



Using Zen Mojo

Version 1.1.2
11 October 2016

Using Zen Mojo

Caché Version 1.1.2 11 October 2016

Copyright © 2016 InterSystems Corporation

All rights reserved.



InterSystems, InterSystems Caché, InterSystems Ensemble, InterSystems HealthShare, HealthShare, InterSystems TrakCare, TrakCare, InterSystems DeepSee, and DeepSee are registered trademarks of InterSystems Corporation.

InterSystems iKnow, Zen, and Caché Server Pages are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support

Tel: +1 617 621-0700

Fax: +1 617 374-9391

Email: support@InterSystems.com

Table of Contents

About This Book	1
1 Overview of Zen Mojo	3
1.1 Introduction to Zen Mojo	3
1.1.1 Useful Background Knowledge	4
1.2 Kinds of Zen Mojo Applications	4
1.2.1 Web Applications	5
1.2.2 Mobile Applications	6
1.2.3 Hybrid Applications	6
1.3 Basic Definition of a Zen Mojo Page	6
1.4 Zen Mojo Helper Plugins and Layout Graphs	7
1.5 Zen Mojo Templates	8
1.5.1 Content Objects	8
1.5.2 Keys	9
1.5.3 Content Objects and Keys	10
1.5.4 The Template System	11
1.6 The Zen Mojo Content Cache	13
1.7 Event Handling	14
1.8 How Zen Mojo Works	15
1.8.1 The Zen Mojo Page Object	16
1.9 Zen Mojo Version Numbers	16
2 Zen Mojo Tutorials	19
2.1 Tutorial 1: Layout Graphs	19
2.1.1 Tutorial 1: Getting Started	19
2.1.2 How This Sample Works	21
2.1.3 Exercises	23
2.2 Tutorial 2: Data Objects	23
2.2.1 Tutorial 2: Getting Started	24
2.2.2 How This Sample Works	25
2.2.3 Exercises	27
2.3 Tutorial 3: Event Handling	28
2.3.1 Tutorial 3: Getting Started	28
2.3.2 How This Sample Works	28
2.3.3 Exercises	30
3 Background Information and Tasks	31
3.1 General-Purpose Client Variables and Functions	31
3.2 Stashing Values	32
3.3 Converting a JSON Object to a String	32
3.4 Establishing Naming Conventions	33
3.4.1 Class Names	33
3.4.2 Method Names	33
3.4.3 Content Objects and JSON Providers	34
3.4.4 Keys	34
4 Summary of Callbacks and Event Handlers	35
4.1 The ongetdata and ongetlayout Callback Attributes	35
4.1.1 When These Callbacks Are Used	35

4.1.2	Specifying ongetdata or ongetlayout	36
4.1.3	Details for getContent()	36
4.1.4	Defining onGetContent()	37
4.1.5	Defining %OnGetJSONContent()	37
4.2	Other Callback Attributes	39
4.3	Event Handlers	39
4.3.1	Where Events Are Supported	39
4.3.2	Implementing Event Handlers	40
5	Creating the Classes	41
5.1	Choosing Plugins and Downloading Files	41
5.2	Using the Zen Mojo Wizard	41
5.3	Creating Zen Mojo Classes Manually	42
5.3.1	Defining the Zen Mojo Application Class	42
5.3.2	Defining a Zen Mojo Page Class	42
5.3.3	Defining a Template Class	43
5.4	Creating a Basic pageContents Definition	44
5.4.1	Registering Plugins	44
5.4.2	Using Multiple documentViews	45
5.5	Next Steps	45
6	Defining Layout Methods	47
6.1	Overview of Layout Methods and Layout Graphs	47
6.2	Referring to the Current Data Object	48
6.3	Invoking Template Methods in a Layout Graph	49
6.4	Specifying Style Attributes in Layout Graphs	51
6.5	Using the sourceData Property in a Layout Graph	51
6.6	Using Stashed Values within sourceData	52
7	Using Keys	53
7.1	Possible Approaches	53
7.1.1	Driving One documentView from Another documentView	53
7.1.2	Using a Document Stack	54
7.2	Specifying the Initial Keys for a documentView	54
7.3	Setting Keys and Updating the Layout	54
7.4	Using the Criteria Argument	55
7.5	Getting Keys of a documentView	56
7.6	Working with Document Stacks	56
8	Interacting with Layout Objects	59
8.1	Methods in This Chapter	59
8.2	Getting or Setting the Value of a Layout Object	59
8.3	Refreshing a Layout Object	60
8.4	Accessing a Layout Object	60
9	Submitting Data to the Server	61
9.1	Submitting Data to the Server	61
9.2	Example	62
10	Working with Multiple Templates (Explicit Dispatch)	63
10.1	Introduction to Explicit Dispatch	63
10.2	Associating Areas with Templates	63
10.3	Setting the Current Area and Key for the Page	64
10.3.1	Implementing changeAreaHandler()	65

10.4 Accessing the Current Template	65
11 Working with Multiple Templates (Dynamic Dispatch)	67
11.1 Introduction to Dynamic Dispatch	67
11.1.1 How Zen Mojo Finds Templates, in Dynamic Dispatch	68
11.2 Modifying the Page to Use Dynamic Dispatch	68
11.3 Defining the Templates	68
12 Using Style Sheets	71
12.1 Specifying Style Information for Layout Objects	71
12.2 Including Style Sheets	71
12.3 Precedence of Styles	72
13 Additional Steps for Mobile and Hybrid Applications	73
13.1 Specifying Meta Tags	73
13.2 Modifying the Page Class to Support Offline Use	74
13.3 Requirements of the Packaged Application	74
13.4 Generating Offline Pages	75
13.4.1 Generating Only the Offline Page	75
13.4.2 Generating an Offline Bundle	75
Appendix A: Tips for Managing Caching During Development	77
A.1 CSP Gateway Cache	77
A.2 Browser Cache	78
A.3 Zen Mojo Page Cache	78
Appendix B: Upgrade Steps for Beta Program Customers	79
B.1 Class Renaming	79
B.1.1 Upgrade Steps	79
B.2 Namespace Changes	79
B.2.1 Upgrade Steps	80
B.2.2 Example	80
B.3 Plugin Refactoring	80
B.3.1 Upgrade Steps and Examples	81
B.4 Event Handling Changes	82
B.4.1 Upgrade Steps	82

About This Book

This book helps programmers use Zen Mojo to create web applications, for mobile devices or for desktops. It consists of the following chapters:

- [Overview of Zen Mojo](#)
- [Zen Mojo Tutorials](#)
- [Background Information and Tasks](#)
- [Summary of Callbacks and Event Handlers](#)
- [Creating the Classes](#)
- [Defining Layout Methods](#)
- [Using Keys](#)
- [Interacting with Layout Objects](#)
- [Submitting Data to the Server](#)
- [Working with Multiple Templates \(Explicit Dispatch\)](#)
- [Working with Multiple Templates \(Dynamic Dispatch\)](#)
- [Using Style Sheets](#)
- [Additional Steps for Mobile and Hybrid Applications](#)
- [Tips for Managing Caching During Development](#)
- [Upgrade Steps for Beta Program Customers](#)

For a detailed outline, see the [table of contents](#).

Also see the following related books:

- [Using Zen Mojo Plugins](#) provides details on working with Zen Mojo plugins in general, as well as specific information on the plugins currently with Zen Mojo.
- [Configuring and Using Ensemble Push Notifications](#) describes how to use the Ensemble push notification feature, which you can use in hybrid Zen Mojo applications.

For information on accessing the widget reference, see “[Widget Reference](#)” in the chapter “[Using the Plugin Documentation and Widget Reference Apps](#)” in *Using Zen Mojo Plugins*. For information on accessing the plugin reference, see “[Plugin Documentation](#),” also in that chapter.

For general information, see the *InterSystems Documentation Guide*.

1

Overview of Zen Mojo

This chapter provides an overview of Zen Mojo and its central concepts. It discusses the following topics:

- [Introduction to Zen Mojo](#)
- [Kinds of Zen Mojo applications](#)
- [Basic definition of a Zen Mojo page](#)
- [Zen Mojo plugins and layout graphs](#)
- [How a page uses a template](#)
- [The Zen Mojo content cache](#)
- [Event handling](#)
- [How Zen Mojo works](#)
- [Zen Mojo version numbers](#)

If you are familiar with Zen, you will find some of these concepts familiar. You can, however, create Zen Mojo applications without having any familiarity with Zen.

Important: If you were a member of the Zen Mojo beta program, see the appendix “[Upgrade Steps for Beta Program Customers](#).”

1.1 Introduction to Zen Mojo

Zen Mojo is a set of classes that enable you to create web pages, suitable for either mobile devices or desktops. Zen Mojo presents the following key features:

- The architecture is designed to minimize how often pages must be loaded, so that you can avoid this expensive operation. You can radically alter a single page without reloading it.
- All communication between the client and the server is packaged as JSON, which is an extremely lightweight format that consumes little bandwidth.
- Zen Mojo pages automatically generate HTML 5.0, which is supported by browsers on all modern mobile devices. You can include custom HTML if needed, but in general it is not necessary to write HTML.

- Via its plugin technology, Zen Mojo provides easy access to common, familiar third-party JavaScript libraries. Rather than needing to write complex JavaScript code that renders layout objects, you can create much simpler JavaScript objects that simply describe the desired layout.

You can also extend Zen Mojo by creating your own plugins to use additional third-party libraries.

- Zen Mojo provides a rich set of tools that you can use to modify the page in response to user actions.

In a Zen Mojo application, your classes define both client methods and server methods, and the client methods are much more common. A *client method* is written in JavaScript and runs in the browser. A *server method* is written in one of the Caché server-side languages, typically Caché ObjectScript.

1.1.1 Useful Background Knowledge

To create web pages with Zen Mojo, it is necessary to know JavaScript, CSS, and some Caché ObjectScript.

It is assumed that you are reasonably familiar with one or more of the JavaScript libraries used by the Zen Mojo plugins.

If you are creating a hybrid application (see the next section), it is necessary to be acquainted with PhoneGap, a third-party tool not described in this book.

It is also helpful if you can read extremely simple and short XML documents, and if you know the XML terms *element*, *attribute*, and *namespace*. If you have not been exposed to XML, however, just carefully mimic the format of the XML documents presented in this book and in the Zen Mojo samples. It is not necessary to have a detailed understanding of XML, and this book will provide tips where needed.

Acquaintance with Zen can be helpful, but is not necessary.

1.2 Kinds of Zen Mojo Applications

Zen Mojo is designed for client-server interaction, and you can use it to create applications whose web clients are on either mobile devices or on desktops. The server is a Caché server. There are several kinds of applications you can create with Zen Mojo:

- Web applications designed for desktops.

This type of application is the most familiar to existing Caché users, who have previously built web applications in Zen and CSP. Note that in contrast to Zen and CSP, a Zen Mojo web application consists of a single page.

- Web applications designed for mobile devices. As with the previous item, this application consists of a single page. In this case, the page is either specifically tailored for a mobile environment or uses responsive design and automatically adapts to various mobile devices. In general, web applications only have limited access to a mobile device's native capabilities. They may also need some fine-tuning to run smoothly on different mobile devices.

To create this kind of application, you would generally use different plugins than for desktop web applications. Some plugins are better suited to mobile environments, and others are better suited for desktops.

- Hybrid applications for mobile devices. A hybrid application consists of a native application running a web application in an embedded (and therefore invisible) browser. The native container provides access to the device's capabilities. The application can also use push notifications.

The following subsections describes the parts of the [web application](#) for Zen Mojo, and additional considerations for [mobile](#) and [hybrid](#) applications.

1.2.1 Web Applications

Formally, a Zen Mojo web application consists of the following parts:

a page class

A Zen Mojo page class is a class derived from `%ZEN.Mojo.basePage`. Zen Mojo is designed so that your application can use a single, fairly simple page class.

template classes

A Zen Mojo template class is a class derived from `%ZEN.Mojo.Component.contentTemplate`. A template class provides all the logic for your application, including data and layout information.

You can have multiple template classes to expand your application.

If you are familiar with Zen, note that Zen Mojo templates are unrelated to Zen templates.

an application class

An optional class derived from `%ZEN.Mojo.baseApplication` that provides application-wide behavior such as the style sheet. The style sheet is specified as an XData block in the class, and this block contains CSS style instructions. You can also place a style sheet within the page class, but you may find it convenient to keep style information in the application class.

supporting classes

Zen Mojo provides a set of supporting classes, including plugin classes. In your implementation, you specify indirectly how the supporting classes are used. You do not generally subclass them.

JavaScript include files

Many of the plugin classes require an external, third-party JavaScript include files.

CSS style sheets

The plugin classes typically use one or more CSS style sheets, which help define the appearance of the layout objects provided by the plugin. You can use the third-party style sheets that accompany the plugins or you could develop your own style sheets.

web application definition

A web application definition is a configuration that resides in the CACHESYS database. You create application definitions in the Management Portal, in the same part of the portal where you define users and roles.

This configuration specifies information such as how the application can be accessed, the default timeout period, default error pages, and other details.

When you create a namespace, the Management Portal provides the option of creating an associated web application at the same time. You could use this default application or define another. In either case, you should review the application definition and make sure that it is appropriate for your needs.

For specific guidance, see “Zen Application Configuration” in *Developing Zen Applications*; this section is relevant because Zen Mojo is based on Zen. For more generic information on web applications, see “Web Applications” in the *Caché Security Administration Guide*.

Your classes (and the underlying Zen Mojo classes) define both client methods and server methods. A client method is written in JavaScript and runs in the browser. A server method is written in one of the Caché server-side languages, typically Caché ObjectScript. As noted previously, most of the methods in a typical Zen Mojo application are client methods.

1.2.2 Mobile Applications

To use Zen Mojo to create an application for a mobile environment, you create a Zen Mojo page and then generate a static HTML version of the page (this is the *offline version* of the page), collect the needed JavaScript libraries and CSS files, package these items together as a unit, and distribute the unit as a web application via the same channels used for other mobile applications.

To package the items as a unit, you use a third-party tool such as PhoneGap, not described in this book.

This application contacts the server when appropriate and also provides minimal functionality when the server is not available. The details depend upon the application. It is important to consider the following points:

- Choose plugins that use libraries that are designed for use on mobile devices. For example, the JQuery Mobile and Chocolate-Chip UI libraries are meant for use on mobile devices, but the Dojo libraries are meant for use on larger screens.
- The web page should be small, simple enough to display and use on smaller screens, and should be responsive to screen rotation.
- You should ensure that the page provides at least minimal functionality without needing access to the server.

For example, when the user initially accesses the application, if the server is inaccessible, the page could use data that was previously cached.

1.2.3 Hybrid Applications

For a hybrid application, you also must create a native container specific to the device. To create the native container, use using SWIFT or Objective-C (for iOS devices), Java (for devices running Android, or a language from the .NET family (for Windows phones). Caché provides all the options needed to connect your native interface with your data and application logic. Caché provides support for JSON and REST, so you can use JSON over REST services. Caché also supports web services (still frequently used for this purpose). You can also create custom HTTP requests if needed. For any approach, Caché provides a session context, and it is easy to switch to a secured connection using SSL3.

In a hybrid application, the client methods of your web page can access hardware features of the mobile device.

Your application can also use push notifications. For details on push notifications, see *Configuring and Using Ensemble Push Notifications*.

To package the native container with the offline page and other components, use PhoneGap, not described in this book.

1.3 Basic Definition of a Zen Mojo Page

To define a Zen Mojo page, you define a page class that includes an XData block like the following simple example:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen"
xmlns:mojo="http://www.intersystems.com/zen/mojo"
layout="none">

<mojo:documentView id="mainView"
ongetlayout = "return zenPage.getContent('layout',key,criteria);"
ongetdata = "return zenPage.getContent('data',key,criteria);">
<mojo:mojoDefaultPageManager>
  <mojo:HTML5Helper/>
</mojo:mojoDefaultPageManager>
</mojo:documentView>

</pane>
}
```

The next chapters provide details on the page class. For now, let us just examine this XData block.

The text within the curly braces is an XML document that Zen Mojo uses to generate the page. This example definition consists of one `<mojo:documentView>` element (or more simply, *documentView*). This element is the general-purpose container provided by Zen Mojo. This simple example demonstrates a typical page, and the following notes use this example to provide an overview:

- The `ongetData` callback specifies how to retrieve data for use in this `documentView`. When this `documentView` retrieves data, Zen Mojo invokes the method specified by `ongetData`, passing several arguments to that method. (As we will see later, this callback returns a JSON object.)
- The `ongetLayout` callback specifies how to build the layout of this `documentView`. When this `documentView` builds its visual appearance, Zen Mojo invokes the method specified by `ongetLayout`, again passing several arguments to that method.

The `ongetLayout` callback returns a JSON object that is a Zen Mojo layout graph. A *layout graph* is a combination of layout objects. *Layout objects*, in general, include items such as paragraphs, tables, charts, groups, menus, check boxes, buttons, and other web page elements.

Where suitable, the layout objects refer to data returned by the `ongetData` callback. For example, a dropdown list can display a list of values returned by the `ongetData` callback.

- This `documentView` contains one child element, `<mojo:mojoDefaultPageManager>`, which is a *page manager plugin* or *page manager* provided by Zen Mojo. Each `documentView` must contain exactly one page manager, which automatically manages the page rendering.

Zen Mojo provides multiple page managers, some of which can be used only with specific helper plugins. See the next bullet item.

- The page manager contains one child element, `<mojo:HTML5Helper/>`. This child element is a *helper plugin* provided by Zen Mojo and it defines a set of layout objects. The `ongetLayout` callback returns a layout graph that consists of layout objects defined by *this* helper plugin.

A page manager can contain multiple helper plugins, not just one as in this example. Often it is useful to use multiple helper plugins.

Tip: The `<mojo:HTML5Helper/>` element is empty. Notice that the example contains only a single empty tag for this element rather than a matching opening and closing tag. An opening tag has the form `<element>` and a closing tag has the form `</element>`, as you can see in the example above. An empty tag has the form `<element/>`

The [next section](#) introduces helper plugins and layout objects in more detail.

1.4 Zen Mojo Helper Plugins and Layout Graphs

Zen Mojo uses helper plugins and layout graphs to determine the appearance of your web page. A *helper plugin* is a class that defines a set of layout objects, each which represents a graphical element. The plugin class defines methods that specify how to render those elements, typically using third-party JavaScript libraries.

You can include multiple helper plugins in a `documentView`. Currently, Zen Mojo provides the helper plugins that provide access to the ChocolateChip-UI framework, the Dojo library, the Google Maps API, the jQueryMobile library, and HTML5 standard. Also, a default helper provides special utility objects for use in layout graphs. You can create your own helper plugins, as well.

A *layout graph* is a JavaScript object that describes the layout in terms of layout objects. It is not necessary for you to write complex JavaScript or HTML rendering code, because Zen Mojo generates this code for you, based on your layout graphs. The following shows an example layout graph:

