In this case, the response has been hard-coded. You'll pull it from the XML file in a moment, but the important thing is that in this state, you create a form that moves to the new state (`who`) and passes the ID for the joke, as shown in Figure 6.

Figure 6. Asking the server "Who's There?"



Laughing (or not)

Once the user gets the punch line, you provide an opportunity to provide feedback, as shown in Listing 6.

Listing 6. Providing the punch line

```

...
    break;
}

case "Who" : {

    $who = $_GET["who"];
    $whoItReallyIs = "I didn't know you could yodel.";

    echo "<h1>" . $whoItReallyIs . "</h1>";
    echo "<form action=\"KnockKnockHTML.php\" method=\"get\">";
    echo "    <input type=\"hidden\" name=\"request\" value=\"HaOrNot\" />";
    echo "    <input type=\"hidden\" name=\"who\" value=\"".$who."\" />";
    echo "    <input type=\"submit\" name=\"isItFunny\" value=\"Ha!\" />";
    echo "    <input type=\"submit\" name=\"isItFunny\" value=\"Not!\" />";
    echo "</form>";

    break;
}

case "HaOrNot" : {

    $who = $_GET["who"];
    $isItFunny = $_GET["isItFunny"];

    echo "<h1>Thank you!</h1>";
    echo "<a href=\"KnockKnockHTML.php?request=start\">Start</a>";

    break;
}

default: {
...

```

Again, the response has been hard-coded for the moment. Also, at this point, you're not actually doing anything with the feedback, but you are collecting it, as shown in Figure 7.

Figure 7. Viewing the punch line



Adding the knock-knock library

In future parts of this "[Spice up PHP applications with OpenLaszlo](#)" series, you'll create PHP that accesses the database of jokes (or in this case, the XML file), and you can see the complete KnockKnockLib.php file in the source code for this tutorial (see [Resources](#)), but for now, just understand that you're providing several functions, as shown in Listing 7.

Listing 7. The final KnockKnockHTML.php page

```
<html>
<head><title>Knock Knock Ha! or Not!</title></head>
<body>

<?php
require("KnockKnockLib.php");
$jokeDoc = getJokeDocument();
...
switch ($request){
...
    case "WhosThere" : {
        $whosThere = getWhosThere($jokeDoc);
        echo "<h1>" . $whosThere . "</h1>";
    ...
    }
    case "Who" : {
        $who = $_GET["who"];
        $whoItReallyIs = getWho($jokeDoc, $who);
        echo "<h1>" . $whoItReallyIs . "</h1>";
    ...
    }
    case "HaOrNot" : {
        $who = $_GET["who"];
        $isItFunny = $_GET["isItFunny"];
        if ($isItFunny == "Ha!"){
            $haOrNot = "Ha";
        } else {
            $haOrNot = "Not";
        }
        saveHaOrNot($jokeDoc, $who, $haOrNot);

        echo "<h1>Thank you!</h1>";
        echo "<a href=\"KnockKnockHTML.php?request=start\">Start</a>";

        break;
    }
    default: {
    ...

```

Now that you have a working prototype in PHP, let's look at OpenLaszlo.

Section 6. Your first OpenLaszlo application

In this section, you will create a simple Hello World application. This will consist of a text field to prompt the user, a button to give the user something to click, and a text field to display a feedback message.

The Hello World application

Create a new file in C:\openlaszlo\Server\lps-3.1.1\my-apps\helloworld.lzx. Copy and paste the text from Listing 8 into the file.

Listing 8. The Hello World application

```
<canvas width="100%" height="100%">
  <simplelayout axis="y"/>
  <text resize="true">
    Press this button to say hello to the
    world:
  </text>
  <button
onclick="responseText.setAttribute('visible',
true)">
    Hello World
  </button>
</canvas>
```

Run this program by pointing your browser to <http://localhost/lzxnet/my-apps/helloworld.lzx>.

The outer `<canvas>` element is the top level of the Laszlo application. It sets the width and height to 100 percent, so the application takes up the entire area of the browser. The canvas is a special view that serves as the root of the "view hierarchy." It can contain visible (or hidden) subviews (like text, buttons, windows, and other widgets), as well as invisible helper objects (like `<simplelayout>`).

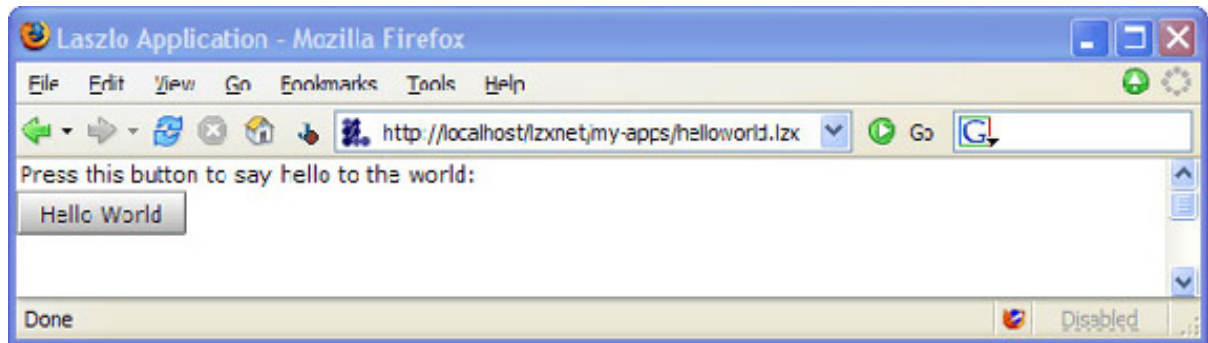
The `<simplelayout>` element is a helper object that manages the layout of the visible elements at the same level of the view hierarchy (the text and button elements in this example). The attribute `axis="y"` means to arrange the views vertically along the y axis, from top to bottom. Without the `<simplelayout>` element, the text and the buttons would all overlap in the upper-left corner.

The first `<text>` element is a text label view, and the content "Press..." provides the

text to display. The `resize="true"` attribute resizes the view to fit the width of the text. (Otherwise, it might be clipped or take up too much space).

The `<button>` element is a button view, and the content Hello World provides the label. The response should look something like Figure 8.

Figure 8. The Hello World application



Adding interactivity

That's all very nice, but it would be great if you could get it to actually do something. Let's add a little interactivity, as shown in Listing 9.

Listing 9. Adding interactivity

```
<canvas width="100%" height="100%">
  <simplelayout axis="y"/>
  <text resize="true">
    Press this button to say hello to the world:
  </text>
  <button
    onclick="responseText.setAttribute('visible', true)">
    Hello World
  </button>
  <text id="responseText"
    resize="true"
    visible="false">
    The world says hello to you too.
  </text>
</canvas>
```

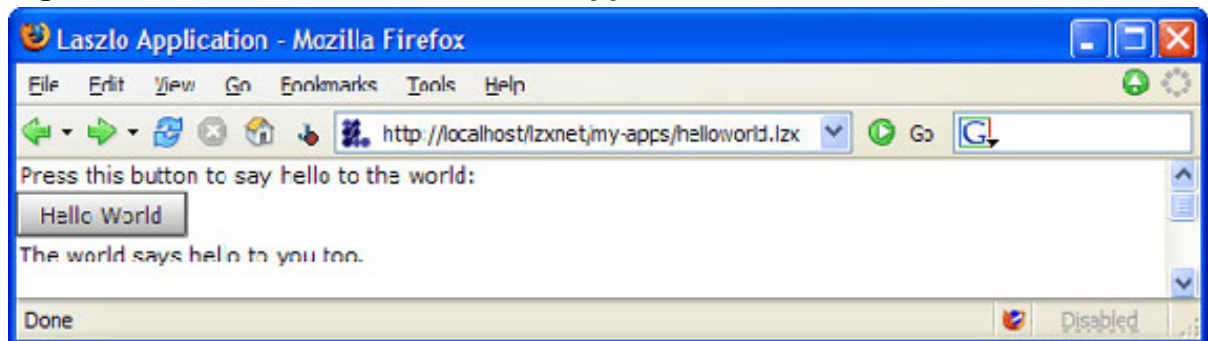
The attribute `onclick="responseText.setAttribute('visible', true)"` is the JavaScript click event handler that gets called when the button is selected. It sets the visibility of the `responseText` view to `true` when the user clicks the button.

The second `<text>` element is a text label view that starts out hidden (notice the `visible="false"` attribute), but is made visible when the user clicks the button.

The XML element has an attribute `id="responseText"` -- which gives it a global name, so other objects and JavaScript code can easily refer to it. Each ID in a program must be unique, of course.

Note that the button `onclick` handler calls `responseText.setAttribute('visible', true)` to change the value, rather than `responseText.visible = true` because `setAttribute` tells the constraint system to call any handlers interested in the changing value, which causes all expressions depending on the attribute to be recalculated and updates the display. If you just set the value without calling `setAttribute`, the magic stuff won't happen, and the text won't appear.

Figure 9. The interactive Hello World application



The debugger

A discussion of the OpenLaszlo debugger is outside the scope of this tutorial, but to interactively explore and debug the application, run it with debugging enabled, by opening `http://localhost/lzxnet/my-apps/helloworld.lzx?debug=true`.

That will load the application and open a Laszlo Debugger window, with an input text field at the bottom. The interactive debugger (shown in Figure 10) is a powerful -- and fun -- tool for testing and debugging OpenLaszlo programs.

Figure 10. The interactive debugger

set of interface elements. Also, you'll be able to add more decorative elements, such as the animated door knocker and sounds.

Creating a simple view with a resource

Start by creating the actual application. Create a directory called `C:\openlaszlo\Server\lps-3.1.1\my-apps\KnockKnockStandAlone`. In that directory, create new file called `KnockKnockStandAlone.lzx`, and add the following from Listing 10.

Listing 10. The initial application

```
<canvas
  width="100%"
  height="100%"
  proxied="false"
>

  <!-- KnockKnockStandAlone.lzx
       Knock-Knock Stand-Alone OpenLaszlo
       Application.
       By Don Hopkins. -->

  <resource name="door_picture"
  src="images/WoodDoor.png" />

  <view name="door"
        resource="door_picture"
  />

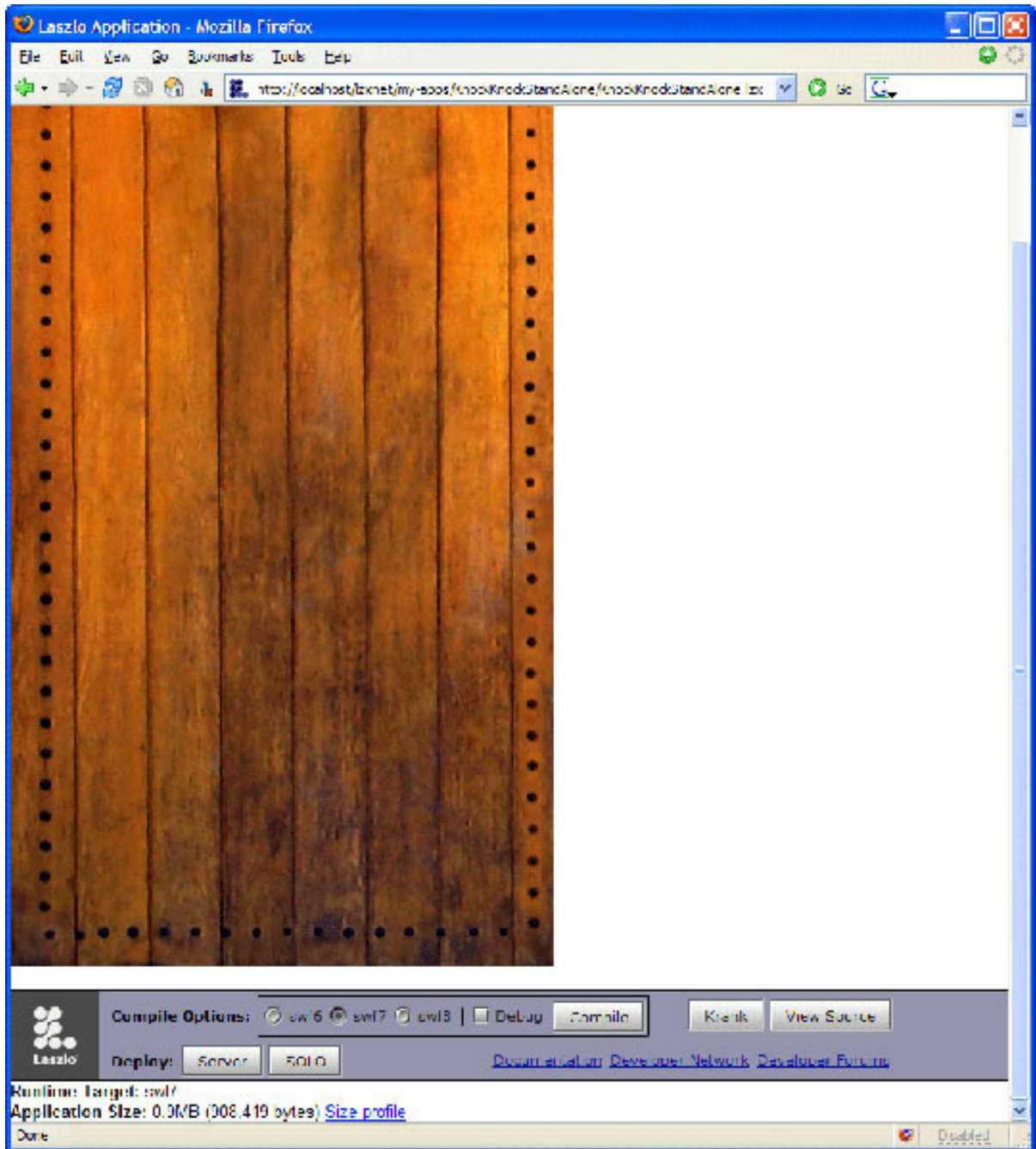
</canvas>
```

Once again, you start with a simple canvas, but in this case, you're specifying a resource, `door_picture`, which refers to a specific file. (You can find this file with the source code for this tutorial in [Download](#).)