```
content = {
  children:[
    {type: '$header', caption: '[sectionHeader]'},
    {type: '$listview', value: '[personList]', filter: true, children: [
      {type: '$listviewitem', key: 'drill-person', value: '[id]',
        label: '[name]', content: '[ssn]', clickable: true}
    ]}
  ]
};
```

In this example, `$header`, `$listview`, and `$listviewitem` are types of layout objects and are provided by a plugin (specifically the jQuery Mobile helpers). Other information in this layout graph specifies data to display in these layout objects and describes the behavior of these elements.

1.5 Zen Mojo Templates

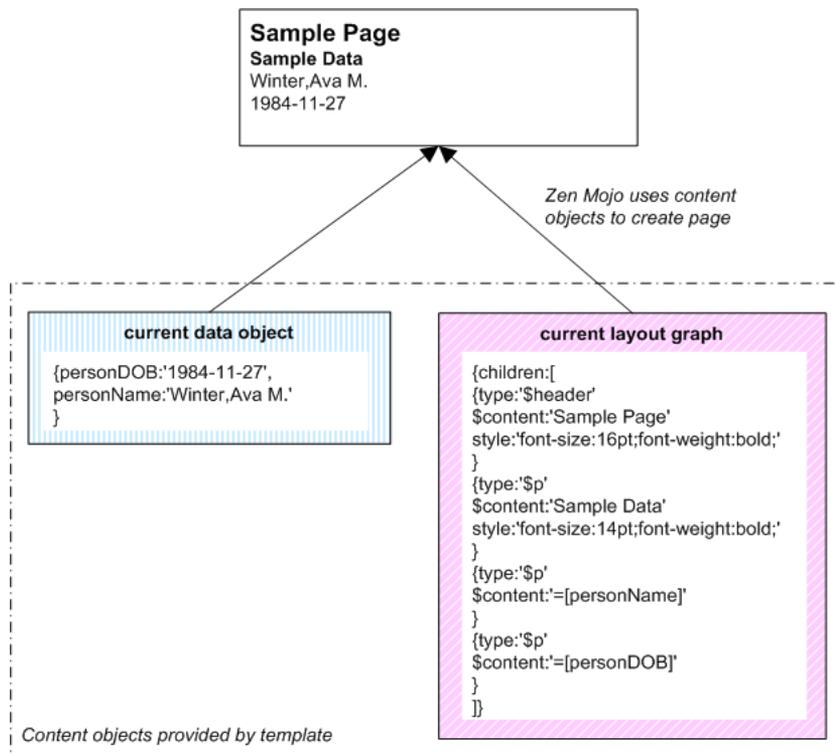
The purpose of a Zen Mojo template is to define the appearance and behavior of a Zen Mojo page. It is important, therefore, to understand how a Zen Mojo page uses a template. This section discusses the following topics:

- [Content objects](#)
- [Keys](#)
- [Content objects and keys](#)
- [Templates](#)

For simplicity, [caching](#) is discussed later in this chapter.

1.5.1 Content Objects

For any `documentView`, at any given time, Zen Mojo uses two JSON objects to determine the contents of that `documentView`. Generically, these JSON objects are known as *content objects*. The following figure shows an example (with a page that consists of a single `documentView`):



The two content objects are called the data object and the layout graph.

The current *data object* contains data that is currently available for use in the `documentView`. Often, but not necessarily always, this is obtained from the server.

The current *layout graph* determines the current appearance of the `documentView`. In its basic form, this object consists of a combination of the following:

- Zen Mojo layout objects that are defined in the relevant plugins
- References to the values provided by the data object
- Constant values and other JavaScript expressions

The page retrieves both content objects from the template class and then uses them to generate the `documentView`.

In the next chapter, [Tutorial 1](#) and [Tutorial 2](#) provide a demo of these objects.

1.5.2 Keys

Zen Mojo uses a system of keys to support user interaction. This system works as follows:

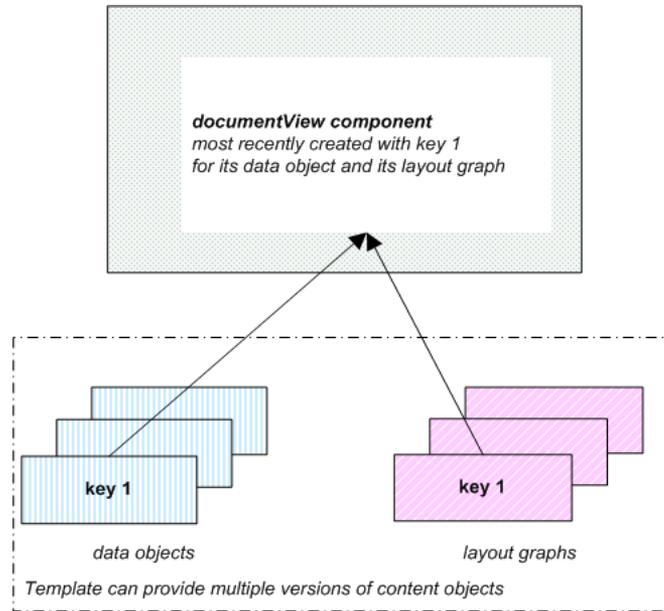
- Within a layout graph, you associate keys with items that can be selected, such as menu items, buttons, and so on.
- Within page callbacks, the current key is available as the JavaScript `key` variable. For example, if a user clicks an item in a dropdown list, Zen Mojo invokes the `onselect()` callback. In this callback, the `key` variable contains the key of the selected list item.
- Zen Mojo provides methods to set the current keys for the data object and layout graph, respectively. It also provides a method to update the layout.

In a common scenario, a user selects an option, which invokes the `onselect()` callback. This callback then updates the data and layout keys and then updates the layout, changing the appearance of the page.

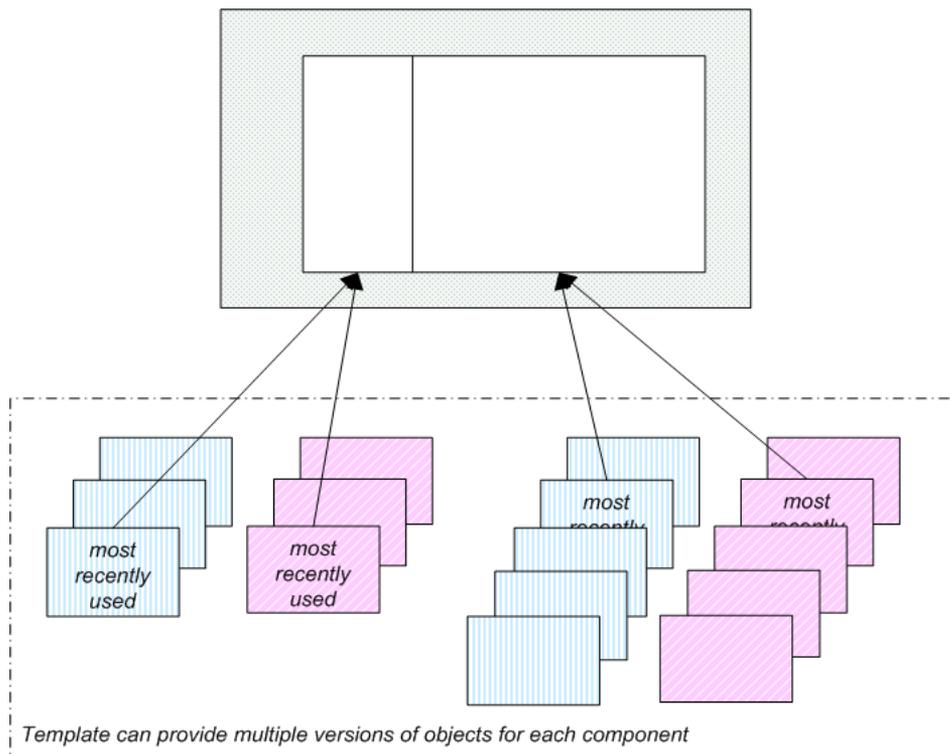
Note that additional variables are available within the callbacks, so your logic does not have to depend entirely on the keys.

1.5.3 Content Objects and Keys

The content objects are key-specific. That is, when a `documentView` retrieves a data object or a layout graph, it uses a specific key.



Depending on the plugins you use, a page can have multiple `documentView` components, and the preceding consideration applies to each of them.



1.5.4 The Template System

A Zen Mojo template is responsible for all application logic, which means that it is responsible for returning all the data and layout graphs needed on a page. Ultimately, a Zen Mojo page uses two methods that you implement in your template class: `onGetContent()` and `%OnGetJSONContent()`.

Similarly, a Zen Mojo template is responsible for submitting data to the server. In this case, a Zen Mojo page uses the `%OnSubmitData()` method that you implement in the template class.

This section describes how the page and template work together and explains how and when these methods are used.

1.5.4.1 Getting Content Objects

The page uses a single, central method (`getContent()`) to obtain content objects from the template. This is an existing method provided by Zen Mojo. So that you can specify the object to return, `getContent()` has arguments that enable you to specify:

- The content object to retrieve — this is the *providerName* argument
- The key to use when retrieving that content object — this is the *key* argument

There are other arguments as well, but they are not critical to this discussion.

The method `getContent()` invokes the `onGetContent()` and `%OnGetJSONContent()` methods in the template class and passes these parameters to them.

The following steps outline the overall flow of logic. For simplicity, these steps do not discuss the Zen Mojo cache mechanism, which is discussed [later](#) in this chapter.

1. A user performs some action in response to which your page is designed to get new data from the server, modify the display, or both. This action might be to press a button.
2. That action invokes the `getContent()` method of the page class.

This is a client method provided by Zen Mojo. Recall that all client methods are written in JavaScript and run in the browser.

3. The `getContent()` method invokes another Zen Mojo method, an internal method not meant for direct use.
4. That internal method does some processing and then invokes the `onGetContent()` method in the associated template class.

`onGetContent()` is an application-specific client method.

5. The `onGetContent()` method is intended to be a dispatcher. It examines the *providerName* argument and then either:
 - Invokes another client method, which returns the requested object.

`onGetContent()` passes the *key* value and other arguments to that method, so that the returned object is appropriate for the given key.
 - Returns null. In this case, see the next step.

6. If `onGetContent()` returns null, the template invokes the `%OnGetJSONContent()` method in the template class.

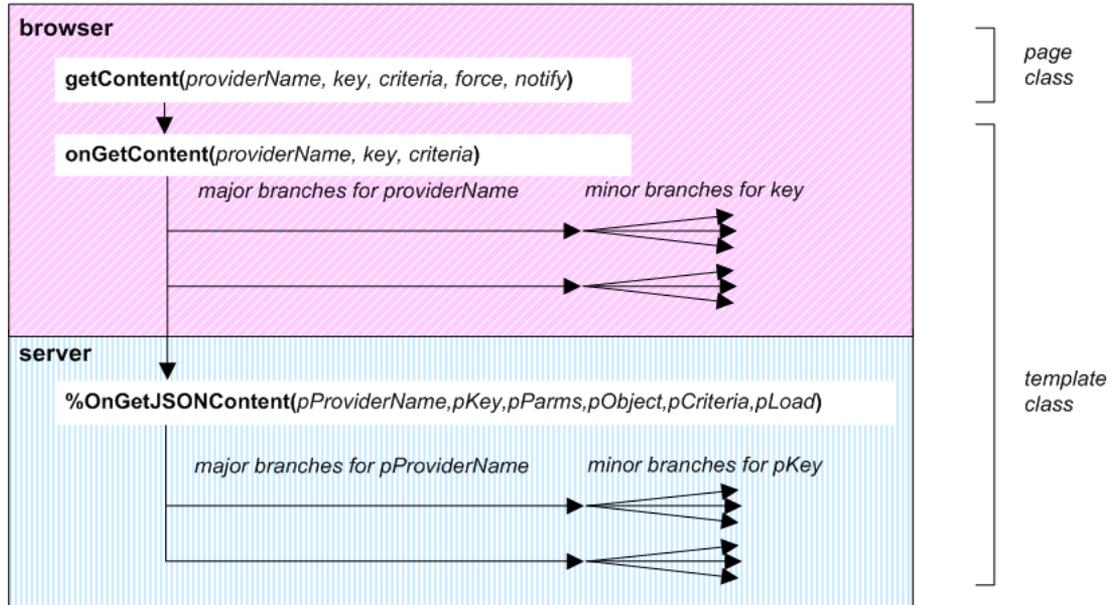
The `%OnGetJSONContent()` method is an application-specific server method. Recall that server methods are written in Caché ObjectScript and run on the server.

7. The `%OnGetJSONContent()` method first examines the *pProviderName* argument, to determine the `documentView` for which the object is needed.

Then an inner branch examines the *pKey* argument to determine which version of the object to return.

The method then constructs and returns the object to the page, and the page then uses it.

The following figure summarizes these steps:

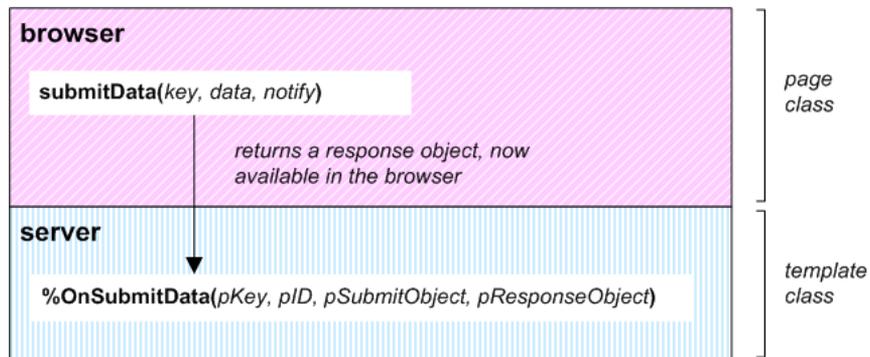


Commonly, **onGetContent()** returns layout graphs (because these should not require communicating with the server), and **%OnGetJSONContent()** returns data objects.

1.5.4.2 Submitting Data

When the page sends data to the server, the template provides the needed logic. The overall flow of logic is as follows:

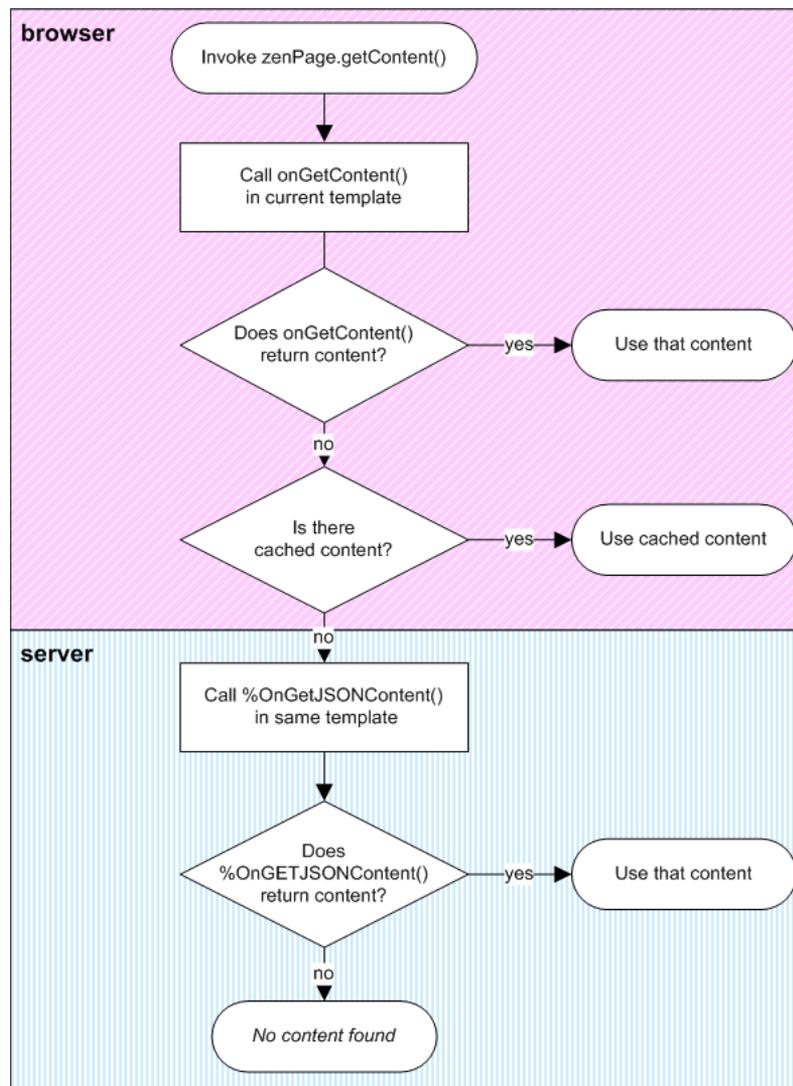
1. A user clicks a Save or Submit button or performs some other action on the page.
 Your implementation of this button invokes the **submitData()** page method.
 The **submitData()** page method takes three arguments:
 - *key* — The key to use when sending data to the server.
 - *data* — A JSON object that carries the data to be sent to the server.
 - *notify* — Either null or a function. If this argument is a function, the page sends the data *asynchronously* to the server (rather than synchronously). Then , when the data has been sent, the page executes the function named by this argument.
2. The **submitData()** method invokes the **%OnSubmitData()** method in the associated template class.
%OnSubmitData() is an application-specific method that you have implemented.
3. The **%OnSubmitData()** method receives the key and submit object as input.
 This method first examines the key to determine which branch of code to use. Then it examines and uses the submit object, as needed by your application. Finally, it creates a response object and returns it as output. Zen Mojo then makes the response object (another JSON object) available on the page.



1.6 The Zen Mojo Content Cache

To avoid unnecessary calls to the server (and to avoid other unnecessary calls to the current template), Zen Mojo caches content in a local array and uses that cache when possible.

Specifically, the page method `getContent()` uses the cache if it is available. If there is no cached content for the given *key* and *criteria* arguments, `getContent()` then invokes `onGetContent()` of the current template, as described earlier in this chapter.



When you need to force Zen Mojo to call `%OnGetJSONContent()` for a given `documentView`, you can invalidate the cache for that `documentView`. You would do this if you needed to force the page to retrieve new data from the server.

1.7 Event Handling

When an event occurs on the page, Zen Mojo automatically invokes an event handler in the associated template class. The Zen Mojo *event handlers* are the `onselect()`, `onchange()`, and `onevent()` methods, which have no behavior by default.

- If a user selects a layout object, Zen Mojo invokes the `onselect()` method.
- If a user changes the value of a layout object, Zen Mojo invokes the `onchange()` method.
- If an event of another type occurs, Zen Mojo invokes the `onevent()` method.

To make your page interactive, define any or all of these methods in your template class. For each method, Zen Mojo automatically passes information about the context in which the event occurred. This information indicates the `documentView`, the item key, and the item value. (Notice that the layout graph should associate keys with any items that can be selected, such as menu items, buttons, and so on.) For `onevent()`, Zen Mojo also passes the event type.

For example, a user might select a menu item whose key is `key1`. Zen Mojo then invokes the `onselect()` callback, passing `key1` as the value for the `key` argument. This callback could then set the data and layout keys equal to `key1` and then update the layout, thus changing the appearance of the page.

Within your methods, you have access to tools that you can use for tasks like these:

- Setting the current keys for the data object and layout graph.
- Updating the layout.
- Submitting data to the server.
- Pushing new layouts onto a document stack (or popping them off the stack).

Zen Mojo provides methods to set the current keys for the data object and layout graph, respectively. It also provides a method to update the layout. Additional methods enable you to interact with the page and submit data to the server. Later chapters in this book describe these tools.

Note that additional variables are available within the callbacks, so your logic does not have to depend entirely on the keys.

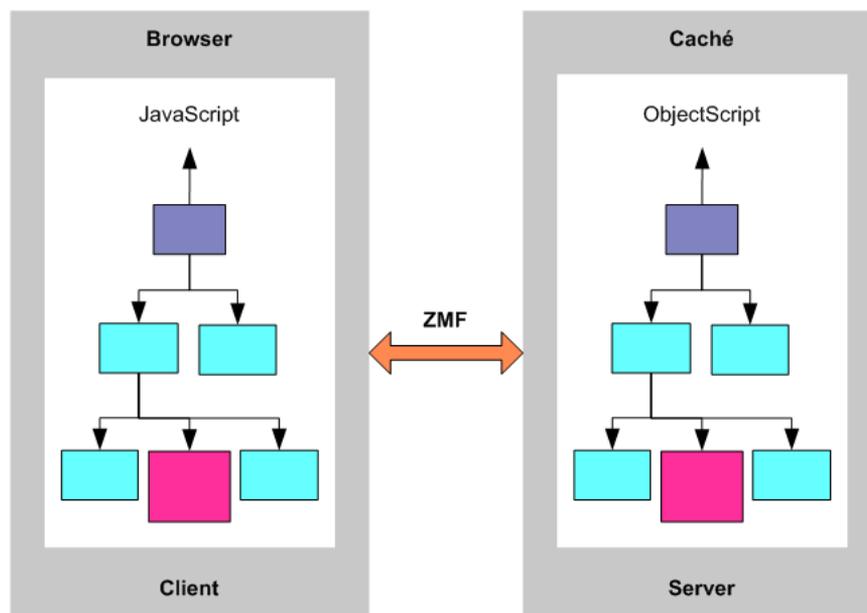
1.8 How Zen Mojo Works

Zen Mojo works much in the same way that Zen does, except that Zen Mojo is much more lightweight.

When the client sends a page request, Zen Mojo generates the page and all of its components, creating an *object tree*. The object tree generates all the CSS (style sheet), JavaScript (client logic) and HTML (layout instructions) needed to display the page within the client browser.

On the client browser, the system automatically recreates the object tree as a set of JavaScript objects. Properties of the object tree are thus available to the client. The client can also, of course, use the client methods.

This object tree is much smaller for Zen Mojo than it is for Zen. In particular, a typical Zen page includes a large number of component objects. In contrast, a typical Zen Mojo page includes one `documentView` object.



As the user interacts with a page, events are fired that invoke the various methods of the client object tree. Zen Mojo automatically manages communication with the server and also handles session context (if any), security, and synchronizing changes between the client and the server.

With Zen Mojo, it is possible for an application to run without a session, which means that the application can run without a connection to the server. In contrast, Zen does not support this scenario.

1.8.1 The Zen Mojo Page Object

It can be useful to know a few things about the object tree (and the corresponding Caché classes) that Zen Mojo uses to represent a page. This object tree includes the following:

- An instance of your page class. The page class inherits from `%ZEN.Mojo.basePage`. This class provides methods and properties that enable you to work with the page instance.
- An instance of each `documentView` that you have added to `pageContents`. Typically `pageContents` contains one or two `<mojo:documentView>` components. For some of the plugins, only a single `<mojo:documentView>` is supported; in general, if a page manager plugin is intended for use on mobile devices, that plugin uses the entire page and does not support multiple `documentViews`.

Each `<mojo:documentView>` is an instance of the class `%ZEN.Mojo.Component.documentView`. This class provides methods that enable you to work with this instance and its related objects (such as its plugins).

- Each `documentView` instance includes child objects, which are instances of the plugins specified by that `documentView`. Each page manager plugin is a subclass of `%ZEN.Mojo.Plugin.basePageManager`, and each helper plugin is a subclass of `%ZEN.Mojo.Plugin.baseHelperPlugin`.
- A *content provider* component, which is included automatically on every Zen Mojo page. This component enables the page to access and use the associated templates. The content provider is an instance of `%ZEN.Mojo.Component.contentProvider`.

The following objects are attached to the content provider:

- An instance of each template class that is associated with the page class.
- One or more *JSON providers*. A JSON provider is an instance of `%ZEN.Auxiliary.jsonProvider`. The content provider has a JSON provider for each data object that the template returns. It does not have JSON providers for the layout graphs, because those do not usually require the server.

The content provider is responsible for some of the internal processing discussed earlier in this chapter.

It is not necessary to work with the content provider directly, except when invalidating caches.

Zen Mojo provides methods to access parts of this object tree as needed. That is, it is not necessary for you to know the detailed structure of this object tree.

1.9 Zen Mojo Version Numbers

There are separate version numbers for the Zen Mojo core, for each plugin, and for the plugin reference documentation. To see these version numbers, use the `%PrintVersion()` method of `%ZEN.Mojo.Utils`:

```
do ##class(%ZEN.Mojo.Utils).%PrintVersion()
```

For example:

```
do ##class(%ZEN.Mojo.Utills).%PrintVersion()

Zen Mojo version 1.0.10

Page Managers

"0.5.3" : %ZEN.Mojo.Plugin.chui352PageManager
"0.5.2" : %ZEN.Mojo.Plugin.chuiPageManager
"1.0.2" : %ZEN.Mojo.Plugin.dojo191PageManager
"1.0.1" : %ZEN.Mojo.Plugin.dojoPageManager
"1.0.4" : %ZEN.Mojo.Plugin.jQM132PageManager
"1.0.0" : %ZEN.Mojo.Plugin.jQM143PageManager
"1.0.3" : %ZEN.Mojo.Plugin.jQMPageManager
"1.0.2" : %ZEN.Mojo.Plugin.mojoDefaultPageManager

Helper Plugins

"1.0.11" : %ZEN.Mojo.Plugin.HTML5Helper
"1.0.1" : %ZEN.Mojo.Plugin.charts101Helper
"0.5.5" : %ZEN.Mojo.Plugin.chui352Helper
"0.5.3" : %ZEN.Mojo.Plugin.chuiHelper
"1.0.5" : %ZEN.Mojo.Plugin.dojo1912DChartHelper
"1.0.5" : %ZEN.Mojo.Plugin.dojo191DijitHelper
"1.0.3" : %ZEN.Mojo.Plugin.dojo2DChartHelper
"1.0.4" : %ZEN.Mojo.Plugin.dojoDijitHelper
"1.0.4" : %ZEN.Mojo.Plugin.dojoGridX130Helper
"1.0.3" : %ZEN.Mojo.Plugin.dojoGridXHelper
"1.0.6" : %ZEN.Mojo.Plugin.googleMaps3Helper
"1.0.5" : %ZEN.Mojo.Plugin.googleMapsHelper
"1.0.1" : %ZEN.Mojo.Plugin.highCharts401Helper
"1.0.17" : %ZEN.Mojo.Plugin.jQM132Helper
"1.0.2" : %ZEN.Mojo.Plugin.jQM143Helper
"1.0.15" : %ZEN.Mojo.Plugin.jQMHelper
"1.0.7" : %ZEN.Mojo.Plugin.mojoDefaultHelper

Plugin Documentation installed: 0.8.6
```

For information on installing and accessing the Zen Mojo plugin reference documentation, see [Using Zen Mojo Plugins](#).

2

Zen Mojo Tutorials

This chapter contains tutorials to help you to become familiar with the basics of Zen Mojo. It discusses the following topics:

- [Tutorial 1: layout graphs](#)
- [Tutorial 2: data objects](#)
- [Tutorial 3: event handling](#)

The tutorials use the sample classes in the ZMbasics package, which are provided in the Zen Mojo *version number* Demos.ZIP file. For setup information, see the Readme.txt file in that ZIP file.

2.1 Tutorial 1: Layout Graphs

This tutorial explores a simple Zen Mojo page, focusing on layout graphs. This tutorial has the following parts:

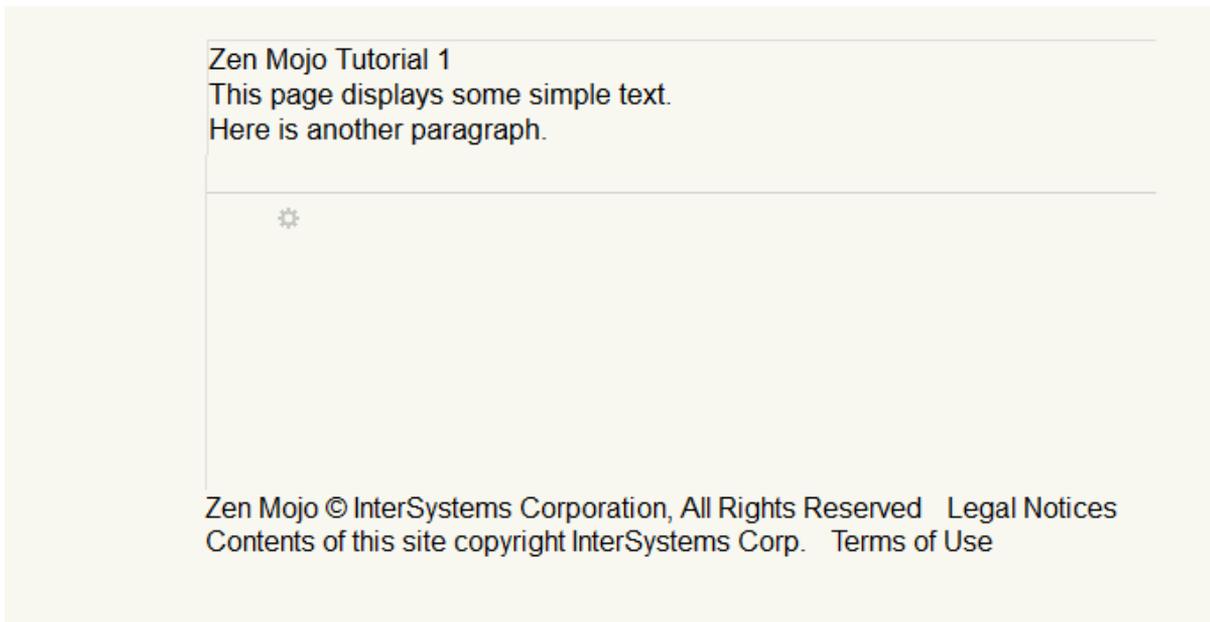
1. [Getting started with the tutorial](#)
2. [How this sample works](#)
3. [Exercises](#)

2.1.1 Tutorial 1: Getting Started

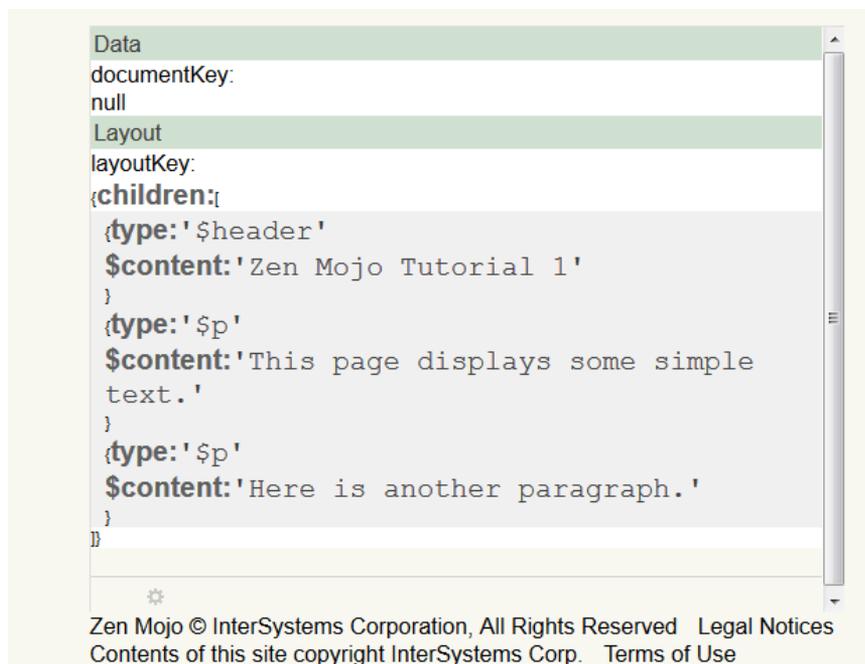
To start this tutorial:

1. Open Studio and switch to the SAMPLES namespace.
2. Open the class ZMbasics.Tutorial1.HomePage.
3. Click **View > Web Page**. Or press **F5**.

You then see the following page:



4. Click the Developer Details button . The page then looks like this:



The **Data** section displays data that is currently *available* to this part of the page. In this example, no data is available. We will examine this section again in the next tutorial.

The **Layout** section displays the information used to lay out the visual contents of this part of the page. This information is also contained in a JSON object called a *layout graph*. This object has a property named `children`, which is an array of objects.

Important: The Developer Details option does not display these objects in their literal form. It does not display valid JavaScript syntax.

5. Click the Developer Details button  again to display the page normally.

2.1.2 How This Sample Works

This section discusses how the sample works. It discusses the following topics:

- [A look at the page definition](#)
- [Introduction to getContent\(\)](#)
- [Introduction to Zen Mojo layouts](#)
- [A look at how the geometry is defined](#)

2.1.2.1 Page Definition

First, let us see how the page is defined. `ZMbasics.Tutorial1.HomePage` contains the following XData block:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen"
xmlns:mojo="http://www.intersystems.com/zen/mojo" layout="none">
<mojo:documentView id="mainView"
developerMode="true"
ongetlayout="return zenPage.getContent('mainViewLayout',key,criteria);">
<mojo:mojoDefaultPageManager>
<mojo:HTML5Helper/>
</mojo:mojoDefaultPageManager>
</mojo:documentView>

</pane>
}
```

This definition controls the `pageContents` pane, which is (in most cases) the only area of the page that you customize. In this case, the `pageContents` pane is a large rectangle in the middle of the page; for most plugins, however, the `pageContents` pane is automatically resized to fill the entire page.

Each item *within* the `pageContents` pane is a *component* and is displayed as a rectangular area within this pane. This `pageContents` pane contains one component, which is an instance of `<mojo:documentView>` (or simply `documentView`).

This `documentView` is defined as follows:

- The `id` attribute is `mainView`. This attribute determines the unique identifier of the component.
- The `developerMode` attribute is `true`, which means that the `documentView` displays the Developer Details button .
- The `ongetlayout` attribute specifies how to lay out the physical appearance of this `documentView`. This attribute is responsible for the values in the **Layout** section of the Developer Details view.

Component is a Zen term. A Zen Mojo page can include other components (such as content providers), but typically you work directly only with the `documentView` component (or multiple `documentView` components). This book thus generally uses the more specific term *documentView*.

Depending on the plugins you use, a page can contain multiple `documentViews`; in other cases, only one `documentView` is supported. In general, if a page manager plugin is intended for use on mobile devices, that plugin uses the entire page and does not support multiple `documentViews`.

2.1.2.2 Introduction to getContent()

The `ongetlayout` attribute is defined as follows:

```
ongetlayout ="return zenPage.getContent('mainViewLayout',key,criteria);"
```

The **getContent()** method is a Zen Mojo system method that returns a *content object* (a data object or a layout graph). In this example, the only argument that is used is the first one (*providerName*). Each *documentView* can have its own content objects. In this case, the page has only one *documentView*, which has one possible content object (*mainViewLayout*), as you will see later.

The **getContent()** method does the following:

1. It invokes the **onGetContent()** method of the associated template class. If that method returns content, the system returns that content.
To find the initial template class, Zen Mojo examines the *TEMPLATECLASS* parameter of the page class.
2. If **onGetContent()** returns null, the system checks to see whether the page cache contains any content. If so, the system returns that content.
3. If there is no cached content, the system invokes the **%OnGetJSONContent()** method of the associated template class, uses the result to create the content, and then returns that content.

The **onGetContent()** and **%OnGetJSONContent()** methods are application-specific methods of the template class. The **getContent()** method is a system method of the page class.

2.1.2.3 How the *documentView* Component Is Laid Out

As noted in the previous section, **getContent()** invokes the **onGetContent()** method of the associated template class. In this case, the template class is *ZMbasics.Tutorial1.Template*, and the **onGetContent()** method of that class is as follows:

```
ClientMethod onGetContent(providerName, key, criteria) [ Language = javascript ]
{
    var content = null;

    // dispatch to convenient methods
    // if content is null, then the %OnGetJSONContent method will be called

    switch(providerName) {
    case 'mainViewLayout':
        content = this.myGetMainViewLayout(key,criteria);
        break;
    }
    return content;
}
```

As you can tell by its name, **myGetMainViewLayout()** is an application-specific method. It is defined as follows:

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
    var myLayoutGraph = {};
    //The standard technique is to have a switch/case construct that creates
    //branches based on the key argument;
    //In this case only one key value is possible, so there is no need to branch

    myLayoutGraph = {
        children: [
            { type: '$header', $content: 'Zen Mojo Tutorial 1'},
            { type: '$p',      $content: 'This page displays some simple text.'},
            { type: '$p',      $content: 'Here is another paragraph.' },
        ]
    }

    return myLayoutGraph;
}
```

This method returns a Zen Mojo *layout graph*, which is a JSON object. A layout graph has a property named *children*, which contains an array of layout objects: one *\$header* and two *\$p* objects. These layout objects are defined in the helper plugin.

Note that Zen Mojo *generally* lays out the items in the order in which the layout lists them, so in this case, the header is displayed at the top, followed by the paragraphs. For finer control over placement, InterSystems recommends using CSS.

A browser uses the Zen Mojo layout graph (together with other instructions provided by Zen Mojo), generates HTML 5.0, and displays the page.

2.1.2.4 How the Sample Defines the Geometry

When you use the default page manager, the page class should define the method `adjustContentSize()`. The purpose of this method is to specify the size and position of each `documentView` in the `pageContents` pane.

This method receives, as input, the width and height of `pageContents` pane. These dimensions vary depending on the current screen size and rotation. The method also receives the input argument `load`, which is 1 when the page is loaded and is 0 at other times.

In this case, the `adjustContentSize()` method is as follows:

```
ClientMethod adjustContentSize(load, width, height) [ Language = javascript ]
{
    // This method should have an if{} block for each component.