Now make a directory called `C:\openlaszlo\Server\lps-3.1.1\my-apps\KnockKnockStandAlone\images` and put the `WoodDoor.png` file into the images directory. The door view refers to the `door_picture` resource, which loads the file `images/WoodDoor.png`.

If you open your browser and point it to `http://localhost/lzxnet/my-apps/KnockKnockStandAlone/KnockKnockStandAlone.lzx`, you should see the door graphic, as shown in Figure 11.

Figure 11. The door graphic (scrolled down to view the OpenLaszlo compiler options)



Add an animated resource

You want the door knocker to be animated: When the user clicks, it should appear to come out and knock on the door with a knocking noise. You can create the actual animation information right in the resource, as shown in Listing 11.

Listing 11. Adding an animated resource

```

...
    Knock-Knock Stand-Alone OpenLaszlo Application.
    By Don Hopkins. -->

    <resource name="door_picture" src="images/WoodDoor.png"/>

    <resource name="knocker_picture">
      <frame src="images/KnockDown.png"/>
      <frame src="images/KnockUp.png"/>
    </resource>

    <view name="door"
      resource="door_picture"
    />

    <view name="knocker"
      x="140" y="80"
      resource="knocker_picture"
      onmousedown="this.knockUp()"
      onmouseup="this.knockDown()"
    >

      <method name="knockUp">
        <![CDATA[

          this.setAttribute("frame", 2);

        ]]>
      </method>

      <method name="knockDown">
        <![CDATA[

          this.setAttribute("frame", 1);

        ]]>
      </method>

    </view>

</canvas>

```

Starting at the top, you are creating the `knocker_picture` resource with two frames, one in the normal `Down` state hanging against the door and the other in the `Up` state when the user presses the mouse button. These are, respectively, frames 1 and 2.

Now create a view called "knocker," which puts the picture on the page. That view has two methods: `knockUp()` and `knockDown()`, shown in the body of the view element. (Notice that the code is represented as `CDATA` to prevent problems quoting special XML characters like `&` and `<`.) These methods simply set the frame attribute for the view, which causes the resource in the view -- in this case, `knocker_picture` -- to display the image file associated with that frame. The methods themselves are called for the `onmousedown` and `onmouseup` events, as specified by the attributes on the view element.

If you reload the application in your browser, you should see the knocker, and you

should be able to see the animation when you click on it, as shown in figures 12 and 13.