    var mainView = zen('mainView');
    if (mainView) {
        mainView.setSize(width, height);
        var mainDiv = mainView.getEnclosingDiv();
        mainDiv.style.top = '0px';
        mainDiv.style.left = '0px';
    }
}
```

The syntax `zen('mainView')` is a reference to the `documentView` whose `id` is `mainView`. To specify the geometry of that component, the method invokes instance methods of and sets properties of that component.

2.1.3 Exercises

A good way to learn a technology is to make your own modifications to a functioning sample. Try the following exercises:

- Modify the `pageContents XData` block so that it does not include the HTML helper plugin. Recompile the page class and redisplay the page. What do you see? Then undo your change and recompile.
- In the `myGetMainViewLayout()` method, customize the `$header` or `$p` objects. For example, set the `style` attribute of each. This attribute specifies an inline CSS style to apply to that layout object.
- Disable developer mode for the `documentView` and see the page as a user would see it. Hint: To do this, modify the `pageContents XData` block.
- In the `pageContents XData` block, change the `id` of the `documentView` from `mainView` to `mainViewNEW` and recompile. How does this change affect the page? Assuming that you cannot change this `id` back to its original value, what additional changes are necessary to make the page work again?

2.2 Tutorial 2: Data Objects

This tutorial looks at a slightly more complex Zen Mojo page. This tutorial has the following parts:

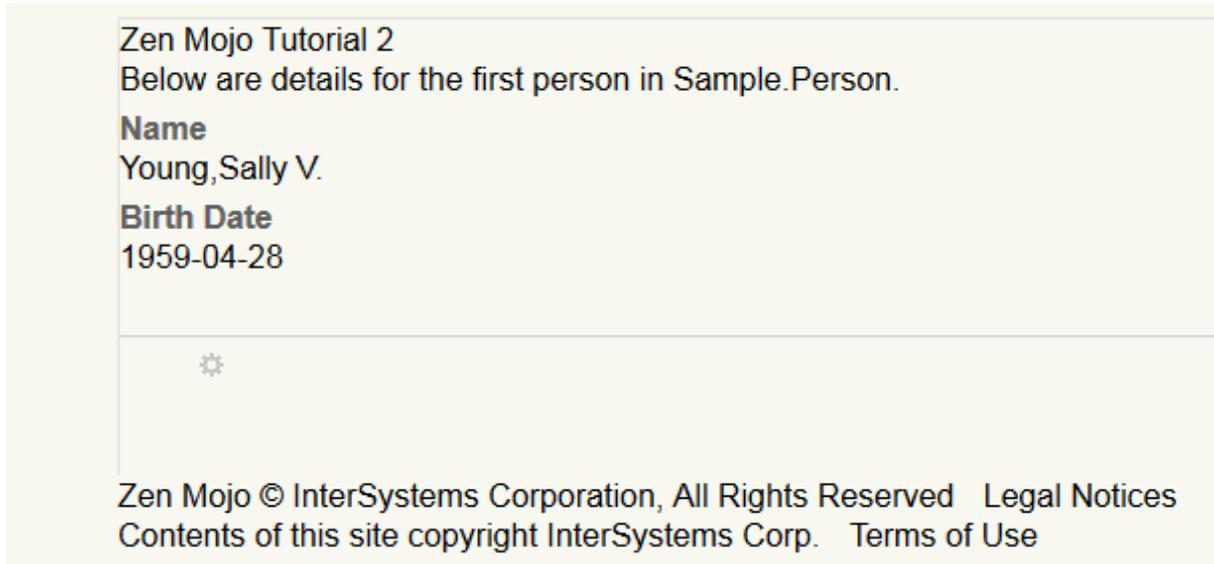
1. [Getting started with the tutorial](#)
2. [How this sample works](#)
3. [Exercises](#)

2.2.1 Tutorial 2: Getting Started

To start this tutorial:

1. In Studio, in the SAMPLES namespace, open the class ZMbasics.Tutorial2.HomePage.
2. Click **View > Web Page**. Or press **F5**.

You then see the following page (but with different data):



3. Click the Developer Details button  at the bottom of the page. The central area of the page then displays the following Developer Details view:

```

Data
documentKey:
{personDOB: '1959-04-28 '
personName: 'Young, Sally V. '
}
Layout
layoutKey:
{children:[
{type: '$header'
$content: 'Zen Mojo Tutorial 2'
}
{type: '$p'
$content: 'Below are details for the first
person in Sample.Person.'
}
{type: '$p'
title: 'Name '
$content: '= [personName] '
}
{type: '$p'
title: 'Birth Date '
$content: '= [personDOB] '
}
}]

```

Zen Mojo © InterSystems Corporation, All Rights Reserved Legal Notices
 Contents of this site copyright InterSystems Corp. Terms of Use

The **Data** section displays data that is currently *available* to this part of the page. This data is contained in a JSON object called a *data object*. This object has properties called `personDOB` and `personName`.

As we saw in the previous tutorial, the **Layout** section displays the information that is contained in the *layout graph*. Note that in this case, the layout graph refers to values that you see in the **Data** section.

2.2.2 How This Sample Works

This section discusses how the sample works. It discusses the following topics:

- [A look at the page definition](#)
- [Introduction to getting data from the server](#)
- [Introduction to JSON providers](#)
- [Another look at layout graphs](#)

2.2.2.1 Page Definition

First, let us see how the page is defined. `ZMbasics.Tutorial2.HomePage` contains the following XData block:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen"
xmlns:mojo="http://www.intersystems.com/zen/mojo" layout="none">
<mojo:documentView id="mainView"
developerMode="true"
ongetdata="return zenPage.getContent('mainViewData',key,criteria);"
ongetlayout="return zenPage.getContent('mainViewLayout',key,criteria);">
<mojo:mojoDefaultPageManager>
<mojo:HTML5Helper/>
</mojo:mojoDefaultPageManager>
</mojo:documentView>

</pane>
}
```

This definition contains one new element that was not present in the previous sample. The `documentView` specifies the `ongetdata` attribute, which specifies how to retrieve data from the server, to be available in this `documentView`. This attribute is responsible for the values in the **Data** section of the Developer Details view.

2.2.2.2 How the Sample Gets Data from the Server

The `ongetdata` attribute specifies how to retrieve the data for use by this `documentView`. This attribute is specified as follows:

```
ongetdata="return zenPage.getContent('mainViewData',key,criteria);"
```

As noted in the [previous tutorial](#), `getContent()` invokes the `onGetContent()` method of the associated template class, passing its arguments to that method. In this case, the template class is `ZMbasics.Tutorial2.Template`, and its `getContent()` method is as follows:

```
ClientMethod onGetContent(providerName, key, criteria) [ Language = javascript ]
{
    var content = null;

    // dispatch to convenient methods
    // if content is null, then the %OnGetJSONContent method will be called

    switch(providerName) {
    case 'mainViewLayout':
        content = this.myGetMainViewLayout(key,criteria);
        break;
    }
    return content;
}
```

As you can see, this method returns null if the *providerName* argument is `mainViewData`. Because the `onGetContent()` method of a template returns null in this case, Zen Mojo then automatically calls the `%OnGetJSONContent()` method of the same class. In this example, this method is as follows:

```
ClassMethod %OnGetJSONContent(pProviderName As %String, pKey As %String, ByRef pParms,
    Output pObject As %RegisteredObject, pCriteria As %RegisteredObject, pLoad As %Boolean = 0)
    As %Status
{
    // The standard technique is to have an outermost if/elseif construct
    // based on the pProviderName argument; in this case there is only one
    // possible value for pProviderName.
    if (pProviderName = "mainViewData") {

        // Within a pProviderName branch, the standard technique is to have an
        // if/elseif construct based on the key argument.
        // In this case, there are no keys, so there is no need to branch

        set pObject = ##class(%ZEN.proxyObject).%New()

        set tPerson = ##class(Sample.Person).%OpenId(1)
        set pObject.personName=tPerson.Name
        set pObject.personDOB=$zdate(tPerson.DOB,3)

    } ; additional pProviderName branches would go here

quit $$$OK
```

```
}
}
```

This method creates an instance of `%ZEN.ProxyObject` and sets properties of that object, to carry data to return to the client. (The class `%ZEN.ProxyObject` does not have any predefined properties but instead is a special-purpose container object that can have any property. For an introduction, see “Zen Proxy Objects” in *Developing Zen Applications*.)

This method returns the `%ZEN.ProxyObject` as an output object. Zen Mojo converts that object to a JSON object and passes the JSON object to the client. The client receives the data that you see in the Developer Details view:

```
Data
documentKey:
{personDOB: '1959-04-28'
personName: 'Young, Sally V.'
}
```

2.2.2.3 How the Sample Gets Data from the Server: JSON Providers

One more element is needed to make data available from the server: the `PROVIDERLIST` parameter. This parameter must list the names of all data objects to be created on the server; this must include the main branches in `%OnGetJSONContent()`.

In this example, the page class defines this parameter as follows:

```
Parameter PROVIDERLIST = "mainViewData";
```

2.2.2.4 Another Look at Layout Graphs

For this template, the following method defines the layout graph used on this page:

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
    var myLayoutGraph = {};

    //The standard technique is to have a switch/case construct based on the key argument;
    //In this case only one key value is possible, so there is to branch

    myLayoutGraph = {
        children: [
            { type: '$header', $content: 'Zen Mojo Tutorial 2' },
            { type: '$p', $content: 'Below are details for the first person in Sample.Person.' },
            { type: '$p', title: 'Name', $content: '='+[personName]' },
            { type: '$p', title: 'Birth Date', $content: '='+[personDOB]' }
        ]
    }

    return myLayoutGraph;
}
```

Note that the last two `$p` objects display data obtained from the data object. The syntax `=[name]` retrieves the value of a given property of the data object.

2.2.3 Exercises

A good way to learn a technology is to make your own modifications to a functioning sample. Try the following exercises:

- Modify the template so that it returns additional data. For example, add the following lines to `%OnGetJSONContent()`, at a suitable location:

```
set pObject.HomeCity = tPerson.Home.City
set pObject.FavoriteColors = tPerson.FavoriteColors
```

Then recompile the template class, view the Zen Mojo page, and display the Developer Details.

- Modify the **myGetMainViewLayout()** method of the template class to display the new data. For example, add a comma after `'=[personDOB]'` } and then add the following lines:

```
{ type: '$p', $content:'=[HomeCity]'},
  type: '$p', $content:'=[FavoriteColors]'
```

Then recompile the template class and view the Zen Mojo page.

- In the **myGetMainViewLayout()** method, customize the `$header` or `$p` objects.
- Change the value of the *PROVIDERLIST* parameter from `mainViewData` to `mainViewNEWProvider` and then recompile. How does this change affect the page? Assuming that you cannot change the *PROVIDERLIST* parameter back to its original value, what additional changes are necessary to make the page work again?

2.3 Tutorial 3: Event Handling

This tutorial focuses on event handling in Zen Mojo. This tutorial has the following parts:

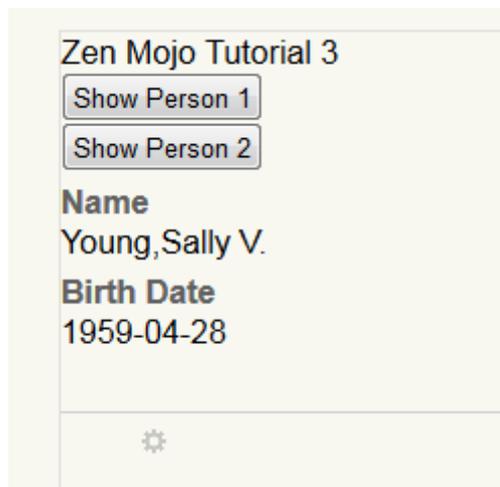
- [Getting started with the tutorial](#)
- [How this sample works](#)
- [Exercises](#)

2.3.1 Tutorial 3: Getting Started

To start this tutorial:

- In Studio, in the `SAMPLES` namespace, open the class `ZMbasics.Tutorial3.HomePage`.
- Click **View > Web Page**. Or press **F5**.

You then see the following page (but with different data):



- Press the **Show Person 2** button and notice how the display changes.

2.3.2 How This Sample Works

This section discusses how the sample works. This sample is different from the one in the [previous tutorial](#) in two ways:

- The layout graph includes button elements.
- The template class defines the method **onselect()**.

First, let us examine the layout graph, which is provided by the following method in the template class:

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
    var myLayoutGraph = {};

    //The standard technique is to have a switch/case construct based on the key argument.
    //In this case, the layout is not key-specific layout, so there is no need to branch.

    myLayoutGraph = {
        children: [
            { type: '$header', $content: 'Zen Mojo Tutorial 3'},
            { type: '$p'},
            { type: '$button', $content:'Show Person 1', key:'showPerson1'},
            { type: '$p'},
            { type: '$button', $content:'Show Person 2', key:'showPerson2'},
            { type: '$p'},

            { type: '$div', key:'person1',
              children:[
                { type: '$p', title:'Name', $content:'=[person1Name]' },
                { type: '$p', title:'Birth Date', $content:'=[person1DOB]' }
              ]},

            { type: '$div', key:'person2', hidden:true,
              children:[
                { type: '$p', title:'Name', $content:'=[person2Name]' },
                { type: '$p', title:'Birth Date', $content:'=[person2DOB]' }
              ]},

        ]

    }

    return myLayoutGraph;
}
```

Notice that this layout graph includes two `$button` layout objects and specifies a key value for each one. After the buttons, the layout graph has two `$div` objects, one of which is initially visible and one of which is hidden.

Now let us examine the **onselect()** method. Zen Mojo automatically calls this method when the user selects an item on the page. This method is defined as follows:

```
ClientMethod onselect(key, value, docViewId) [ Language = javascript ]
{
    console.log('in '+docViewId+ ' select: ' + key + ' value: ' + value);
    if (docViewId=='mainView') {
        var person1=zen('mainView').getItemByKey('person1');
        var person2=zen('mainView').getItemByKey('person2');

        if (key=='showPerson1') {
            person1.$show();
            person2.$hide();
        } else if (key=='showPerson2') {
            person1.$hide();
            person2.$show();
        }
    }
}
```

The method receives, as an argument, the id of the documentView where the select event occurred; this enables us to customize the behavior in each documentView. In this case, there is only one documentView. The other arguments are the key and value of the selected item.

The `console.log` line writes a line to the JavaScript console. In this case, the line just provides information about the values passed to the **onselect()** method.

The method uses the syntax `zen()` to access the documentView component and then uses the method **getItemByKey()** to access specific items in the layout graph.

Depending on the selected key, the method then controls which parts of the layout graph are visible. Note that the HTML5 helper plugin provides the `$show()` and `$hide()` methods. Other helper plugins provide other tools. For details on each plugin, see [Using Zen Mojo Plugins](#).

2.3.3 Exercises

- Display the JavaScript console for the browser that you are using. The details depend upon the browser, but this console is generally found with other developer tools. Click buttons on the Zen Mojo page and notice the corresponding log entries in this console.
- Add another button to the display. For this button, display both the two `$div` objects.
- Add another button, specifically to display the third person from `Sample.Person`.

3

Background Information and Tasks

As background for the following chapters, this section describes basic techniques that you can use when you define client methods in Zen Mojo. It also contains a section on naming conventions to consider before starting to create Zen Mojo applications. It discusses the following topics:

- [General-purpose variables and functions](#)
- [How to stash values in the client](#)
- [How to convert a JSON object to a string](#)
- [Naming conventions](#)

3.1 General-Purpose Client Variables and Functions

For reference, this section lists some key general-purpose variables and functions that you can use within a Zen Mojo client method:

- `this` — This standard JavaScript variable represents the current instance of the template or page, depending on where you use the variable.

That is, if you use `this` within a template method, it returns the template instance. If you use `this` within a page method, it returns the page instance.

- `zenPage` — This InterSystems variable represents the current page object.
- `zen(id)` — This InterSystems function returns a reference to the `documentView` or other component that has the given `id`.

Note that this function cannot be used to access layout objects, which are not components. To access layout objects, you use methods of the `documentView` component; see “[Interacting with Layout Objects](#),” later in this book.

- `zenGet(property_or_array_item, optional_default_value)` — This InterSystems function examines the given object property or array item and returns its value, an optional default value, or an empty string, depending on the scenario.

Specifically, if the property or array item is defined, the function returns its value. If the item is not defined, and if the second argument *is* specified, this function returns the value specified by the second argument. If the item is not defined, and if the second argument is *not* specified, this function returns an empty string (`' '`).

For example, consider the following code:

```
if (myobj.prop4 == undefined) {  
    var returnval = 'No information available';  
}
```

That code is equivalent to the following:

```
var returnval=zenGet(myobj.prop4,'No information available');
```

- `console.log(argument)` — This standard JavaScript function generates an entry in the web console log.
- `alert(argument)` — This standard JavaScript function generates a popup that displays the given message. InterSystems recommends that you use this function only for diagnostic purposes.
- `zenPage.showMessage(argument)` — This InterSystems function generates a popup message that displays the given message. InterSystems recommends that you use this function for messages intended for the user.

For other variables and functions provided by InterSystems, see “Client Side Functions, Variables, and Objects” in *Developing Zen Applications*. For additional standard JavaScript options, see any suitable JavaScript documentation.

3.2 Stashing Values

In some cases, you may want to temporarily save client-side data so that you can use it within multiple layouts. For example, it may be necessary to collect data from the user via multiple layouts before submitting that to the server. Or you might want to display data from one layout within a different layout. Or you might need to keep track of an internal identifier. In such cases, you can *stash* values and then later use them. To stash a value, save it in a property of the page instance or the template instance, depending on your preference.

- To save a value in a property of the page instance, set a property of the `zenPage` variable. The `zenPage` variable is available in client methods in both the page and template classes.

Start the property name with an underscore character (`_`), which ensures that this property is defined only on the client. To avoid collision with internally used property names, consider starting your property names with `_myApp` or a similar short string.

- To save a value in a property of the template instance, set properties of the `this` variable in a client method in a template class.

Note that you can also use `this` in a page class. In that context, `this` refers to the page instance. To avoid confusion, it is best to choose a single context for the stashed values (page or template) and then use that context consistently.

When you no longer need the stashed value, use `delete` to remove it. For example:

```
delete this._MyNewValues;
```

3.3 Converting a JSON Object to a String

Zen Mojo packages all communication between the client and server as JSON objects, as follows:

- To retrieve data from the server, you (indirectly) call the template method `%OnGetJSONContent()`, which returns a Zen proxy object as output. Zen Mojo uses that to create an equivalent JSON object and then sends the JSON object to the client.
- To submit data to the server, you call the `submitData()` method of the page. As input, you must use a JSON object that contains the data.

When you stash values, however, it is important to remember that you can stash only single JavaScript values (not objects). If you need to stash an entire JSON object, first use the utility method `JSON.stringify()`, which takes one argument, the JSON object, and returns a string. Then stash the returned string.

The `JSON.stringify()` utility method is available in client methods.

3.4 Establishing Naming Conventions

It is worthwhile to establish naming conventions to avoid confusion. This section discusses the following topics:

- [Class names](#)
- [Method names](#)
- [Content objects and JSON providers](#)
- [Keys](#)

Zen Mojo programs are case-sensitive. Remember that `a` is not the same as `A`.

3.4.1 Class Names

InterSystems strongly recommends that you do not create any classes in a package called `ZEN.Component` using any combination of uppercase and lowercase characters. If you create a package called `ZEN.Component`, that interferes with the way that Zen generates client-side code.

Also, note that because a Zen Mojo application can be easily extended to use multiple templates, you might want to place your template classes in a subpackage.

Important: Because these classes are automatically projected to XML, there is an additional consideration. If multiple classes have the same short class name (that is, the class name without the package), be sure that the *NAMESPACE* parameter is unique for each such class.

This rule is a consequence of the fact that the short class name becomes the name of a global XML element, and those elements must be unique in an XML namespace. In Caché, each XML-enabled class must have a unique combination of the short class name and the *NAMESPACE* parameter.

3.4.2 Method Names

InterSystems classes use case to distinguish between client and server methods. You might find it helpful to follow these conventions as well.

Kind of Method	Convention for Method Name	Example
Client method (that is, a method written in JavaScript)	Start with a lowercase letter and use an initial capital letter for each successive word (in-word capitalization).	<code>myMethodName ()</code>
Server method (that is, a method written in ObjectScript, Caché Basic, or Caché MVBasic)	Start with an uppercase letter and use in-word capitalization. (Or start with a <code>%</code> character, followed by an uppercase letter.)	<code>MyMethodName</code> or <code>%MyMethodName</code>

3.4.3 Content Objects and JSON Providers

Each `documentView` is specified by two content objects — a data object and a layout graph, as described in “[The Template System](#),” earlier in this book. You must use the names of these objects consistently in several places in the code, so it is useful to have a convention for them. One approach is as follows:

- Use the name `document` for the data object of your primary `documentView`
- Use the name `layout` for the layout graph of your primary `documentView`
- Use more specific names for the content objects of other `documentView` components

Another approach is as follows:

- Use the name `componentidData` for the data object of the `documentView` whose id is `componentid`
- Use the name `componentidLayout` for the layout graph of the `documentView` whose id is `componentid`

Note that (via the `PROVIDERLIST` parameter) the page class includes a JSON provider for each data object. For this reason, the names of the data objects also become names of the JSON providers. The terms *data object* and *JSON provider* are sometimes used interchangeably, although this book simply uses *data object*.

3.4.4 Keys

You can associate keys with many of the layout objects, and your application can end up with a large number of keys. It is particularly useful to establish a naming convention for them. A simple system would be as follows:

- Use names that make the code easier to read.
- Use a verb for the name of a key in a control.
- Use a noun for the name of a key in any other scenario.
- Adopt a hierarchical (or semi-hierarchical) set of names for related keys. For example, you might use the key name `accounts` for a layout element that displays a table of accounts, and then use the key name `account-detail` for a layout element that displays details for one account.

Important: The name of a key cannot include a colon (`:`) character.

4

Summary of Callbacks and Event Handlers

This chapter summarizes all the event handlers and callbacks that apply to a Zen Mojo page. To define the behavior of the page, you must implement some or all of these items. This chapter discusses the following topics:

- [The ongetdata and ongetlayout Callback Attributes](#)
- [Other Callback Attributes](#)
- [Event Handlers](#)

4.1 The ongetdata and ongetlayout Callback Attributes

The two most basic callback attributes of a `documentView` are `ongetdata` and `ongetlayout`, which were introduced in earlier parts of this book.

4.1.1 When These Callbacks Are Used

When a Zen Mojo page is displayed or when the method `updateLayout()` is executed, Zen Mojo examines both `ongetdata` and `ongetlayout`, and executes the code specified in these callbacks. `ongetdata` is executed first, followed by `ongetlayout` (but see the following notes).

Typically, these callbacks invoke the page method `getContent()`.

Within both callbacks, the variables `key` and `value` are available. The variable `key` equals the key of the selected layout object, if any, and the variable `value` equals the value of the selected layout object, if any.

Furthermore, within `ongetlayout`, it is possible to access the data object returned by `ongetdata` (but see the following notes).

Notes:

- The `ongetdata` callback is *not* invoked if the `ongetlayout` returns a layout graph that contains the `sourceData` object.
- The `ongetdata` callback is *not* invoked if the cache contains data for the given key.

Zen Mojo caches both data objects and layout graphs. You can invalidate the cache for a given `documentView`. To do so, you use the `invalidate()` method of the `documentView`. For example:

```
var mainView = zen('mainView');
mainView.invalidate();
```

- It is also possible to invoke *only* the `ongetData` callback. To do this, you use the `getSourceData()` method of the `documentView`. For example:

```
var mainView = zen('mainView');
var data = mainView.getSourceData();
```

The `getSourceData()` method returns the source data for the `documentView` instance. This may come from the layout graph (if it defines a `sourceData` property), the data cache for the current level, or from the `ongetData` callback.

4.1.2 Specifying `ongetData` or `ongetlayout`

To define either of these callback attributes, use the following general procedure:

- Specify a value for the attribute within the definition of a `documentView`.

In most cases, the value is a JavaScript expression that invokes the `getContent()` method of the page, as in the following example:

```
<mojo:documentView id="mainView"
ongetlayout = "return zenPage.getContent('layout',key,criteria);"
ongetData = "return zenPage.getContent('data',key,criteria);" >
...
</mojo:documentView>
```

- Implement the `onGetContent()` and `%OnGetJSONContent()` methods in the associated template class. The following sections describe these methods formally.

For an overview of `onGetContent()` and `%OnGetJSONContent()`, see “[The Template System](#),” earlier in this book.

4.1.3 Details for `getContent()`

The `ongetData` and `ongetlayout` callback attributes typically call the `getContent()` method of the page. This section provides detailed information on this method.

(Note that it can also be useful to call this method in other scenarios. For example, see the `onselect()` method in the sample `ZMdemo.LoadAsync.baseTemplate`.)

The `getContent()` method returns a *content object* (a data object or a layout graph). This method has the following arguments:

```
getContent(providerName, key, criteria, force, notify)
```

Where:

- *providerName* specifies the name of the content object to get (see “[Content Objects](#)” in the chapter “[Zen Mojo Templates](#)”). Each `documentView` can have its own content objects.
- *key* is a short, unique name that indicates the specific version of the content. .
- *criteria* contains any additional information to pass to the content template. This can be a search string entered by the user or other criteria used internally in your code.
- *force* specifies whether to reload content from the server.
- *notify* is either null or a function. If this argument is a function, the page sends the data *asynchronously* to the server (rather than synchronously). Then, when the data has been sent, the page executes the function named by this argument.

For an example, see the `onselect()` method in the sample `ZMdemo.LoadAsync.baseTemplate`.

The `getContent()` method does the following:

1. It invokes the `onGetContent()` method of the associated template class. If that method returns content, the system returns that content.

To find the initial template class, Zen Mojo examines the *TEMPLATECLASS* parameter of the page class.

2. If **onGetContent()** returns null, the system checks to see whether the page cache contains any content. If so, the system returns that content.
3. If there is no cached content, the system invokes the **%OnGetJSONContent()** method of the associated template class, uses the result to create the content, and then returns that content.

For a diagram of this activity, see “[The Template System](#),” earlier in this book.

4.1.4 Defining onGetContent()

Each Zen Mojo template class should define the **onGetContent()** method. This method is a dispatcher that returns a content object. This method has the following signature:

```
ClientMethod onGetContent(providerName, key, criteria) [ Language = javascript ]
```

Where:

- *providerName* specifies the content object to return.
- *key* specifies the key to use when retrieving the content object.
- *criteria* is any additional criteria to use when retrieving the content object.

In this method, the standard technique is to use an outer branching construct that uses the *providerName* argument.

- If *providerName* corresponds to a layout graph, **onGetContent()** should call a suitable layout method. See the chapter “[Defining Layout Methods](#).”

Also, **onGetContent()** should pass the *key* and *criteria* arguments to that method.

- If *providerName* corresponds to a data object, **onGetContent()** should return null.

In this case, Zen Mojo gets the object by calling **%OnGetJSONContent()** in this class (see the next subsection, “[Defining %OnGetJSONContent\(\)](#)”).

The following shows an example. In this scenario, *mainViewLayout* and *leftViewLayout* are the keys for layout graphs. Notice that the switch construct contains branches *only* for these layout graphs. The method thus returns null when it is invoked with the key for a data object.

```
ClientMethod onGetContent(providerName, key, criteria) [ Language = javascript ]
{
    var content = null;

    // dispatch to convenient methods
    // if content is null, then the %OnGetJSONContent method will be called

    switch(providerName) {
    case 'mainViewLayout':
        content = this.myGetMainViewLayout(key,criteria);
        break;
    case 'leftViewLayout':
        content = this.myGetLeftViewLayout(key,criteria);
        break;
    }
    return content;
}
```

4.1.5 Defining %OnGetJSONContent()

In most cases, each Zen Mojo template class should define the **%OnGetJSONContent()** method. This method should return the requested data object as an output parameter. This method has the following signature:

```
ClassMethod %OnGetJSONContent(pProviderName As %String,
                             pKey As %String,
                             ByRef pParms,
                             Output pObject As %RegisteredObject,
                             pCriteria As %RegisteredObject,
                             pLoad As %Boolean = 0) As %Status
```

Where:

- *pProviderName* specifies the data object to return.
- *pKey* specifies the key to use when retrieving the data object.
- *pParms* is currently unused.
- *pObject* is the instance of %ZEN.ProxyObject that contains the data to return to the client.

The class %ZEN.ProxyObject does not have any predefined properties but instead is a special-purpose container object. For an introduction, see “Zen Proxy Objects” in *Developing Zen Applications*.

- *pCriteria* is any additional criteria sent from the client.
- *pLoad* specifies whether to load the data. This argument is true when the page is first served.

In this method, the standard technique is to use an outer branching construct that uses the *pProviderName* argument.

Then, within any given *pProviderName* branch, use an inner branching construct that examines the *pKey* argument and that creates a different version of *pObject* depending on that argument. The following shows an example where *pProviderName* has only one possible value, but *pKey* has multiple possible values:

```
ClassMethod %OnGetJSONContent(pProviderName As %String, pKey As %String, ByRef pParms,
                             Output pObject As %RegisteredObject, pCriteria As %RegisteredObject,
                             pLoad As %Boolean = 0) As %Status
{
    if (pProviderName = "mainViewData") {
        //nothing to return for MyButton1 so there is no branch for that
        if (pKey = "MyButton2") {
            //create proxy object that the Zen Mojo will convert to a JSON string
            //and send to the client
            set pObject = ##class(%ZEN.proxyObject).%New()
            set random=$R(100)+1
            set tPerson = ##class(Sample.Person).%OpenId(random)
            set pObject.personName=tPerson.Name
            set pObject.personDOB=$zdate(tPerson.DOB,3)
        }
        elseif (pKey = "MyButton3") {
            //create proxy object that the Zen Mojo will convert to a JSON string
            //and send to the client
            set pObject = ##class(%ZEN.proxyObject).%New()
            set tList = ##class(%Library.ListOfObjects).%New()
            set pObject.children = tList
            For i=1:1:5 {
                // we could retrieve the same company more than once this way
                // but this is a demo so that doesn't matter
                set tNumber=$RANDOM(20)+1
                set tCompany = ##class(Sample.Company).%OpenId(tNumber)
                set tCompanyData = ##class(%ZEN.proxyObject).%New()
                set tCompanyData.companyName = tCompany.Name
                set tCompanyData.companyMission = tCompany.Mission
                set tCompanyData.companyRevenue = tCompany.Revenue
                Do tList.Insert(tCompanyData)
            }
            set pObject.rowCount = tList.Count()
        }
    }
    ; additional pProviderName branches would go here
    quit $$$OK
}
```

4.2 Other Callback Attributes

Zen Mojo provides several additional callback attributes for a `documentView`:

- `onload`
Used to perform any setup logic. For example, you could set the initial document and layout keys based on `localStorage`. This callback is invoked only before the first time the `documentView` is rendered.
- `onrender`
Provides notification that the `documentView` is being rendered.
- `onresolvemethod`
Enables you to define methods that you can use in layout graphs (see “[Invoking Template Methods in a Layout Graph](#)” in the chapter “[Defining Layout Methods](#)”).
- `onresolvepluginconflicts`
Provides information about plugin conflicts if any conflicts have occurred during plugin registration. For details, see “[Detecting and Resolving Plugin Conflicts](#)” in the book *Using Zen Mojo Plugins*.

To use these callback attributes, specify a value for the attribute for a `documentView` used on the page. In most cases, the value is an expression that invokes a method you have implemented in the template class. For example:

```
onresolvemethod="return zenPage.getTemplate().resolve(context,which);"
```

where `getTemplate()` is a method of `basePage` that returns a reference to the instance of the current template associated with the page.

Alternately, you could invoke it indirectly by calling a simple wrapper method of the same name in the page class. If necessary, you can implement the callback method itself in the page class, but best practice is to put all application logic in the template class.

4.3 Event Handlers

When an event occurs on the page, Zen Mojo automatically invokes an event handler in the associated template class. The Zen Mojo *event handlers* are the `onselect()`, `onchange()`, and `onevent()` methods, which have no behavior by default. You define these methods in your template class, in order to make your page interactive.

4.3.1 Where Events Are Supported

In each plugin, events are supported (by default) in the layout objects that are typically used for user interaction. This means that layout objects that represent buttons, menus, sliders, and other controls support events; that is, the plugin inserts event handling for those objects. Layout objects that represent static elements do not support events.

You can suppress the insertion of event handling for a given layout object. To do so, specify one or more of the following attributes of that layout object, as needed:

- `$ignoreSelect` — if you specify this attribute as `true`, the layout object does not have an associated `onselect` handler (`onselect()` is ignored). The default is `false`.

- `$ignoreChange` — If you specify this attribute as true, the layout object does not have an associated onchange handler (**onchange()** is ignored). The default is false, and any layout object that represents a control has an associated onchange handler.
- `$ignoreEvent` — If you specify this attribute as true, the layout object does not have an associated generic event handler (**onevent()** is ignored). The default is false.

4.3.2 Implementing Event Handlers

Define some or all of the following event handler methods in your template class. In a typical implementation, these methods cause changes to the page or `documentView` instance, depending on your needs.

onselect()

```
ClientMethod onselect(key, value, docViewId) [ Language = javascript ]
```

Defines how the page behaves when a user selects a layout object in a `documentView`. The arguments are as follows:

- *key* — key of the selected layout object.
- *value* — value of that layout object, if any.
- *docViewId* — id of the `documentView`.

onchange()

```
ClientMethod onchange(key, value, final, docViewId) [ Language = javascript ]
```

Defines how the page behaves when a user changes the value of a layout object in a `documentView`. The arguments are as follows:

- *key* — key of the changed layout object.
- *value* — new value of the layout object.
- *final* — indicates whether the value passed is the final value.
- *docViewId* — id of the `documentView`.

onevent()

```
ClientMethod onevent(eventType, key, value, docViewId) [ Language = javascript ]
```

Defines how the page behaves when another type of event occurs within a `documentView` (an event other than select or change). The arguments are as follows:

- *evtType* — type of the event. For information on event types, see “zenEvent” in *Developing Zen Applications*.
- *key* — key of the layout object that triggered the event.
- *value* — value of that object, if any.
- *docViewId* — id of the `documentView`.

5

Creating the Classes

When you create a Zen Mojo application, most of the work is the task of defining the template class or classes. So that you can start that work as quickly as possible, this chapter describes the prerequisite work of defining a set of interrelated classes as a starting point. It discusses the following topics:

- [Choosing plugins and downloading files](#)
- [Using the Zen Mojo wizard](#)
- [Defining the classes manually](#)
- [Creating a pageContents definition in the page class](#)
- [Next steps](#)

5.1 Choosing Plugins and Downloading Files

Early in development, you should decide which plugins you will use. In some cases, you will need to download files for use by a plugin. For a list of plugins and files that they need, see [Using Zen Mojo Plugins](#).

5.2 Using the Zen Mojo Wizard

Studio provides a convenient wizard that you can use as a starting point for this set of classes. The wizard generates a page class, a template class, and (optionally) an application class. To use this wizard:

1. Click **File > New**, click **Zen**, and click **Zen Mojo Site**.
2. Click **OK**.
3. Specify values for some or all of the following options:
 - **Package Name** — (Required) Specify the name of the package to contain the classes.
 - **Page Class Name** — (Required) Specify the short class name for the page class.
 - **Page Name** — Specify the logical name of this page within its application.
 - **Page Manager** — (Required) Select a page manager.

- **Chosen Helper Plugins** — Select the helper plugins to use in this page class. To select a helper plugin, click an item in the **Available Helper Plugins** list. (To remove a helper plugin from the **Chosen Helper Plugins**, click it in that list.)
- **Template Class Name** — (Required) Specify the short class name for the template class. The class is created in the package given by **Package Name**.
- **Application Class Name** — Specify the short class name for the application class (if any). The class, if specified, is created in the package given by **Package Name**.
- **Application Name** — Specify the logical name of this application.
- **Domain** — Specify a short string. This option specifies the localization domain of any localizable strings generated by this class.

Other than the class names, you can change all the details creating the classes.

4. Click **Finish**.

Studio then generates a Zen Mojo page class, template class, and (if you specified **Application Class Name**) application class, all in the package that you specified for **Package Name**.

The page class contains preliminary parameter definitions, a preliminary pageContents XData block, and an empty XData Style block. It also specifies the *JSINCLUDES* and *CSSINCLUDES* class parameters, based on the helper plugins you selected.

The template class contains preliminary definitions of `%OnGetJSONContent()`, `onGetContent()`, and other methods.

The application class (if generated) contains preliminary parameter definitions and an empty XData Style block.

The next step is to examine the page class and modify its pageContents XData block as needed. See “[Creating a Basic pageContents Definition](#),” later in this chapter.

5.3 Creating Zen Mojo Classes Manually

As an alternative to using the Zen Mojo wizard, you can create the initial [application class](#), [page class](#), and [template class](#) manually. This section describes the requirements.

5.3.1 Defining the Zen Mojo Application Class

A Zen Mojo application class is optional.

To define a Zen Mojo application class, define a class that extends `%ZEN.Mojo.baseApplication` and optionally include a Style XData block in this class. For details, see the chapter “[Using Style Sheets](#).”

For additional options for the application class, see the class references for `%ZEN.Mojo.baseApplication`.

5.3.2 Defining a Zen Mojo Page Class

To create a basic Zen Mojo page class, create a class that extends `%ZEN.Mojo.basePage`. In your class:

- Optionally define the *APPLICATION* parameter. For the value, specify the name of the matching application class, from the [previous section](#). For example:

```
Parameter APPLICATION = "ZMdemo.dojo.Application";
```

- Define the *TEMPLATECLASS* parameter. For the value, specify the name of a Zen Mojo template class, which is discussed [later in this chapter](#). For example:

```
Parameter TEMPLATECLASS = "ZMdemo.dojo.baseTemplate";
```

- Define the *PROVIDERLIST* parameter. For the value, specify a comma-separated list of the names of the content objects needed for this page. This list includes the major branches of *%OnGetJSONContent()* in your template class (see the following subsection “[Defining a Template Class](#)”). For example:

```
Parameter PROVIDERLIST = "data,layout";
```

- Override the inherited pageContents XData block and create a basic definition of the page (see the following section, “[Creating a Basic pageContents Definition](#)”).

In this definition, you typically use one or more plugins. Make a note of the plugins you use.

- Update the *JSINCLUDES* parameter and *CSSINCLUDES* (or *CSS3INCLUDES*) parameters as needed by the plugins. For details, see [Using Zen Mojo Plugins](#).

In most cases, these parameters specify external JavaScript libraries and CSS style sheets, respectively, to be loaded into the page instance. For the Google maps helper plugin, however, *JSINCLUDES* indicates the URL of the Google Maps API. The libraries are loaded in the order in which they are listed, from left to right; this is important if one library depends upon another.

Note that not all plugins require these parameters.

- Define the *DOMAIN* parameter. This parameter specifies the localization domain of any localizable strings generated by this class. For example:

```
Parameter DOMAIN = "MyZenMojoApp";
```

This parameter is not required, but InterSystems recommends that you always specify it.

5.3.3 Defining a Template Class

To create a template class, create a class that extends *%ZEN.Mojo.Component.contentTemplate*. In your class, do the following as a starting point:

- Define the *NAMESPACE* parameter. This parameter specifies the XML namespace to which this template belongs. Each Zen Mojo template should have a unique combination of *NAMESPACE* and short class name.

For example:

```
Parameter NAMESPACE = "http://www.intersystems.com/zen/mojo/demo/dojo";
```

- Define the *DOMAIN* parameter. This parameter specifies the localization domain of any localizable strings generated by this class. For example:

```
Parameter DOMAIN = "MyZenMojoApp";
```

This parameter is not required, but InterSystems recommends that you always specify it.

- Create a preliminary version of the *onGetContent()* method, which acts as the dispatcher for all requests for content objects. See “[Defining onGetContent\(\)](#),” in the previous chapter.
- Create a preliminary version of the *%OnGetJSONContent()*, which returns data objects from the server. See “[Defining %OnGetJSONContent\(\)](#),” in the previous chapter.
- Ensure that in the corresponding [page class](#), the *PROVIDERLIST* parameter lists the name of each possible data object. The server-side page object cannot return data objects unless the *PROVIDERLIST* parameter lists those objects.