Figure 12. Adding the knocker

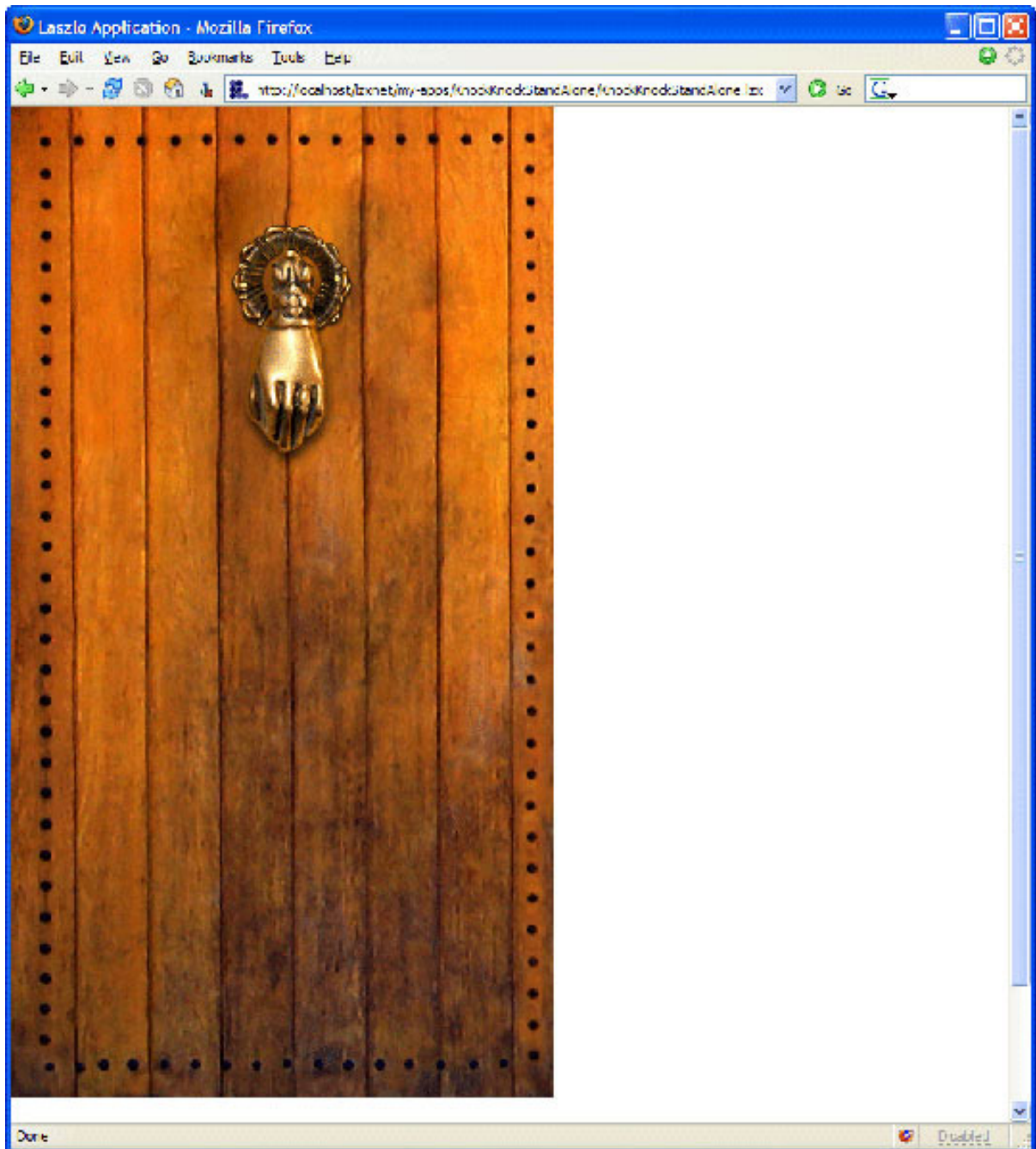
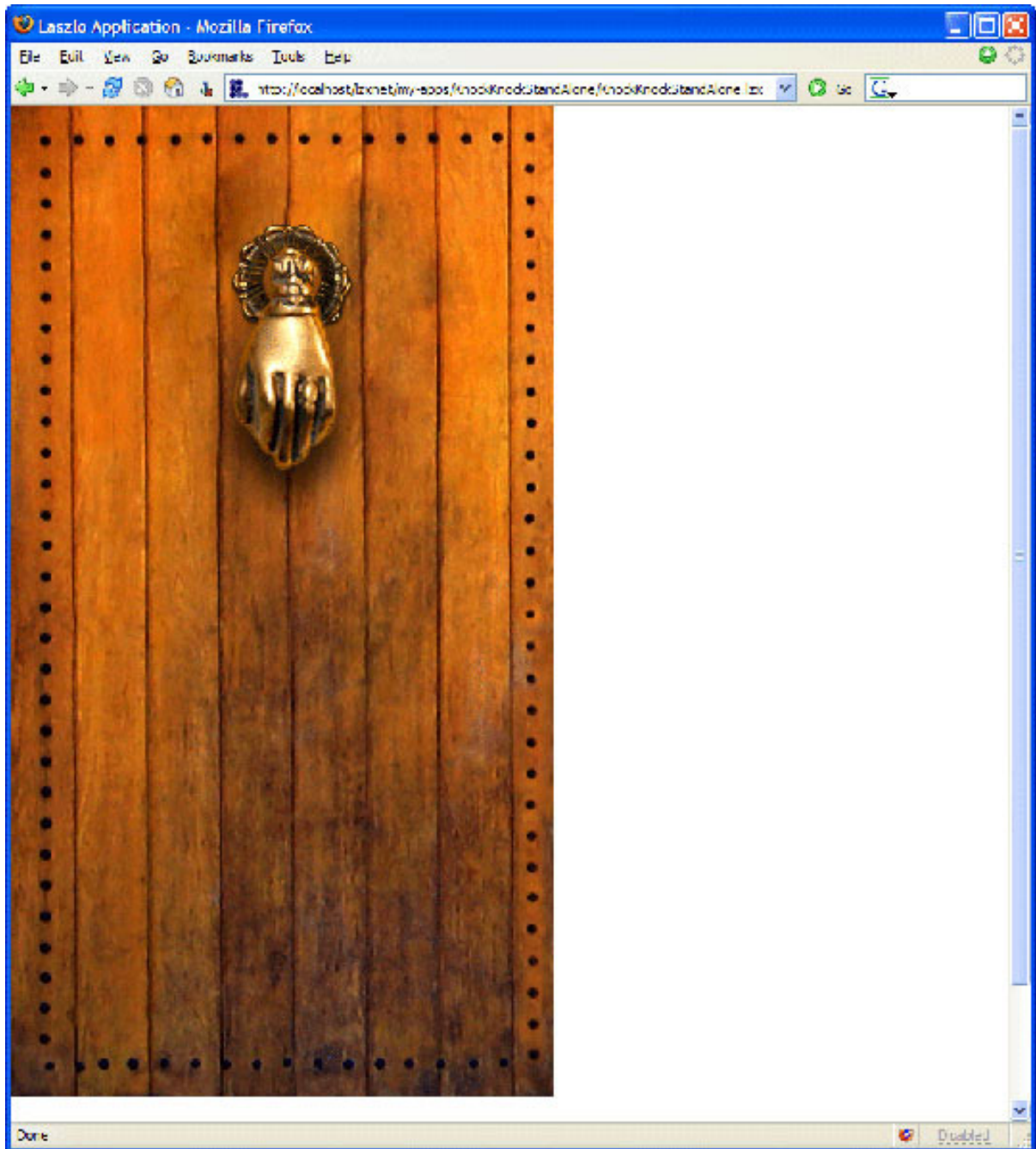


Figure 13. The knocker knocking



Creating classes

One nice OpenLaszlo feature is the ability to create and extend reusable components by defining classes based on prototypes of other objects. For example, you created a text label and a button in your Hello World application, and you can create classes that extend these elements by adding their own attributes, as shown in Listing 12.

Listing 12. Creating classes

```
...
  <frame src="images/KnockUp.png" />
</resource>

<class name="bigtext"
  extends="text"
  fgcolor="0xffffffff"
  multiline="true"
  fontsize="24"
  fontstyle="bold"
  width="{parent.width}"
/>

<class name="bigbutton"
  extends="button"
  fontsize="24"
  fontstyle="bold"
  width="{parent.width}"
/>

<view name="door"
...

```

Each of these new classes declares its name (like `name="bigtext"`), the name of the class it extends (like `extends="text"`), and adds attributes you might expect (like `fontstyle="bold"`). You can make instances of any class by writing an XML element of the same name, with attributes that define its properties (like `<bigtext text="Easy!" />`). By defining classes, you can create your own special-purpose XML vocabulary.

Let's look at the `width` attribute. This attribute is actually a computed value -- or rather, a constraint. In each case, you are telling OpenLaszlo that the object should be the same width as its parent, which could be a window, view, or the canvas itself. The handy thing is that when the parent is resized -- for example, if the user resizes the window -- the width of the object will change accordingly. OpenLaszlo automatically updates constraints, without you having to write any more code.

You'll also see this notation later when calculating other values.

Views and subviews

Now let's add these objects to the page, as shown in Listing 13.

Listing 13. Adding the objects to a subview

```
...
    this.setAttribute("frame", 1);
  <>
</method>