5.4 Creating a Basic pageContents Definition

In the [page class](#), the pageContents XData block defines the contents of the primary area of the page, as described in “[Parts of a Zen Mojo Page](#),” earlier in this book. The following shows the skeletal structure of this XData block:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen" xmlns:mojo="http://www.intersystems.com/zen/mojo"
layout="none">
  your contents here
</pane>
}
```

In place of *your contents here*, include one or more `<mojo:documentView>` elements. For each of these, include a page manager plugin and all needed helper plugins (see the next section, “[Registering Plugins](#)”). Also specify the `id` attribute of each `documentView`, as well as either or both of the `ongetlayout` and `ongetdata` XML attributes, depending on your needs. For `ongetlayout` and `ongetdata`, use values like the following:

```
return zenPage.getContent('contentObject',key,criteria);
```

Note that `getContent()` is a built-in page method and *contentObject* is the name of a content object provided, ultimately, by the template.

The following shows an example:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen"
xmlns:mojo="http://www.intersystems.com/zen/mojo" layout="none">
  <mojo:documentView id="mainView" developerMode="true"
    ongetlayout = "return zenPage.getContent('mainViewLayout',key,criteria);">
    <mojo:mojoDefaultPageManager>
      <mojo:HTML5Helper/>
    </mojo:mojoDefaultPageManager>
  </mojo:documentView>
</pane>
}
```

In this example, child elements are indented for readability. It is not necessary for you to do the same.

5.4.1 Registering Plugins

You should register one or more plugins for use in each `documentView`. To do so:

- Include one page manager plugin as a child element of `<mojo:documentView>`.
- Include one or more helper plugins as child elements of the page manager plugin.

The following shows the general structure (with extra indentation for clarity; you do not need to include this indentation):

```
<mojo:documentView id="mainView" other attributes...>
  <mojo:somePageManager>
    <mojo:someHelper/>
    <mojo:anotherHelper/>
    <mojo:yetAnotherHelper/>
  </mojo:somePageManager>
</mojo:documentView>
```

For details, see [Using Zen Mojo Plugins](#).

If you use custom plugins, it is important to consider the order in which you list the helper plugins, because of the possibility of plugin conflict. A *plugin conflict* occurs if a single `documentView` uses multiple helper plugins and those plugins have layout objects with the same name. For all plugins provided by InterSystems, each layout object has a unique name, but custom plugins could potentially have layout objects with the same names as InterSystems layout objects. This is not an

error condition, but rather a scenario that requires special handling. For more information, see “[Detecting and Resolving Plugin Conflicts](#)” in *Using Zen Mojo Plugins*.

5.4.2 Using Multiple documentViews

Depending on the plugins you use, a page can contain multiple documentViews that can affect each other. For example, you could have two documentViews, side by side. The left documentView could display options that control what is shown in the right documentView.

For some plugins, only one documentView is supported. For details, see *Using Zen Mojo Plugins*. In general, if a page manager plugin is intended for use on mobile devices, that plugin uses the entire page and does not support multiple documentViews.

5.5 Next Steps

Now that you have created a set of Zen Mojo classes to use as a starting point, do the following to create your Zen Mojo application:

- Read the relevant chapters of *Using Zen Mojo Plugins* to learn about additional implementation steps that are required or recommended for the plugins you are using.

Important: Depending on the page manager plugin you are using, you may need to implement additional methods in the page class to make your page contents visible.

- Create layout methods. See “[Defining Layout Methods](#).”
- Define event handlers and other options to specify the behavior of the page. The chapter “[Summary of Callbacks and Event Handlers](#)” summarizes the options. For information on options you can use *within* the event handlers and callbacks, see the following chapters:
 - “[Using Keys](#)”
 - “[Interacting with Layout Objects](#)”
 - “[Submitting Data to the Server](#)”
- Define and apply style information as needed. See the chapter “[Using Style Sheets](#).”

Later chapters in this book discuss additional topics.

6

Defining Layout Methods

This chapter describes how to create layout methods, which are methods in the template class that describe the layout of your page. This chapter discusses the following topics:

- [Overview of Layout Methods and Layout Graphs](#)
- [Referring to the Current Data Object](#)
- [Invoking Template Methods in a Layout Graph](#)
- [Specifying Style Attributes in Layout Graphs](#)
- [Using the `sourceData` Property in a Layout Graph](#)
- [Using Stashed Values within `sourceData`](#)

6.1 Overview of Layout Methods and Layout Graphs

As noted earlier in this book, a *layout method* is a method, in a template class, that returns a layout graph. The `onGetContent()` method of the template class is responsible for invoking the layout methods in the same class. The `onGetContent()` method can pass two arguments to any layout method:

- *key* specifies the key to use when retrieving the layout graph.
- *criteria* is any additional criteria to use when retrieving the layout graph.

Formally, a *layout graph* is a JSON or JavaScript object that meets the following requirements:

- It must have a `children` property.
- The `children` property must be an array of *layout objects*.

Each layout object represents an item within the `documentView`. The `documentView` generally displays these items in the same order that they are listed.

- Each layout object must have a `type` property, which specifies the overall appearance and behavior of this object

You can specify additional properties for the object to control the appearance and behavior of the corresponding part of the page. Each object has specific properties. For details, see the plugin reference documentation (see “[Plugin Documentation](#)” in the book *Using Zen Mojo Plugins*).

Typically you specify properties that determine the value of the layout object, captions or titles to display, and style information.

Most layout objects support events, if the object has a property named `key`. You also use this property to refer to that object from within a method (see the chapter “[Interacting with Layout Objects](#)”). Conversely, do not specify the `key` property if it is not needed.

- The value of any property must be either a valid JavaScript expression, a reference to the current data object, or (in special cases) to variables provided by the layout object.

Zen Mojo provides special syntax to refer to the current data object or to variables provided by the layout object (see the next section, “[Referring to the Current Data Object](#)”).

- In the layout graph, you can use the Zen macro `$$$Text()` for localization. The best practice is to use this macro around all user-visible strings within the layout graph.

Note that the strings are localizable only if the page and template classes both define the `DOMAIN` parameter. For details on localization, see “Zen Localization” in *Developing Zen Applications*.

- The layout graph can have a property named `sourceData`, which is treated specially. See “[Using the sourceData Property in a Layout Graph](#)” later in this chapter.

The following simple example returns a static layout graph (which does not refer to the data object, to any variables, or to the special `sourceData` object). This example uses layout objects defined in the HTML5 helper plugin (see “[Helper Plugin for HTML5](#)” in the book *Using Zen Mojo Plugins*):

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
  var myLayoutGraph = {};
  //The standard technique is to have a switch/case construct that creates
  //branches based on the key argument;
  //In this case only one key value is possible, so there is no need to branch

  myLayoutGraph = {
    children: [
      { type: '$header', $content: $$$Text('Tutorial 1 - Sample Zen Mojo Page')},
      { type: '$p',      $content: $$$Text('This page displays some simple text.')},
      { type: '$p',      $content: $$$Text('Here is another paragraph.') },
    ]
  }

  return myLayoutGraph;
}
```

This method returns an object (called `myLayoutGraph` within this method), which has a property named `children`. The `children` property is an array that contains three objects. The first object is of type `$header`, and the second and third objects are of type `$p`. Note the use of the `$$$Text()` macro.

6.2 Referring to the Current Data Object

A layout graph can (and typically does) refer to properties in the data object. To refer to the value of a property (*propertyname*) of the data object, use `'=[propertyname]'`

To refer to a property of a property, you can use dot syntax. For example: `'=[propertyname.subprop1]'`

Similarly, if a property is an array, you can use square brackets and the array item number. For example:

`'=[arrayprop[3]]'` or `'=[arrayprop[3].subprop]'`

Within a layout graph, when Zen Mojo finds a layout object property with a value of the form `'=[propertyname]'`, it tries to find the given property (*propertyname*) in the current data object. If it finds the property, Zen Mojo inserts its value into the corresponding part of the layout graph. If it does not find the property, Zen Mojo uses the null value instead. Zen Mojo does not throw an error.

Note: This syntax applies specifically within a layout graph and is not available elsewhere.

Suppose that the data object for a given `documentView` has the following properties:

- `name` (a string)
- `dob` (a string)
- `homeaddress` (an object that has the properties `street`, `city`, and `postalcode`)
- `favoritecolors` (an array of strings)

The following example layout method displays data for this object. This example also uses layout objects from the HTML5 helper plugin (see “[Helper Plugin for HTML5](#)” in the book *Using Zen Mojo Plugins*).

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
  var myLayoutGraph = {};
  myLayoutGraph = {
    children: [
      { type: '$header', $content: $$$Text('Data Object Demo Template')},
      { type: '$p', title:'Name', $content:'=[name]' },
      { type: '$p', title:'Birth Date', $content:'=[dob]' },
      { type: '$p', title:'Home City', $content:'=[homeaddress.city]' },
      { type: '$p', title:'First Favorite Color', $content:'=[favoritecolors[1]]' },
      { type: '$p', title:'Number of Favorite Colors', $content:'=[favoritecolors.length]' },
    ]
  }
  return myLayoutGraph;
}
```

Note: `'=[$variable]` **Syntax**

A few layout objects support the syntax `'=[$variable]`, where *\$variable* is a variable provided by the layout object. For example, the `$loop` layout object defines variable *\$loopNumber* for each item in a loop. See “[The \\$loop Layout Object](#)” in the book *Using Zen Mojo Plugins* for details and extended examples.

6.3 Invoking Template Methods in a Layout Graph

You can invoke a method from within a layout object using the syntax:

```
myLayoutProperty: '=[ $method.methodname ]'
```

where *methodname* is a string that will be passed to a callback method defined in the template class. When the callback is invoked, it will perform the operation defined for *methodname* and return an appropriate value.

Use the following general procedure to implement the callback:

- *Specify callback attribute* `onresolvemethod`

In your page class, add the `onresolvemethod` attribute to the `documentView` element of the **pageContents** XData block. For example:

```
onresolvemethod="return zenPage.getTemplate().resolve(context,which);"
```

where **resolve()** is the callback method (implemented in the template class) and basePage method **getTemplate()** returns a reference to the current template. (See “[Summary of Callbacks and Event Handlers](#)” for general information on `documentView` callbacks).

- *Add the callback method to the template class*

The callback method must have a signature of the form `methodname(context, which)`. When invoked, it receives the following arguments:

- *context* — is an object that contains properties of the layout object that invoked the callback.

- *which* — is string *methodname* from the '`=[$method.methodname]`' expression

The callback method typically implements a `switch` based on the value of *which* to determine the appropriate action.

- *Invoke the callback in your layout graph*

The following `$p` layout object assigns a value to property `myValue`, and invokes the callback to define property `$content`:

```
{ type: '$p', myValue: '345' , $content: '=[ $method.readValue ]' }
```

In this example, callback argument *which* will be string `'readValue'`, and argument *context* will hold a variable *\$instance* that resolves to the original layout object (including property `myValue`).

The following example describes a simple `onresolvemethod` callback and demonstrates how it is used.

Example: Using the `resolve()` callback with simple values and loops

This example implements a callback method named `resolve()`. The `onresolvemethod` attribute is defined as follows:

```
onresolvemethod="return zenPage.getTemplate().resolve(context,which);"
```

The `resolve()` method is implemented in the template class:

```
ClientMethod resolve(context, which) [ Language = javascript ] {
  switch (which) {
    case 'readValue': return 'myValue = \'' + context.$instance.myValue + '\'';
    case 'readLoop':  return 'myLoop('+context.$loopNumber+') = \'' + context.$loopValue +
  '\'';
  }
}
```

The `resolve()` method deals with two different types of *context* object:

- `'readValue'` assumes that it will be invoked by a simple layout object, and that *context* will hold a variable *\$instance*, which resolves to the original layout object.
- `'readLoop'` assumes that the calling object is a `$loop`. Each time the callback is invoked, it will process the current iteration of the loop using the current values of special loop variables:
 - `$loopValue` resolves to the data binding of the outer loop and allows direct access to the data for each iteration.
 - `$loopNumber` is the 1-based iteration count of the loop.

See “[The \\$loop Layout Object](#)” in the book *Using Zen Mojo Plugins* for complete details and examples.

The following template method contains a layout graph that uses both callback options. It consists of two main layout objects: `$p` invokes the callback once (with '`=[$method.readValue]`'), and `$loop` invokes it repeatedly (with '`=[$method.readLoop]`').

```
ClientMethod myMainLayout(key, criteria) [ Language = javascript ] {
  var myLayoutGraph = {};
  myLayoutGraph = { children: [
    { type: '$p', myValue: 'a single value' , $content: '=[ $method.readValue ]' },
    { type: '$loop', value: ['nergles', 'frabsters'],
      children: [ { type: '$div', $content: '=[ $method.readLoop ]' } ]
    }
  ] }
  return myLayoutGraph;
}
```

The layout objects are rendered as follows:

- The `$p` object invokes `'=[$method.readValue]'`, which returns a string that includes the value of layout object property `myValue` (via `context.$instance.myValue`).
- The `$loop` object defines an array of two values, and invokes `'=[$method.readLoop]'` once for each of them. Each iteration returns a string value that includes the current loop count (contained in `context.$loopNumber`), and the current item in the `value` array (contained in `context.$loopValue`).

When `myLayoutGraph` is rendered, the following text is displayed:

```
myValue = "a single value"
myLoop(1) = "nergles"
myLoop(2) = "frabsters"
```

6.4 Specifying Style Attributes in Layout Graphs

A layout graph has attributes that control its overall style. Within a layout graph, you can specify attributes for layout objects that control their appearance as well.

The following fragment shows an example:

```
content = {
  documentStyle: 'background:white;min-height:100%;',
  children:[
    {type: '$para',
      style: 'line-height:150%;',
      blockStyle: 'padding-left:20px;padding-bottom:20px;',
      title: 'inbox',
      titleStyle: 'font-size:36px;color:#606060;',
      text: 'Messages and answers you have received',
    },
    ...
  ],
}
```

Most attributes that control appearance fall into two categories:

- Some control the `style` attribute in the corresponding part of the generated HTML. These attributes generally have names like `style`, `blockStyle` and so on.

For these, specify a CSS style instruction.

- Some control the `class` attribute in the corresponding part of the generated HTML. These attributes generally have names like `paraClass`, `tableClass` and so on.

For these, specify the name of a CSS class, as given in one of the available CSS style sheets (see the chapter “[Including Style Sheets](#)” for details).

6.5 Using the `sourceData` Property in a Layout Graph

Within a layout method, you can bypass the data object, and instead use the special `sourceData` object.

Specifically, the layout graph can include an object-valued property named `sourceData`, which is handled specially. In this case, Zen Mojo uses the `sourceData` object as the current data object, instead of the usual data object.

Important: If the layout graph includes the `sourceData` property, Zen Mojo does not invoke the `ongetdata` callback and does not have access to the data object normally provided by that method.

If the layout graph includes the `sourceData` property, the syntax `'=[propertyname]'` refers to properties of the `sourceData` object.

The following example show an example:

```
ClientMethod myGetMainViewLayout(key, criteria) [ Language = javascript ]
{
    var myLayoutGraph = {};
    myLayoutGraph = {
        sourceData: {
            name : 'Higgins,Bert',
            dob: '15-July-1973',
            homeaddress: {
                street: '4595 Center Street',
                city: 'Hopkinton',
                postalcode: '89304'
            },
            favoritecolors : ['blue']
        },
        children: [
            { type: '$header', $content: $$$Text('sourceData Demo')},
            { type: '$p', title:'Name', $content:'=[name]' },
            { type: '$p', title:'Birth Date', $content:'=[dob]' },
            { type: '$p', title:'Home City', $content:'=[homeaddress.city]' },
            { type: '$p', title:'First Favorite Color', $content:'=[favoritecolors[1]]' },
            { type: '$p', title:'Number of Favorite Colors', $content:'=[favoritecolors.length]' },
        ]
    }
    return myLayoutGraph;
}
```

In this example, the `sourceData` object is defined inline (that is, within the layout graph). You could instead use a reference to an object defined earlier:

```
sourceData: myObject
```

6.6 Using Stashed Values within sourceData

The `sourceData` object provides a convenient way to use values that you have previously *stashed* on the client, so that you can avoid calling the server unnecessarily. Specifically, you can do the following:

1. In appropriate places in the client methods, stash values by saving them in properties of the page instance or the template instance. For example:

```
zenPage._MyNewValues={name:mainView.getControlValue('enterName'),
                      city:mainView.getControlValue('enterCity')};
```

For details, see “[Stashing Values](#)” in the chapter “[Background Information and Tasks](#).”

2. When you create a `sourceData` object, include references to the stashed values.
3. Within the layout graph, refer to the `sourceData` object (as previously described in “[Using the sourceData Property in a Layout Graph](#)”). That is, use the syntax `'=[propertyname]'` to refer to properties of the `sourceData` object.

7

Using Keys

Zen Mojo provides a system of keys ([introduced](#) in the first chapter) that you can use to drive user interaction. This chapter describes how to use this system. It discusses the following topics:

- [Possible approaches](#)
- [How to specify initial keys](#)
- [How to set keys and update the layout](#)
- [How to use the criteria argument](#)
- [How to get the keys currently in use](#)
- [How to work with document stacks](#)

At an early stage in development, make sure to define a naming convention for your keys; see “[Naming Conventions](#),” earlier in this book.

7.1 Possible Approaches

You can use keys in different ways. This section describes two primary approaches. A hybrid approach and other approaches are also possible.

7.1.1 Driving One documentView from Another documentView

In this approach, the pageContents area contains two documentView components and one of them (typically, the left or top one) drives the other one (typically the right or bottom one).

The typical scenario is that one documentView (A) contains a set of buttons or menus, each with a unique key. When the user selects an item in that documentView, the **onselect()** [event handler](#) does the following:

1. It obtains the current data key, layout key, or both from documentView A.
2. Based on that key or keys, it then sets the data key, layout key, or both for the other documentView (B).

Note that within **onselect()**, the current value of the selected item is also available, so the logic could use that information as well.

3. It then updates the layout of documentView B.

Note that not all page managers support multiple documentViews.

7.1.2 Using a Document Stack

Zen Mojo also provides support for document stacks; see the section “[Working with Document Stacks](#).”

In the document stack approach, the `pageContents` area contains a single `documentView` component. In a typical scenario, when the user selects an item, the `onselect()` event handler does the following:

1. It obtains the current data key, layout key, or both.
2. Based on that key or keys, it calls `pushDocument()` to add a new version onto the document stack and display it within the `documentView`. Zen Mojo automatically provides a header that includes a back button, which the user can return to the previous stack level of the application.

Note that within `onselect()`, the current value of the selected item is also available, so the logic could use that information as well.

Note that document stacks are different from the browser history stack. Zen Mojo provides a separate set of tools that you can use to manage the browser history stack.

For information on `pushDocument()` and related methods, see the section “[Working with Document Stacks](#).”

7.2 Specifying the Initial Keys for a documentView

To specify the initial keys for a `documentView`, specify either or both of the following properties of the `<mojo:documentView>` element in the page class:

- `initialLayoutKey` — Initial layout key of this `documentView`
- `initialDocumentKey` — Initial data key of this `documentView`

For both properties, `home` is the default value.

For example:

```
<:mojo:documentView id="mainView"
developerMode="true"
initialDocumentKey="initialKey"
initialLayoutKey="initialKey"
ongetlayout = "return zenPage.getContent('layout',key,criteria);"
ongetdata = "return zenPage.getContent('data',key,criteria);">
...
</mojo:documentView>
```

7.3 Setting Keys and Updating the Layout

To set the keys for a `documentView` instance and then update the layout of that `documentView`, do the following within a suitable client method:

1. Use the syntax `zen('docViewId')` to refer to the `documentView` that has the given id.
2. Call the `setDocumentKey()` method, the `setLayoutKey()` method, or both of that `documentView`.
3. Call the `updateLayout()` method of that `documentView`.

For example:

```
var docView = zen('mainView');
docView.setDocumentKey(key);
docView.updateLayout();
```

The following list gives the details for these methods:

setDocumentKey()

```
ClientMethod setDocumentKey(key, criteria, level) [ Language = javascript ]
```

Sets the key and optional criteria for the data object (for the given level of the document stack). The arguments are as follows:

- *key* is a short, unique name that indicates the specific version of the content.
- *criteria* is either a string or an object that contains any additional information for the template to use. For example, this can be a search string entered by the user. Or it could be an object containing multiple properties used internally in your code.

Zen Mojo does not directly use the *criteria* argument. You can use this argument within your client methods, typically in methods called by your `ongetdata` and `ongetlayout` callbacks. For examples, see “[Using the Criteria Argument](#),” later in this chapter.

- *level* is the level (of the document stack) for which you are setting the key and criteria. (For each `documentView`, each level in the document stack can have its own key and criteria for its data object.)

If you are not using methods such as `pushDocument()` or `popDocument()`, you can simply ignore this argument. For information on the document stack, see the chapter “[Working with Document Stacks](#).”

This method also invalidates the cache (for the given level of the document stack). To update the layout, use `updateLayout()`, described later.

setLayoutKey()

```
ClientMethod setLayoutKey(key, criteria, level) [ Language = javascript ]
```

Sets the key and optional criteria for the layout graph (for the given level of the document stack). The arguments are similar to those used by `setDocumentKey()`; the difference is that `setLayoutKey()` affects the key and criteria used by the layout graph, rather than by the data object.

This method also invalidates the cache (for the given level of the document stack). To update the layout, use `updateLayout()`, described next.

updateLayout()

```
ClientMethod updateLayout(effect) [ Language = javascript ]
```

Invokes the `ongetdata` and `ongetlayout` callbacks of the `documentView`, using the current data and layout keys and criteria, (thus updating its current layout).

The *effect* argument specifies the transition effect to use, *if you are using the default page manager*. This argument is ignored for other page managers. This argument can be `'fade'`, `'fade-flakes'`, or `'fade-tiles'`.

7.4 Using the Criteria Argument

This section shows examples that use the criteria argument. In the first example, the client method creates an object for use as the *criteria* argument:

```

var reqId = {HistoryStart:this.historyStart,HistoryEnd:this.historyEnd};
reqId.SessionID = data.SessionID;
reqId.SessionMRN = data.SessionMRN;

docView.setDocumentKey(key, reqId);
docView.updateLayout();

```

The next example creates a *criteria* object that contains a search term entered by the user and a property used internally by the layout graph:

```

docView.setDocumentKey(key, {term:this.searchTerm, initLoad:true});
docView.setLayoutKey(key);
docView.updateLayout('fade');

```

The *criteria* argument is available within the `%OnGetJSONContent()` method as *pCriteria*. In the following example, the `%OnGetJSONContent()` method first checks to make sure that *pCriteria* is an object, and then extracts a property from it, for use in retrieving data from the server:

```

if $IsObject(pCriteria) {
    set tDocId=pCriteria.docId }
else {
    set tDocId=""
}
//use tDocId to retrieve specific data, if applicable

```

7.5 Getting Keys of a documentView

Zen Mojo also provides methods that you can use to *get* the current data and layout keys for any given documentView. These methods are methods of the documentView instance. The methods are as follows:

getDocumentKey()

```
ClientMethod getDocumentKey() [ Language = javascript ]
```

Returns the data key for this documentView (for the current level of the document stack).

getLayoutKey()

```
ClientMethod getLayoutKey() [ Language = javascript ]
```

Returns the layout key for this documentView (for the current level of the document stack).

7.6 Working with Document Stacks

To work with a document stack, you invoke methods of a documentView instance. To invoke one of these methods from a client method, use a couple of steps like the following:

```

var mainView = zen('mainView');
mainView.methodname();

```

In the first line, `zen('mainView')` is a reference to the documentView whose id is 'mainView'. The second line invokes a method associated with that documentView.

getLevel()

```
ClientMethod getLevel() [ Language = javascript ]
```

Returns the current document stack level. The initial stack level is 0. When you use **pushDocument()**, the level is incremented by 1. When you use **popDocument()**, the level is decremented by 1.

popAll()

```
ClientMethod popAll(render,clearDataCache) [ Language = javascript ]
```

Removes all document versions except for the original, updates the display, and sets the document stack level to 0. If *render* is true, the method re-renders the newly revealed previous document version. If *clearDataCache* is 1, the method clears the cache for the panel that is now displayed.

popDocument()

```
ClientMethod popDocument(render,clearDataCache) [ Language = javascript ]
```

Removes the most recent document version from the document stack, updates the display, and decrements the document stack level by 1. For information on *render* and *clearDataCache*, see **popAll()**.

popDocuments()

```
ClientMethod popDocuments(number, render, clearDataCache) [ Language = javascript ]
```

Removes the given number (*number*) of most recent document version from the document stack, updates the display, and decrements the document stack level as needed. For information on *render* and *clearDataCache*, see **popAll()**.

pushDocument()

```
ClientMethod pushDocument(layoutKey, layoutCriteria, dataKey, dataCriteria) [ Language = javascript ]
```

Adds a new document version onto the document stack and displays it (by default) in front of the previous versions. The new document version uses the given layout key and criteria and data key and criteria to fetch its layout and data. If *dataKey* is not provided, then the data from the current level is used.

This method also increments the document stack level by 1.

8

Interacting with Layout Objects

This chapter describes how to interact with layout objects currently displayed on a Zen Mojo page. It discusses the following:

- [Methods in this chapter](#)
- [How to get or set the value of a layout object](#)
- [How to refresh a single layout object](#)
- [How to access a layout object directly](#)

Note: The [jQuery Mobile Helper plugins](#) and the [HTML5 Helper plugin](#) also provide methods associated with the layout objects. These methods include `$show()` and `$hide()`. Such methods will be provided in all plugins in future releases, if possible. For details, see the applicable chapters of *Using Zen Mojo Plugins*.

8.1 Methods in This Chapter

The methods in this chapter are methods of each `documentView` instance. To invoke one of these methods from a client method, use a couple of steps like the following:

```
var mainView = zen('mainView');
mainView.methodname();
```

In the first line, `zen('mainView')` is a reference to the `documentView` whose id is `'mainView'`. The second line invokes a method associated with that `documentView`.

8.2 Getting or Setting the Value of a Layout Object

Use the following methods of the `documentView` to get or set values of layout objects:

getControlValue()

```
ClientMethod getControlValue(key, level) [ Language = javascript ]
```

Returns the value of the layout object that has given key, for the given level of the document stack.

setControlValue()

```
ClientMethod setControlValue(key, value, level) [ Language = javascript ]
```

Sets the value of the layout object that has the given key, for the given level of the document stack. This method does not fire a change event.

setItemValue()

```
ClientMethod setItemValue(key, value, level) [ Language = javascript ]
```

Sets the value of the layout object that has the given key, for the given level of the document stack. This method does not fire a change event.

8.3 Refreshing a Layout Object

To refresh a single layout object, use the following method of the `documentView`:

refreshItem()

```
ClientMethod refreshItem(key, level) [ Language = javascript ]
```

Re-renders the HTML for the layout object that has the given key, for the given level of the document stack.

Or invoke the `$refresh()` method of the layout object.

Note that most but not all layout objects can be refreshed. In some cases, there is no practical scenario for refreshing an object, so a few objects simply ignore these methods. The plugin reference documentation indicates which layout objects can be refreshed.

8.4 Accessing a Layout Object

Sometimes it is necessary to access a layout object directly. To do so, use the following method of the `documentView`:

getItemByKey()

```
ClientMethod getItemByKey(key, level) [ Language = javascript ]
```

Returns the layout object that has the given key, for the given level of the document stack.

This method is useful for layout objects that provide methods. The [jQuery Mobile Helper plugins](#) and the [HTML5 Helper plugin](#) provide such methods, which include `$show()` and `$hide()`. Such methods will be provided in all plugins in future releases, if possible. For details, see the applicable chapters of *Using Zen Mojo Plugins*.

9

Submitting Data to the Server

This chapter describes how to submit data from a Zen Mojo page to the server. It discusses the following:

- [Basic steps](#)
- [Example](#)

For an overview, see “[Submitting Data](#),” in the first chapter.

9.1 Submitting Data to the Server

To enable your page to send data to the server, do the following:

- In the template class, define a method that sends a submit object to the server. In this method:
 1. Obtain any needed values from the page. To get the value of a control, use `getControlValue()`. Also see the chapter “[Interacting with Layout Objects](#).”
 2. Create a submit object, which is a JSON object that carries the data to be submitted. This object can have any properties you need.
 3. Invoke the page method `submitData()`, using the submit object as an argument.

This method has the following signature:

```
ClientMethod submitData(key, data, notify) [ Language = javascript ]
```

Where:

- *key* — The key to use when sending data to the server
- *data* — A JSON object that carries the data to be sent to the server. You can omit this argument if there is no data to submit; that is, you do not need to include an empty object.
- *notify* — Either null or a function. If this argument is a function, the page sends the data *asynchronously* to the server (rather than synchronously). Then , when the data has been sent, the page executes the function named by this argument.

This method invokes the `%OnSubmitData()` method of the template class. See the next bullet item.

4. If desired, use data contained in the response object returned by `submitData()`.
- In the template class, define the `%OnSubmitData()` method. This method has the following signature:

```

ClassMethod %OnSubmitData(pKey As %String,
                          pID As %String,
                          pSubmitObject As %RegisteredObject,
                          ByRef pResponseObject As %RegisteredObject) As %Status

```

This method returns *pResponseObject* by reference.

- In the page class, within a suitable [event handler or callback](#), invoke the method that you created in the first bullet item.

9.2 Example

The following shows a simple example for a method that sends a submit object to the server:

```

ClientMethod myMainViewSelect(key, value) [ Language = javascript ]
{
  var mainView = zen('mainView');

  switch(key) {
  case 'submitEdits':
    var controlvalue1=mainView.getControlValue('enterName');
    var controlvalue2=mainView.getControlValue('enterCity');
    var submitobject={name:controlvalue1,city:controlvalue2};
    var response = zenPage.submitData('submitEdits',submitobject);
    if (response && response.message) {
      zenPage.showMessage(response.message);
    }
    mainView.updateLayout('fade');
    break;
  }
}

```

The template class defines the `%OnSubmitData()` method as follows:

```

ClassMethod %OnSubmitData(pKey As %String, pID As %String, pSubmitObject As %RegisteredObject,
                          ByRef pResponseObject As %RegisteredObject) As %Status
{
  Set tSC = $$$OK
  Try {
    if pKey="submitEdits"{
      set id=1
      set tPerson          = ##class(Sample.Person).%OpenId(id)
      set tPerson.Name     = pSubmitObject.name
      set tPerson.Home.City = pSubmitObject.city

      Set pResponseObject = ##class(%ZEN.proxyObject).%New()
      Set tSC = tPerson.%Save()
      If $$$ISERR(tSC) {
        Set pResponseObject.message = "Your changes were not successfully saved"
        Quit}

      Set pResponseObject.message = "Your changes have been successfully saved"
    }
  }
  Catch(ex) {
    Set tSC = ex.AsStatus()
  }
  Quit tSC
}

```

10

Working with Multiple Templates (Explicit Dispatch)

So far this book has discussed only single-template applications, but your Zen Mojo application can use multiple template classes, which enables you to divide your application code into more easily maintained units.

Zen Mojo provides two ways to work with multiple templates: explicit dispatch (discussed in this chapter) and dynamic dispatch (discussed in the [next chapter](#)). In either case, the methods in the templates can use all the tools described in the earlier chapters of this book.

This chapter discusses the following topics:

- [Introduction to the explicit dispatch mechanism](#)
- [How to associate areas with templates](#)
- [How to programmatically specify which area \(or template\) the page should use](#)
- [How to access current area and template from the page instance](#)

10.1 Introduction to Explicit Dispatch

With the explicit dispatch mechanism, each template corresponds to an application *area*. Areas serve as short logical names for the templates, which have longer and less convenient class names.

The user makes selections on the page, invoking the `goToArea()` method or other methods that set the current area and therefore determine which template to use. The page methods `getContent()` and `submitData()` always use the current template, whichever that is.

Creating a page like this is only slightly more complex than creating a simple page. When a page has been set up this way, it is easy to extend the page by adding more templates.

10.2 Associating Areas with Templates

To define areas (and associate a template with each), implement the `%OnGetTemplateList()` method in the Zen Mojo page class. This method is as follows:

```
method %OnGetTemplateList(Output pTemplates) as %Status
```

Where *pTemplates* is expected to be a multidimensional array with the following nodes:

Node	Contents
pTemplate	Count of subnodes
pTemplate(i) where <i>i</i> is an integer	A \$LISTBUILD list that consists of the following items, in order: <ul style="list-style-type: none"> Logical name of the area Full package and class name of the corresponding template XML namespace of the template

The following shows an example implementation:

```
Method %OnGetTemplateList(Output pTemplates) As %Status
{
  Set tSC = $$$OK
  Try {
    Kill pTemplates

    set area="area-home"
    set template="MyApp.Templates.HomeTemplate"
    set ns="http://www.corporate.com/myapp/home"
    Set pTemplates($I(pTemplates))=$LB(area,template,ns)

    set area="area-second"
    set template="MyApp.Templates.SecondaryTemplate"
    set ns="http://www.corporate.com/myapp/secondary"
    Set pTemplates($I(pTemplates))=$LB(area,template,ns)
  }
  Catch(ex) {
    Set tSC = ex.AsStatus()
  }
  Quit tSC
}
```

10.3 Setting the Current Area and Key for the Page

To set the current area for the page, use the following methods of the page instance. To invoke one of these methods from a client method, use the syntax `zenPage.methodname()`.

gotoArea()

```
ClientMethod gotoArea(area, key1, key2, nohistory) [ Language = javascript ]
```

Sets the current area, current key1, and current key2. If *nohistory* is true, then do not push this change onto the history stack.

gotoAreaKB()

```
ClientMethod gotoArea(evt,area, key1, key2, nohistory) [ Language = javascript ]
```

Given a keyboard event (*evt*), sets the current area, current key1, and current key2. If *nohistory* is true, then do not push this change onto the history stack.

If you call any of these method, you should define **changeAreaHandler()** to specify what should happen when the current area and keys change.

10.3.1 Implementing `changeAreaHandler()`

If you call `gotoArea()` or `gotoAreaKB()`, be sure to define `changeAreaHandler()` in the page class. This method is invoked automatically by those methods, and it should specify what happens when the keys change.

The `changeAreaHandler()` method has following signature:

```
ClientMethod changeAreaHandler() [ Language = javascript ]
```

10.4 Accessing the Current Template

To access the current template, you can use the following methods of the page instance. To invoke one of these methods from a client method, use the syntax `zenPage.methodname()`

`getTemplate()`

```
ClientMethod getTemplate() [ Language = javascript ]
```

Returns a reference to the current template. Use this method so that, for example, you can execute methods of the template.

`getTemplateForArea()`

```
ClientMethod getTemplateForArea(area) [ Language = javascript ]
```

Returns the name of the template class associated with a given area.

11

Working with Multiple Templates (Dynamic Dispatch)

So far this book has discussed only single-template applications, but your Zen Mojo application can use multiple template classes, which enables you to divide your application code into more easily maintained units.

Zen Mojo provides two ways to work with multiple templates: explicit dispatch (discussed in the [previous chapter](#)) and dynamic dispatch (discussed in this chapter). In either case, the methods in the templates can use all the tools described in the earlier chapters of this book.

This chapter discusses the following topics:

- [Introduction to the dynamic dispatch mechanism](#)
- [How to modify the page class to use dynamic dispatch](#)
- [How to define suitable templates for dynamic dispatch](#)

11.1 Introduction to Dynamic Dispatch

Compared to explicit dispatch, the dynamic dispatch mechanism uses a larger set of small templates. Each template is key-specific and provides only one part of the page logic. For example, one template might only return content objects for a specific key. Another template would only handle select events for the same key.

When the dynamic dispatch mechanism is enabled (via the `templateDispatchMode` property), Zen Mojo checks for a key-specific template at several specific points within the page logic, loads the template, and then uses it for the designated purpose. Specifically, if dynamic dispatch is enabled:

- When the client invokes the **getContent()** method of the page, Zen Mojo loads the applicable template and calls the **getContent()** method of that template class.

The following subsection explains how Zen Mojo finds the applicable template in this scenario and in the following scenarios.

- When a select event occurs on the page, Zen Mojo loads the applicable template and calls the **onselect()** method of that template class.
- When a change event occurs on the page, Zen Mojo loads the applicable template and calls the **onchange()** method of that template class.

- When an event of any other type occurs on the page, Zen Mojo loads the applicable template and calls the `onevent()` method of that template class.

11.1.1 How Zen Mojo Finds Templates, in Dynamic Dispatch

In dynamic dispatch, Zen Mojo finds the template to load as follows:

1. Zen Mojo looks for a template in the following XML namespace:

```
templateDispatchBaseNamespace/scenario
```

Where `templateDispatchBaseNamespace` is the value of the page property of that name and `scenario` indicates the scenario.

If a data object is requested, `scenario` is `data`. If a layout graph is requested, `scenario` is `layout`. If an event (of any kind) has occurred, `scenario` is `events`.

2. In this namespace, Zen Mojo loads the template whose short class name is `key`, where `key` is the current key.

Note that this discussion assumes that you are using `data` as the provider name when you are retrieving data objects and that you are using `layout` as the provider name when you are retrieving layout graphs.

11.2 Modifying the Page to Use Dynamic Dispatch

To modify the page class to use dynamic dispatch:

- Override the `templateDispatchMode` property of the class and specify its `InitialExpression` keyword as 1:

```
Property templateDispatchMode As %ZEN.Datatype.boolean [ InitialExpression=1 ] ;
```

- Override the `templateDispatchBaseNamespace` property of the class and specify its `InitialExpression` keyword. For the value of this keyword, specify the base part of the XML namespace used by the templates.

For example, the Zen Mojo plugin reference documentation application uses this value:

```
Property templateDispatchBaseNamespace As %ZEN.Datatype.string  
[ InitialExpression="http://www.intersystems.com/zen/mojo/documentation" ] ;
```

11.3 Defining the Templates

To enable dynamic dispatch, define a set of templates in three groups.

- One group returns data objects. These templates must be in the XML namespace `templateDispatchBaseNamespace/data`. (That is, the `NAMESPACE` parameter of each of these template must equal `templateDispatchBaseNamespace/data` where `templateDispatchBaseNamespace` is the value of the page class property of that name.)

If you use a different provider name (a name other than `data`) to retrieve data objects, replace `data` with your provider name.

These templates must implement `onGetContent()` and (if suitable) `%OnGetJSONContent()`.

- One group returns layout graphs. These templates must be in the XML namespace *templateDispatchBaseNamespace/layout*

If you use a different provider name (a name other than `layout`) to retrieve layout graphs, replace `layout` with your provider name.

These templates must implement **onGetContent()**.

- One group defines all event handling. These templates must be in the XML namespace *templateDispatchBaseNamespace/events*

These templates must implement **onselect()**, **onchange()**, or **onevent()**, as needed.

Tip: For clarify and ease of maintenance, you could place these groups of template classes in subpackages named `Data`, `Layout`, and `Events`, respectively. For an example, see the `%ZEN.Mojo.PluginDocumentation.Templates` package, which is part of the plugin reference documentation application.