```

```
</view>

<view name="dialog"
  x="60"
  y="300"
  width="325"
>

  <view
    width="{parent.width}"
  >

    <simplelayout axis="y" spacing="5"/>

    <bigtext
      text="Knock Knock!"
    />

    <bigbutton
      text="Who's There?"
    />

  </view>

  <view
    width="{parent.width}"
  >

    <simplelayout axis="y" spacing="5"/>

    <bigtext
      width="{parent.width}"
      text="Little old lady"
    />

    <bigbutton
      text="Little old lady?"
    />

  </view>

  <view
    width="{parent.width}"
  >

    <simplelayout axis="y" spacing="5"/>

    <bigtext
      text="I didn't know you could yodel."
    />

    <bigbutton
      text="Ha!"
    />

    <bigbutton
      text="Not."
    />

  </view>

</view>

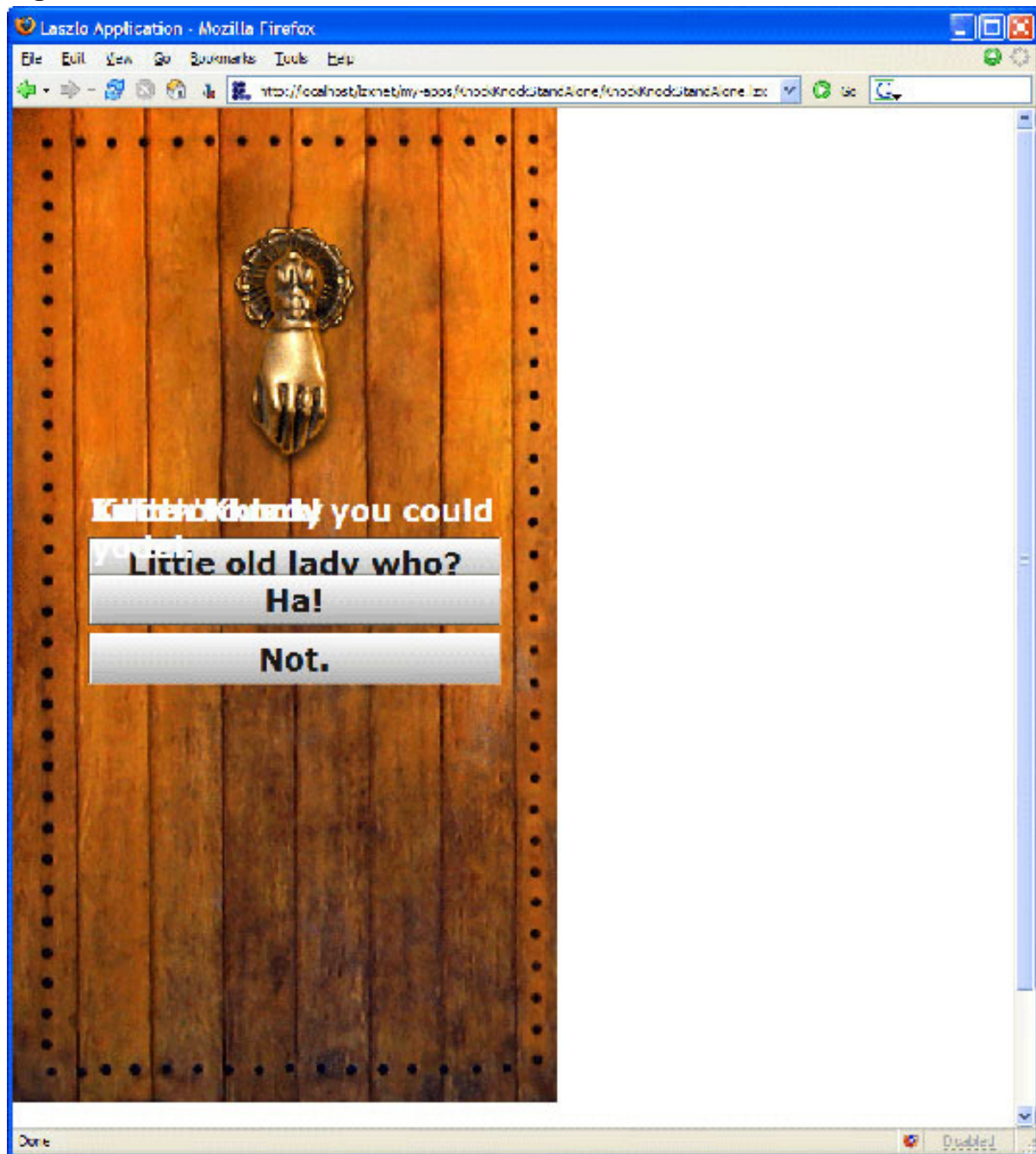
</canvas>
```

Notice that you have added several instances of the `bigtext` and `bigbutton` classes, each with its own value for the `text` attribute. As before, you'll hard-code

the values and replace them with live data later. All of the other attributes specified on the class will still come into play. (You could, however, override them for an individual instance by just specifying the attribute on the instance.)

Then notice that you actually have three views in the dialog view. These are called *subviews*, and each corresponds to a particular state of the application. As it stands now, they'll all be displayed at the same time, as you can see in Figure 14.

Figure 14. All of the subviews at the same time



This is obviously not what you want.

Tracking state information

You can dynamically show and hide views by defining their `visible` attribute to be a constraint whose value is `true` or `false`. To control when the each view becomes visible, you need to track the state of the application, which you will represent as a string, whose initial value is `Start` (see Listing 14).

Listing 14. Tracking state

```
...   width="{parent.width}"
    />

    <attribute name="state" type="string" value="Start"/>

    <view name="door"
...   <view name="dialog"
        x="60"
        y="300"
        width="325"
    >

        <view
            visible="{canvas.state == 'KnockKnock'}"
            width="{parent.width}"
        >
...   </view>

        <view
            visible="{canvas.state == 'WhosThere'}"
            width="{parent.width}"
        >
...   </view>

        <view
            visible="{canvas.state == 'Who'}"
            width="{parent.width}"
        >
...   </view>

    </view>

</canvas>
```

You've created an attribute on the canvas called `state` whose type is `string` (you could name it anything). The constraint expressions set the value of each view's `visible` attribute based on whether the state (accessed as `canvas.state`) is equal to a particular value. If it's not, the expression evaluates to `false`, and the view will not be visible. If it's `true`, then you see the view.

Now you just have to find a way to control the state value.

Defining methods to change state

To change state, you can create methods for changing the `state` attribute as part of the canvas itself, then you can call those methods from the buttons' `onclick` handlers, as shown in Listing 15.

Listing 15. Canvas methods

```
...
<attribute name="state" type="string" value="Start"/>
<method name="doStart"><![CDATA[
    canvas.setAttribute('state', 'Start');
]]>
</method>
<method name="doKnockKnock"><![CDATA[
    canvas.setAttribute('state', 'KnockKnock');
]]>
</method>
<method name="doWhosThere"><![CDATA[
    canvas.setAttribute('state', 'WhosThere');
]]>
</method>
<method name="doWho"><![CDATA[
    canvas.setAttribute('state', 'Who');
]]>
</method>
<method name="doHa"><![CDATA[
    canvas.setAttribute('state', 'Start');
]]>
</method>
<method name="doNot"><![CDATA[
    canvas.setAttribute('state', 'Start');
]]>
</method>

<view name="door"
...
<view name="dialog"
  x="60"
  y="300"
  width="325"
>
```

```
<view
  visible="{canvas.state == 'KnockKnock'}"
  width="{parent.width}"
>
...
  <bigbutton
    onclick="canvas.doWhosThere()"
    text="Who's There?"
  />
</view>

<view
  visible="{canvas.state == 'WhosThere'}"
  width="{parent.width}"
>
...
  <bigbutton
    onclick="canvas.doWho(canvas.who)"
    text="Dishes who?"
  />
</view>

<view
  visible="{canvas.state == 'Who'}"
  width="{parent.width}"
>
...
  <bigbutton
    onclick="canvas.doHa()"
    text="Ha!"
  />

  <bigbutton
    onclick="canvas.doNot()"
    text="Not."
  />
</view>
</view>
</canvas>
```

The methods themselves are pretty simple, and all they do is set the attribute value by calling `setAttribute`. OpenLaszlo calls the methods when the user clicks on the appropriate button. Then the state changes, and the appropriate view is made visible, as you can see in figures 15 and 16.

Figure 15. KnockKnock state

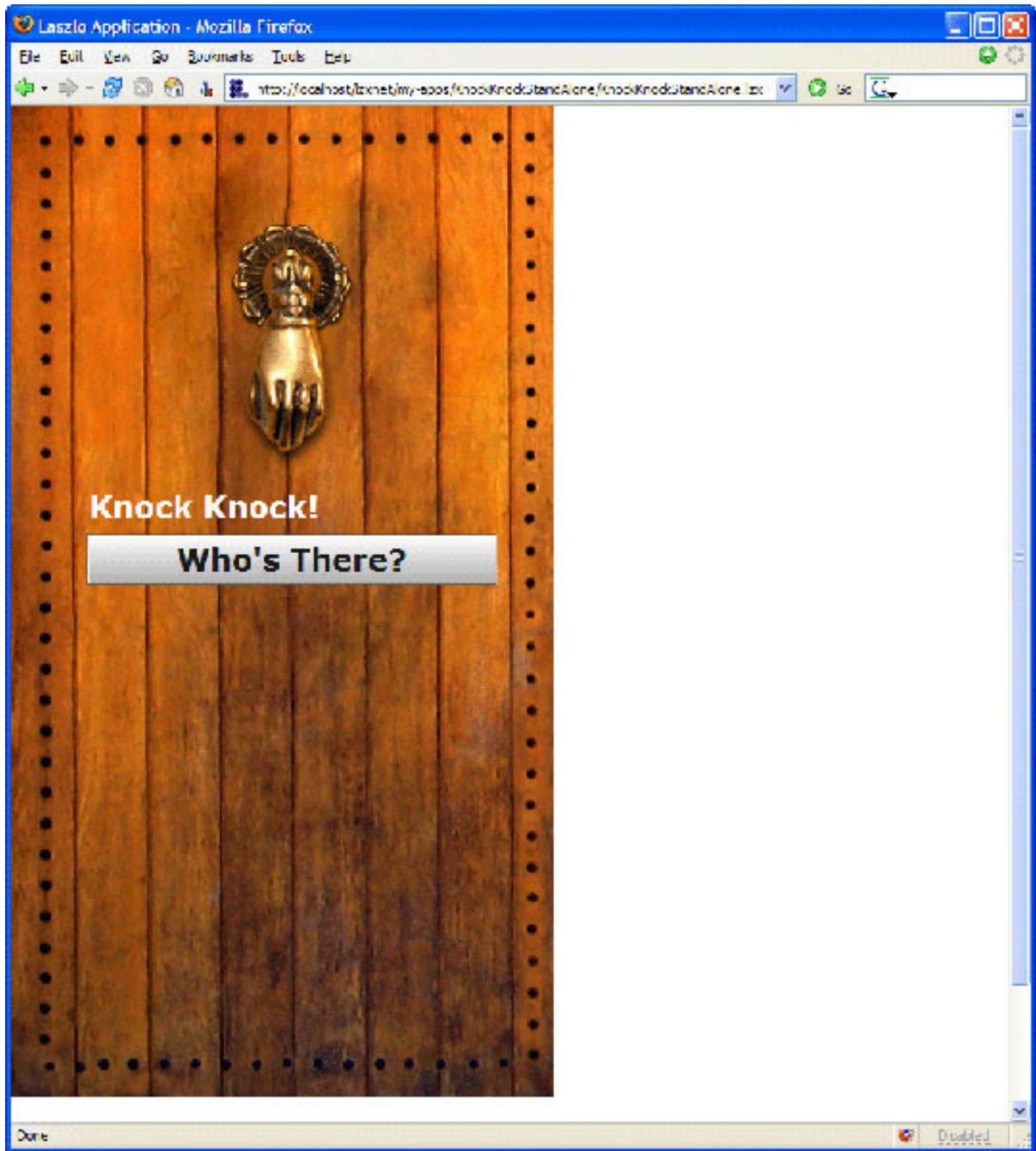
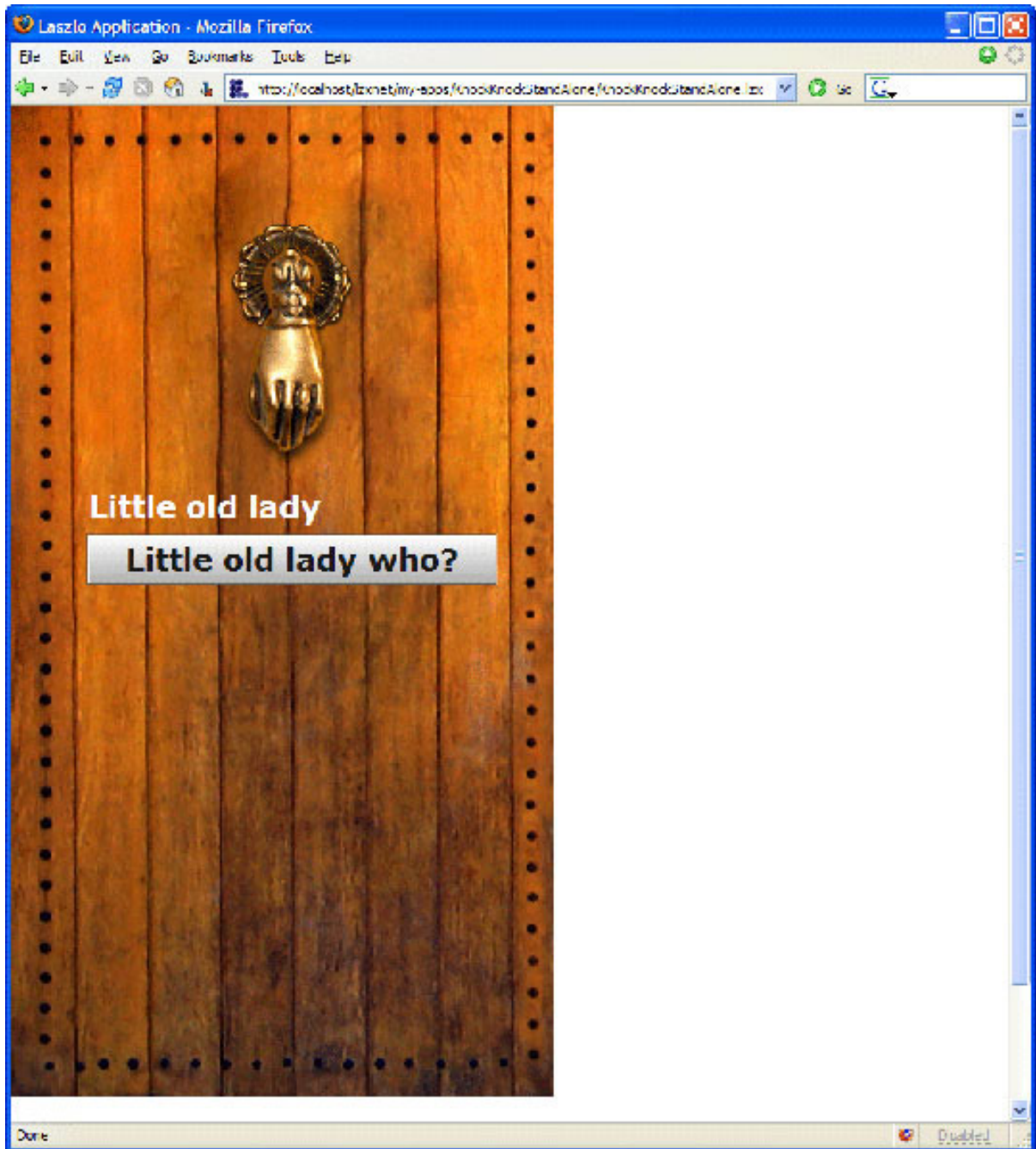