12

Using Style Sheets

This chapter explains how to include and use CSS style sheets in Zen Mojo pages. It discusses the following topics:

- [How to specify style information for layout objects](#)
- [How to include CSS style sheets](#)
- [Precedence of style instructions](#)

12.1 Specifying Style Information for Layout Objects

Within a layout graph, you can typically specify the properties that control the CSS class and CSS style for specific layout objects. For example:

```
{type: '$ContentPane', key: 'layoutContainer-1', style: 'width:100%;height:100%;'  
  ...  
}
```

The details vary by layout object.

12.2 Including Style Sheets

A Zen Mojo application can use CSS style sheets, which you can provide in any combination of the following ways:

- Include the style sheet directly within either the application class or the page class.

To do so, include an XData block named `Style`. The following shows a simple example:

```
XData Style  
{  
<style type="text/css">  
  
  table.tpTable {  
    background: green;  
    border: 1px solid red;  
    font-family: courier new;  
  }  
  
</style>  
}
```

Copy and paste the contents of the style sheet into the XData block, between the `<style type="text/css">` and `</style>` tags.

If you include the style sheet in the page class, its styles are available to that class.

If you include the style sheet in the application class, its styles are available to all page classes that use this application class. For Zen Mojo (in contrast to Zen), an application typically has only a single page.

- Define the *CSSINCLUDES* parameter of either the application class or the page class. For the value, specify a comma-separated list of CSS style sheets. For example:

```
Parameter CSSINCLUDES = "dojo-release-1-9-1/dijit/themes/claro/claro.css,  
dojo-release-1-9-1/gridx/resources/claro/Gridx.css";
```

Caché assumes any relative paths refer to the directory *install-dir/csp/broker*. The parameter in the application class affects all page classes that use the application class; the parameter in the page class affects only that class.

Or, if you are using CSS version 3.0, specify the *CSS3INCLUDES* parameter in the same way.

12.3 Precedence of Styles

When Zen Mojo generates a page, it collects all the CSS style information and applies the information in these sources in the following order. Note that when conflicts exist, the styles that are defined *last* take precedence:

1. Styles defined by the CSS files for the Zen component library. The filenames have the form *ZEN_Component_*.css*.
2. Styles defined within the applicable application class, if any. These styles are evaluated in the following order:
 - An XData Style block containing CSS statements
 - A reference to an external CSS file using the *CSSINCLUDES* parameter
3. Styles defined within the page class. These styles are evaluated in the following order:
 - A reference to an external CSS file using the *CSSINCLUDES* parameter
 - An XData Style block containing CSS statements
 - Properties within the layout graph. For an individual layout object, these may provide:
 - Simple values for HTML attributes, such as "300" for *height*
 - Literal CSS statements, such as "color:red; background: yellow;")
 - The name of a CSS style that is defined somewhere in the cascade of styles

To determine the name of the CSS style used in a specific location, use the browser or a suitable add-on tool to inspect the generated HTML directly.

13

Additional Steps for Mobile and Hybrid Applications

Zen Mojo is designed for client-server interaction, and you can use it to create applications whose web clients are on either mobile devices or on desktops. Most of this book discusses tools and techniques that apply to both scenarios. This chapter, however, discusses additional steps that apply when you create applications for mobile devices. This chapter discusses the following topics:

- [How to specify meta tags](#)
- [How to modify the Zen Mojo page class to accept offline requests](#)
- [Requirements of the packaged application](#)
- [How to generate offline pages](#)

For an overview, see “[Mobile Applications](#)” and “[Hybrid Applications](#)” in the first chapter.

Note: Experienced Zen developers please note that some common design features of desktop Zen and CSP applications should not be used in mobile and hybrid apps. For example, avoid using Zen special variable `%session` in hybrid apps. When a user stops using the app and comes back later, a new session may be started even though the mobile app is still in the state the user left it. If the server requires `%session` to be in a specific state, this could cause serious problems.

13.1 Specifying Meta Tags

When you create a page for use on mobile devices, be sure to implement the `%OnDrawHTMLMeta()` method in the page class so that the page contains the appropriate meta tags. This method has the following signature:

```
Method %OnDrawHTMLMeta() As %Status
```

This method is called at the start of the HTML HEAD section of the page (just after the title). Your implementation should write any needed meta tags. For example:

```
Method %OnDrawHTMLMeta() As %Status
{
    Write "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0,maximum-scale=1,
user-scalable=no\" />"
    Quit $$$OK
}
```

This example sets both the initial and maximum scale equal to 1 and disables user scaling.

13.2 Modifying the Page Class to Support Offline Use

For use in a mobile environment, a Zen Mojo page class must be able to accept requests from the offline version of the page. To modify the page class in this way:

- Override the *ALLOWMOBILE* parameter of the class and specify its value as 1:

```
Parameter ALLOWMOBILE =1;
```

- Override the *remoteBrokerAddress* property of the class and specify its *InitialExpression* keyword. For the value of this keyword, specify the URL of the *%CSP.Broker.cls* file used by the web application. The URL must have the following form:

```
http://<remote-server>:<port>/csp/user/%25CSP.Broker.cls
```

where %25 is used to represent a % character. For example:

```
Property remoteBrokerAddress As %ZEN.Datatype.string  
[ InitialExpression="http://yourserver:55001/csp/yourapp/%25CSP.Broker.cls" ] ;
```

If you authenticate with *CacheUserName* and *CachePassword*, the *remoteBrokerAddress* must also specify *CacheNoRedirect=1* to avoid browser cross-domain policy issues. The following example changes *http:* to *https:*, adds authentication, and specifies *CacheNoRedirect=1* (second line has been split for readability):

```
Property remoteBrokerAddress As %ZEN.Datatype.string  
[ InitialExpression="https://yourserver:55001/csp/yourapp/%25CSP.Broker.cls?CacheUserName=_SYSTEM  
&CachePassword=SYS&CacheNoRedirect=1" ] ;
```

13.3 Requirements of the Packaged Application

Make sure that the packaged application provides the following items, all in a single directory:

- The generated offline page (an HTML file), as discussed in the next section.
- The JavaScript file generated for the page class, *packagename_classname.js* (which you can find in the *install-dir/csp/broker* directory).
- Files required by the core parts of Zen Mojo, specifically, the following files from the *install-dir/csp/broker* directory:
 - Files with names of the form *ZEN*.js*
 - Files with names of the form *ZEN_Mojo*.js*
 - *zenutils.js*
 - *cspxmlhttp.js*
 - *cspbroker.js*
 - *zenutils.js*
- Any files (or subdirectories of files) needed by the plugin or plugins.
- Any additional files (such as image files) needed by the application.

13.4 Generating Offline Pages

This section describes how to generate an offline version (pure HTML) of a Zen Mojo page, for use as part of a mobile application.

Note: Make sure to first modify the page class so that it supports offline use, as described [earlier](#) in this chapter.

13.4.1 Generating Only the Offline Page

To generate an offline version (pure HTML) of a Zen Mojo page, use the `CreateOfflinePage()` method of the `%ZEN.Mojo.Utils`. This method is as follows:

```
ClassMethod CreateOfflinePage(pClassName As %String,
                             ByRef pFileName As %String = "",
                             pVerbose As %Boolean = 1) As %Status
```

Where *pClassName* is the full package and class name of the Zen Mojo page class, *pFileName* is the name of the file generated by this method, and *pVerbose* is a Boolean value to specify whether the method should write output messages. Note that *pFileName* is passed by reference, so you should place a period before this argument, if you use it.

For example:

```
set class="ZMdemo.chui.HomePage"
do ##class(%ZEN.Mojo.Utils).CreateOfflinePage(class,.filename,0)
write filename
```

This method writes an HTML file to the `install-dir/CSP/namespace` directory where *install-dir* is the directory where you installed Caché or Ensemble, and *namespace* is the namespace in which you ran this method.

Note that the generated HTML file requires a set of files that must be in the same directory; without those files, this page appears empty or mostly empty. When you package the application, you copy this HTML file and all of its dependencies into a single directory. See the next section.

13.4.2 Generating an Offline Bundle

Zen Mojo provides an additional utility method that executes `CreateOfflinePage()`, identifies the generated files that it needs, and copies all these files to a target directory. This method, `CreateOfflineBundle()`, is intended to simplify the process of adding the files to a PhoneGap project. This method is as follows:

```
ClassMethod CreateOfflineBundle(pClassName As %String, pTargetFolder As %String) As %Status
```

Where *pClassName* is the full package and class name of the Zen Mojo page class and *pTargetFolder* is the target directory. The method creates this directory if it does not exist.

Important: This method identifies only the files generated by the Zen Mojo page class. It does not attempt to examine the `CSSINCLUDES` or `JSINCLUDES` parameters, because in some cases, it would also be necessary to resolve the dependencies implied by those parameters. Similarly, if the page uses external resources, such as image files, this method does not identify those.

A

Tips for Managing Caching During Development

Several caching mechanisms operate when you display Zen Mojo pages, and these can prevent you from seeing the most recently compiled version of a page. This appendix provides information on disabling these mechanisms and on clearing the caches, so that you can avoid frustration during the development process. This appendix discusses the following topics:

- [The CSP Gateway cache](#)
- [The browser cache](#)
- [The Zen Mojo page cache](#)

A.1 CSP Gateway Cache

Zen Mojo uses the CSP Gateway, which can cache files automatically, depending on how your web application is defined (see “[Parts of a Zen Mojo Application](#),” in the first chapter).

To prevent the CSP Gateway from caching files for your Zen Mojo application:

1. In the Management Portal, click **System Administration > Security > Applications > Web Applications**. This displays the **[System] > [Security Management] > [Web Applications]** page.
2. Click the name of the web application that contains your Zen Mojo page.
3. For the **Serve Files** option, select **Always**.
4. Click **Save**.

To clear this cache, if needed, do the following:

1. In the Management Portal, click **System Administration > Configuration > CSP Gateway Management**.
2. Click **System Status**.

When you are prompted to log in, provide a username and password.

3. In the bottom part of the page, click the **Clear Cache** button for each page that you want to remove from this cache.

Note: When you deploy the application, it is generally best to change the **Serve Files** option to **Always and cached**.

A.2 Browser Cache

For information on disabling caching by the browser and on clearing the browser cache, see the help provided by your browser.

The browser cache is separate from the CSP Gateway cache and is used at a later stage when serving a page. This means that when you clear the caches, you should clear the browser cache *after* clearing the CSP Gateway cache.

Note that your browser might provide different options for refreshing the display (using the cache) and for reloading the page (bypassing the cache). For example, on Chrome, the **F5** button refreshes the display and **Ctrl+R** reloads the page.

A.3 Zen Mojo Page Cache

Zen Mojo also caches information, as described [earlier in this book](#). There is no switch to disable this mechanism as a whole, but you can invalidate this cache at suitable points in your processing. To do so, invoke the **invalidate()** method of the `documentView` instance or of the Zen Mojo page.

B

Upgrade Steps for Beta Program Customers

This appendix describes changes that affect code you created if you were in the Zen Mojo beta program.

B.1 Class Renaming

The Zen Mojo classes have been renamed. Rather than being contained in the %ZEN.Mobile package, the classes are now contained in the %ZEN.Mojo package.

B.1.1 Upgrade Steps

- In each Zen Mojo page class, edit the class so that it extends %ZEN.Mojo.basePage instead of %ZEN.Mobile.basePage. For example, change this:

```
Class Demo.MyPage Extends %ZEN.Mobile.basePage
```

To this:

```
Class Demo.MyPage Extends %ZEN.Mojo.basePage
```

- In each Zen Mojo application class, edit the class so that it extends %ZEN.Mojo.baseApplication instead of %ZEN.Mobile.baseApplication.
- In each Zen Mojo template class, edit the class so that it extends %ZEN.Mojo.Component.contentTemplate instead of %ZEN.Mobile.Component.contentTemplate.

B.2 Namespace Changes

The documentView component and all plugins are now in the XML namespace "http://www.intersystems.com/zen/mojo" rather than "http://www.intersystems.com/zen/mobile". This means that it is necessary to update the use of XML namespaces in the pageContents of your page classes.

Also, the convention is now to use the namespace prefix `mojo` rather than `zmf`, for the Zen Mojo namespace.

B.2.1 Upgrade Steps

- In the <pane> element, change this:

```
xmlns:zmf="http://www.intersystems.com/zen/mobile"
```

To this:

```
xmlns:mojo="http://www.intersystems.com/zen/mojo"
```

- In the rest of the pageContents XData block, change the zmf prefix to mojo

B.2.2 Example

Consider the following fragment, from a class that worked with Zen Mojo beta code:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{<pane xmlns="http://www.intersystems.com/zen"
xmlns:zmf="http://www.intersystems.com/zen/mobile" layout="none">
<zmf:documentView id="mainView"
...

```

To work with the released Zen Mojo, this fragment would have to look like the following instead:

```
XData pageContents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{<pane xmlns="http://www.intersystems.com/zen"
xmlns:mojo="http://www.intersystems.com/zen/mojo" layout="none">
<mojo:documentView id="mainView"
...

```

B.3 Plugin Refactoring

The name of all the plugins have also changed, and there are now two kinds of plugins: page manager plugins and helper plugins.

Each documentView must contain exactly one page manager plugin and can contain multiple helper plugins. The page manager plugin defines the logic that controls the page, and the helper plugins contain the logic that defines the layout objects.

In the pageContents XData block, the documentView element must contain the page manager plugin as a child element. The page manager plugin must contain the helper plugins as its child elements, as follows:

```
<mojo:documentView id="mainView" additional attributes here...>
  <mojo:somePageManagerPlugin>
    <mojo:someHelperPlugin/>
    <mojo:anotherHelperPlugin/>
  </mojo:somePageManagerPlugin>
</mojo:documentView>
```

This example shows indentations for added clarity; you do not need to include any indentation.

For information on the plugins, see *Using Zen Mojo Plugins*.

The core plugin has been removed. There is a new default helper plugin, which provides the \$loop and \$content elements that had formerly been in the core plugin. This helper plugin, however, does not provide the other layout objects of the old core plugin. It is expected that no customers are actually using those layout objects.

B.3.1 Upgrade Steps and Examples

The following list provides examples of specific replacements:

CHUI plugin

If you had used the Chocolate-Chip UI plugin, your `documentView` might have looked like this:

```
<zmf:documentView id="mainView" additional attributes here...>
<zmf:chuiPlugin/>
<zmf:googleMapsPlugin/>
</zmf:documentView>
```

Replace this with the following:

```
<mojo:documentView id="mainView" additional attributes here...>
<mojo:chuiPageManager>
<mojo:chuiHelper/>
<mojo:googleMapsHelper/>
</mojo:chuiPageManager>
</mojo:documentView>
```

If you had not included `<mojo:googleMapsPlugin/>` in the original, omit `<mojo:googleMapsHelper/>` in your new version.

Dojo plugin

If you had used the Dojo plugin, your `documentView` might have looked like this:

```
<zmf:documentView id="mainView" additional attributes here...>
<zmf:dojoPlugin/>
<zmf:googleMapsPlugin/>
</zmf:documentView>
```

Replace this with the following:

```
<mojo:documentView id="mainView" additional attributes here...>
<mojo:dojoPageManager>
<mojo:dojoDijitHelper/>
<mojo:dojo2DChartHelper/>
<mojo:dojoGridXHelper/>
<mojo:googleMapsHelper/>
</mojo:dojoPageManager>
</mojo:documentView>
```

If you had not included `<mojo:googleMapsPlugin/>` in the original, omit `<mojo:googleMapsHelper/>` in your new version. Also note that there are multiple Dojo helper plugins, and that you might not need all of these.

JQuery Mobile plugin

If you had used the JQuery Mobile plugin, your `documentView` might have looked like this:

```
<zmf:documentView id="mainView" additional attributes here...>
<zmf:jqmPlugin/>
<zmf:googleMapsPlugin/>
</zmf:documentView>
```

Replace this with the following:

```
<mojo:documentView id="mainView" additional attributes here...>
<mojo:jQMPageManager>
<mojo:jQMHelper/>
<mojo:googleMapsHelper/>
</mojo:jQMPageManager>
</mojo:documentView>
```

If you had not included `<mojo:googleMapsPlugin/>` in the original, omit `<mojo:googleMapsHelper/>` in your new version.

Core plugin

If you had used the core plugin, your `documentView` might have looked like this:

```
<zmf:documentView id="mainView" additional attributes here...>
<zmf:corePlugin/>
<zmf:googleMapsPlugin/>
</zmf:documentView>
```

Replace this with the following:

```
<mojo:documentView id="mainView" additional attributes here...>
<mojo:mojoDefaultPageManager>
<mojo:mojoDefaultHelper/>
<mojo:googleMapsHelper/>
</mojo:mojoDefaultPageManager>
</mojo:documentView>
```

If you had not included `<mojo:googleMapsPlugin/>` in the original, omit `<mojo:googleMapsHelper/>` in your new version.

Note that the Mojo default helper does not define all the objects that had been contained in the core plugin; it defines only `$loop`, `$if`, and `$content`. If you had used other layout objects, also add another helper plugin such as `<mojo:HTML5Helper/>` and update your layout graphs to use layout objects supported by that plugin.

B.4 Event Handling Changes

The `documentView` component no longer supports the `onselect`, `onchange`, and `onevent` callback attributes. Instead, this component automatically invokes the **`onselect()`**, **`onchange()`**, and **`onevent()`** methods of the associated template, if any. These methods have the following signatures:

```
ClientMethod onselect(key, value, docViewId) [ Language = javascript ]
ClientMethod onchange(key, value, final, docViewId) [ Language = javascript ]
ClientMethod onevent(eventType, key, value, docViewId) [ Language = javascript ]
```

Where:

- *key* — key of the relevant layout object
- *value* — value of that layout object, if any
- *docViewId* — id of the `documentView` in which the select action, change action, or other event occurred
- *final* — indicates whether the value passed is the final value
- *evtType* — type of the event

These methods do nothing by default. Zen Mojo provides them so that you can customize the behavior of your page. For details, see “[Event Handlers](#),” earlier in this book.

B.4.1 Upgrade Steps

To handle this part of the upgrade, use the following approach:

1. Examine the `onselect`, `onchange`, and `onevent` callback attributes that are currently defined in your `documentView` components. For example:

```
onselect="zenPage.myMainViewSelect(key,value);"
```

Remove this attribute from the `documentView`.

Also make a note of the `id` of the `documentView`.

2. Examine the method referred to by the callback. Following recommended practices, this method would typically find the associated template and invoke a method in it, as in the following example:

```
ClientMethod myMainViewSelect(key, value, final) [ Language = javascript ]
{
    // check whether template has a handler for selection; if so, use it
    var template = this.getTemplate();
    if (template && template.myMainViewSelect) {
        return template.myMainViewSelect(key,value,final);
    }
}
```

This method is no longer needed. Remove it.

3. In the template class, override the `onselect()` method (via **Class > Refactor > Override**). For simplicity, you can create this method as a simple dispatcher. Check the value of the `docView` argument and use that to determine which existing template method to invoke. For example:

```
ClientMethod onselect(key, value, docViewId) [ Language = javascript ]
{
    if (docViewId='mainView') {
        myMainViewSelect(key, value);
    }
}
```

If the page has other `documentView` elements that define `onselect`, repeat steps 1 and 2, and add additional branches to the new `onselect()` method.

Use a similar procedure for the now-removed `onchange` and `onevent` callback attributes.