Figure 16. WhosThere state



Adding sound

In terms of the presentation, all you have left is to add the sounds, as shown in Listing 16.

Listing 16. Adding sound

```
...
<resource name="door_picture" src="images/WoodDoor.png"/>
<resource name="knock_sound" src="sounds/Knock.mp3"/>
<resource name="ha_sound" src="sounds/Ha.mp3"/>
<resource name="not_sound" src="sounds/Not.mp3"/>

<resource name="knocker_picture">
...

<method name="doKnockKnock"><![CDATA[

    LzAudio.playSound("knock_sound");
    canvas.setAttribute('state', 'KnockKnock');

]]>
</method>
...
<method name="doHa"><![CDATA[

    LzAudio.playSound('ha_sound');
    canvas.setAttribute('state', 'Start');

]]>
</method>

<method name="doNot"><![CDATA[

    LzAudio.playSound('not_sound');
    canvas.setAttribute('state', 'Start');

]]>
</method>
...

```

The sounds themselves are resources, just like the images were. Any media you're including in your application, like video, can be represented as a resource. Make a directory called C:\openlaszlo\Server\lps-3.1.1\my-apps\sounds, and put the Knock.mp3, Ha.mp3 and Not.mp3 files into the sounds directory.

Now you can modify the `doKnockKnock`, `doHa` and `doNot` methods to play the corresponding sounds. The built-in `LzAudio` class has a `playSound()` method that plays a sound asynchronously, so the application does not wait for the sound to finish before continuing.

Reload the application and click on the knocker to hear the changes.

Adding XML data

At this point, the presentation is handled, but you still need to add the actual data. As you recall, the jokes are stored in XML file, so you need to tell OpenLaszlo how to access this file, as shown in Listing 17.

Listing 17. Adding the data

```

...
<frame src="images/KnockUp.png"/>
</resource>

<dataset name="knockknockjokes_ds"
  src="KnockKnockJokes.xml"
/>

<datapointer id="knockknockjokes_ptr"
  xpath="knockknockjokes_ds:/KnockKnockJokes"
/>

<class name="bigtext"
...

```

The data itself is bound to a data set, as specified by the data-set element. You can embed the actual XML in a data-set element, but for maintainability reasons, it's better to keep the data in a separate file. Put the file KnockKnockJokes.xml into the same directory as the application:

C:\openlaszlo\Server\lps-3.1.1\my-apps\KnockKnockJokes.xml. Notice that you've given the data set a name: knockknockjokes_ds. You'll use that name to refer to it later.

You've also created a data pointer called knockknockjokes_ptr. That points to a specific location within the data set, as specified by the `xpath` attribute. In this case, you are pointing to the root element, `<KnockKnockJokes/>`.

Now let's do something with that data.

Choosing a random joke with XPath

Next, you need to have the OpenLaszlo application choose the joke to be displayed each time. It's the `doWhosThere()` method that handles this task (see Listing 18).

Listing 18. Choosing the joke

```

...
<method name="doWhosThere"><![CDATA[
    // Count the number of <KnockKnockJoke/> element nodes.
    var jokeCount =
      knockknockjokes_ptr.getNodeCount();

    // Randomly calculate the 1-based index of a joke.
    var jokeIndex =
      1 + Math.floor(Math.random() * jokeCount);

    // Perform an XPath query to get the random joke's
    // "who" attribute.
    var who =
      knockknockjokes_ptr.xpathQuery(
        'KnockKnockJoke[' + jokeIndex + ']/@who');

    // Perform an XPath query to get the random joke's
    // text content (the full name).

```

```

    var fullName =
      knockknockjokes_ptr.xpathQuery(
        'KnockKnockJoke[' + jokeIndex + ']/text()');

    canvas.setAttribute('state', 'WhosThere');

  ]]>
</method>

...

```

As you can see from the comments, the first step is to find out how many jokes there are. Remember, you specified the `knockknockjokes_ptr` data pointer to refer to the `<KnockKnockJokes/>` root element. That means the `getNodeCount()` method will return the number of `<KnockKnockJoke/>` children.

Once you know how many jokes there are, you can use standard JavaScript techniques to choose a random joke position. Once you have the position, you can select the `who` and `fullName` values based on that position, using the data pointer and an XPath expression.

Now let's look at the data.

Presenting the joke

To view the actual joke, you need a way to pass information from the local variables in the method to the view. You can do that by creating two new string attributes on the canvas called `who` and `fullName`, as shown in Listing 19.

Listing 19. Viewing the XML joke document

```

...
<attribute name="state" type="string" value="Start"/>
<attribute name="who" type="string" value=""/>
<attribute name="fullName" type="string" value=""/>
...
<method name="doWhosThere"><![CDATA[
...
  var who =
    knockknockjokes_ptr.xpathQuery(
      'KnockKnockJoke[' + jokeIndex + ']/@who');

  var fullName =
    knockknockjokes_ptr.xpathQuery(
      'KnockKnockJoke[' + jokeIndex + ']/text()');

  canvas.setAttribute('who', who);
  canvas.setAttribute('fullName', fullName);

  canvas.setAttribute('state', 'WhosThere');

  ]]>
</method>

...

```

```
<view
  visible="${canvas.state == 'WhosThere'}"
  width="${parent.width}"
>

  <simplelayout axis="y" spacing="5"/>

  <bigtext
    width="${parent.width}"
    text="${this.escapeText(canvas.who)}"
  />
...
</view>

<view
  visible="${canvas.state == 'Who'}"
  width="${parent.width}"
>

  <simplelayout axis="y" spacing="5"/>

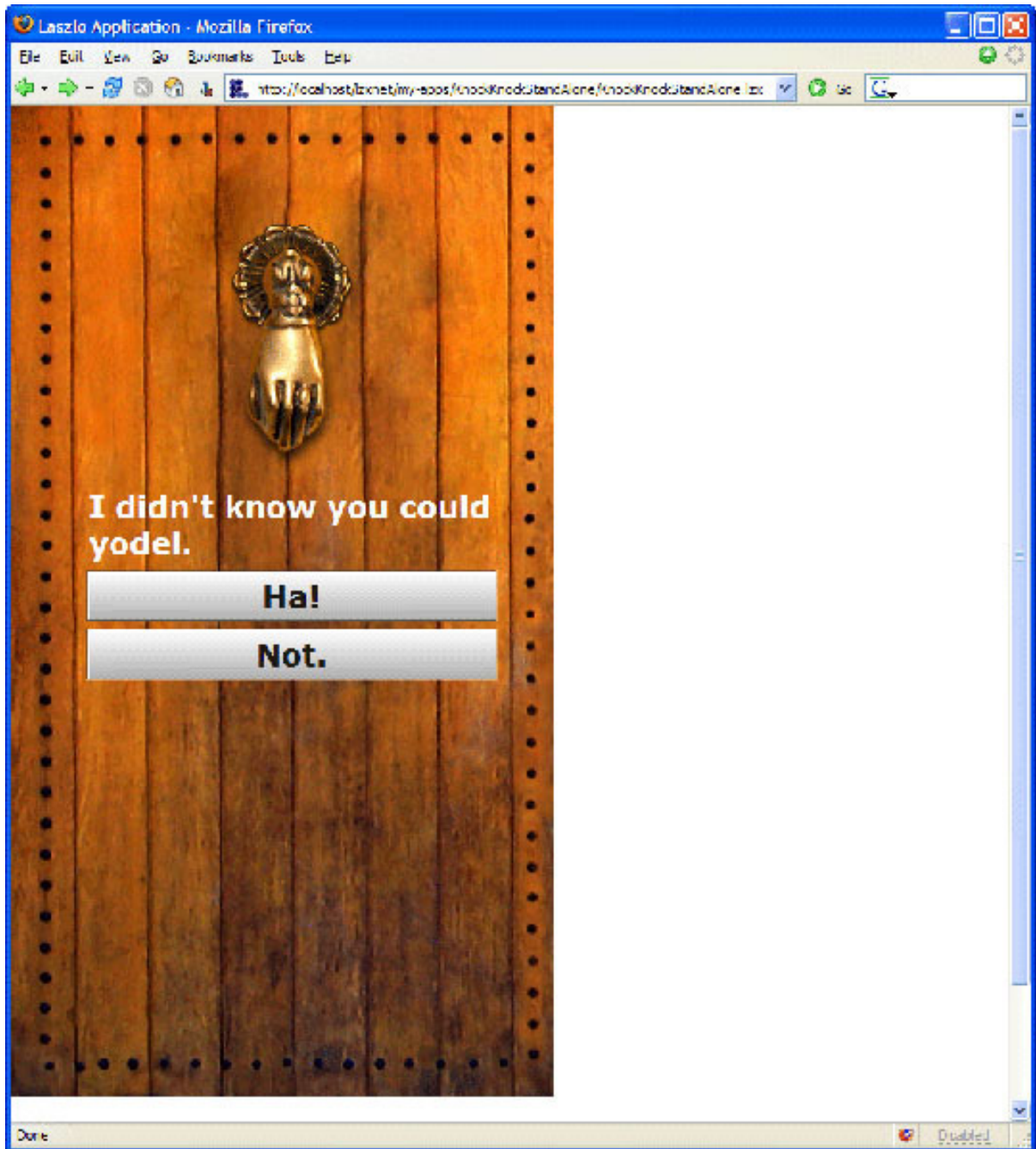
  <bigtext
    text="${this.escapeText(canvas.fullName)}"
  />
...
</view>

</view>

</canvas>
```

By using the `setAttribute` method to set the values of `who` and `fullName` for the canvas, you are not just passing the values back to the canvas but you're also making sure that any constraints involving these attributes are taken into account. That means you are indirectly setting the values of the text for your views, so when they're displayed, they have the appropriate values, as you can see in Figure 17.

Figure 17. Who state



That's it. You have a complete stand-alone OpenLaszlo application! In Part 2, you'll look at using this application to access remote data.

Section 8. Summary

In this tutorial, you learned the basics of OpenLaszlo development by building an OpenLaszlo interface for a simple PHP application. The application includes graphics, sound, and interactivity, as well as native access of XML data. The OpenLaszlo application was rendered in your browser as a Flash movie based on the installation and configuration you did earlier in the tutorial. In Part 2, you'll expand the OpenLaszlo application so it receives data not only from a static XML file but from a PHP-based Web service, too.

Downloads

Description	Name	Size	Download method
Part 1 source code	os-php-openlaszlo1.source.zip	541KB	HTTP

[Information about download methods](#)

Resources

Learn

- See [OpenLaszlo applications](#) in action and learn more about Laszlo at the [OpenLaszlo wiki](#).
- Visit [OpenLaszlo.org](#) for the latest downloads, news, articles and announcements of OpenLaszlo.
- Find like-minded developers in the [OpenLaszlo Developer Zone](#).
- For other good resources for learning OpenLaszlo, check out the [OpenLaszlo overview](#) and the [Laszlo Documentation](#).
- See more of what OpenLaszlo can do by checking out the [Laszlo Developer Community](#).
- The author uses the Laszlo pie menus in his own OpenLaszlo applications, including the [Sims Content Catalog](#), [John von Neumann's 29 State Cellular Automata](#), and the [Pie Menus -vs- Linear Menu Experiment](#).
- Learn the basics of PHP with the "[Learning PHP](#)" series of tutorials from developerWorks.
- For yet more information, check out the [official PHP tutorial](#) on PHP.net.
- Learn the basics of XML with the "[Introduction to XML](#)" tutorial and learn more about the Document Object Model with "[Understanding DOM](#)."
- The "[Getting started with XPath](#)" tutorial will help you understand how OpenLaszlo locates XML data.
- Stay current with [developerWorks technical events and webcasts](#).
- Browse all of the [PHP content](#) on developerWorks.
- Expand your PHP skills by visiting IBM developerWorks [PHP project resources](#).
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Visit PHP.net for [downloads](#) and [documentation](#).
- Apache is popular, free, and supports many modules and extensions. Download the [Apache HTTP Server](#) from [Apache.org](#). And be sure to see the [installation instructions](#).
- The [IDE for Laszlo Project](#) is an open source Eclipse technology project,

donated by [alphaWorks](#).

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Nicholas Chase

Nicholas Chase has been involved in Web-site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).