Customizing the LiveCycle® Workspace ES User Interface

Adobe® LiveCycle® Workspace ES

July 2009

Update 1

© 2009 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle® Workspace ES Update 1 Customizing the Adobe LiveCycle Workspace ES User Interface for Microsoft® Windows®, Linux®, and UNIX®

Edition 2.2, July 2009

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names, company logos and user names in sample material or sample forms included in this documentation and/or software are for demonstration purposes only and are not intended to refer to any actual organization or persons.

Adobe, the Adobe logo, Acrobat, Distiller, Flash, Flex, FrameMaker, LiveCycle, PageMaker, Photoshop, PostScript, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the US and other countries.

All other trademarks are the property of their respective owners.

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by the IronSmith Project (http://www.ironsmith.org/).

This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/).

This product includes software developed by the Indiana University Extreme! Lab (http://www.extreme.indiana.edu/).

This product includes copyrighted software developed by E. Wray Johnson for use and distribution by the Object Data Management Group (http://www.odmg.org/).

Portions © Eastman Kodak Company, 199- and used under license. All rights reserved. Kodak is a registered trademark and Photo CD is a trademark of Eastman Kodak Company.

Powered by Celequest. Copyright 2005-2008 Adobe Systems Incorporated. All rights reserved. Contains technology distributed under license from Celequest Corporation. Copyright 2005 Celequest Corporation. All rights reserved.

Single sign-on, extending Active Directory to Adobe LiveCycle ES provided by Quest Software "www.quest.comlidentity-management" in a subsequent minor release that is not a bug fix (i.e., version 1.1 to 1.2 but not 1.1.1 to 1.1.2) of the Licensee Product that incorporates the Licensed Product.

The Spelling portion of this product is based on Proximity Linguistic Technology.

©Copyright 1989, 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1990 Merriam-Webster Inc. © Copyright 1990 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2003 Franklin Electronic Publishers Inc. © Copyright 2003 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2004 Franklin Electronic Publishers, Inc. © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1991 Dr.Lluis de Yzaguirre I Maura © Copyright 1991 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1990 Munksgaard International Publishers Ltd. © Copyright 1990 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1995 Van Dale Lexicografie bv © Copyright 1996 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1990 IDE a.s. © Copyright 1990 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2004 Franklin Electronics Publishers, Inc. © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1992 Hachette/Franklin Electronic Publishers, Inc. © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2004 Bertelsmann Lexikon Verlag © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2004 MorphoLogic Inc. © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 1990 Williams Collins Sons & Co. Ltd. © Copyright 1990 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. © Copyright 1993-95 Russicon Company Ltd.

© Copyright 1995 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

© Copyright 2004 IDE a.s. © Copyright 2004 All Rights Reserved Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

About This Document ... 6
- Who should read this document? ... 6
- Before you begin ... 6
- Additional information ... 6

1 Introduction ... 8
- Understanding the Workspace ES EAR file structure ... 8
- Understanding the custom EAR file structure ... 10
- Understanding the Workspace API architecture ... 12
- Using Workspace API components ... 14
- Understanding how to customize Workspace ES ... 16
- Understanding the stand-alone ... 17
- Upgrade considerations ... 17

2 Configuring the Development Environment - Installing the Flex SDK ... 18
- Installing the Flex SDK from LiveCycle ES ... 18
- Configuring Flex Builder to use the Flex SDK from LiveCycle ES ... 19

3 Configuring the Development Environment - Installing Ant to Flex Builder ... 21
- Installing the Ant plug-in to Flex Builder ... 21
- Installing the Ant-contrib library ... 22

4 Configuring the Development Environment - Configuring the Workspace Project ... 23
- Importing the Workspace ES project ... 23
- Modifying the build.xml file ... 26
- Creating a new builder for compiling Workspace ES ... 27
- Configuring the Flex project to browse Workspace ES stand-alone ... 28

5 Compiling and Deploying the Workspace Project ... 31
- Compiling the Workspace project ... 31
- Deploying a custom Workspace ES application for testing ... 33
  - Deploying a custom Workspace ES application as an EAR file ... 34
  - Configuring Flex Builder to deploy and run a custom Workspace ES application ... 34
- Testing the Workspace ES application ... 35
  - Testing the localization file ... 36
- Configuring the Workspace project for debugging ... 37
  - Compiling Workspace ES to another locale ... 40

6 Theme Customization - Modifying Colors ... 41
- Modifying colors in the CSS ... 42

7 Theme Customization - Replacing Images ... 45
- Adding custom images to the project ... 46
- Modifying the CSS to replace images ... 46

8 Layout Customization - Simplifying the User Interface ... 49
- Creating the application logic for a simplified Workspace ES ... 50
- Configuring the build.xml file to compile a custom MXML ... 51

9 Localization Customization - Localizing to Spanish ... 53

Adding a new folder to create a new localization file 55
Modifying the Workspace ES properties files 56
Localizing LiveCycle Workspace ES Help 58
Creating an error file 61
Configuring support for the new locale 61

10 Layout Customization - Creating a Three Pane View 63

Creating the application logic for a custom layout 64
Creating a folder for custom components 65
Creating a presentation model component for the content area 66
Creating a view component for the content area 70
Creating the presentation model component for the navigation area 73
Creating a view component for the navigation area 75
Creating the default application for a custom layout 80
Configuring the build.xml file to compile a custom MXML 81

11 Layout Customization - Creating a New Login Screen 82

Creating the application logic for a custom login screen 83
Creating a folder for custom components 83
Creating an ActionScript class to implement the lc:ILogin interface 84
Creating the user interface for the login screen 89
Creating the application logic in the default application file 92
Configuring the build.xml file to compile a custom MXML 94

12 Troubleshooting 95

About This Document

This document describes how to customize the Adobe® LiveCycle® Workspace ES user interface. Customizing the Workspace ES user interface includes customizations such as these:

- Changing the user interface colors, fonts, or images
- Translating the text in the Workspace ES user interface to another language other than the provided English, French, German, or Japanese languages
- Modifying the layout of the Workspace ES user interface

Who should read this document?

This document is intended for programmers who are familiar with Adobe Flex®, ActionScript™ 3.0, MXML, and Adobe Flex® Builder™ (or the Adobe Flex SDK compiler) and those who want to customize their Workspace ES user interface.

It is beneficial if you had exposure to Adobe LiveCycle ES (Enterprise Suite) Update 1 (8.2), are familiar with Workspace ES, and have a good understanding of the application server you plan to use for testing.

Before you begin

Before you can begin customizing the Workspace ES user interface, ensure that you have access to the following items:

- The Flex 3.0.1 SDK that is compatible with LiveCycle ES, Flex Builder 3.0.1, or the Eclipse Flex 3.0.1 plug-in. The Flex 3.0.1 SDK is available on the LiveCycle ES DVD.
- The LiveCycle ES SDK folder. The LiveCycle ES SDK is available from the LiveCycle ES server or from the location where Adobe LiveCycle Workbench ES is installed on your computer.
- The LiveCycle ES server from your computer. This server is required for deploying and testing the Workspace ES user interface customizations.

Additional information

The resources in this table can help you learn more about LiveCycle ES.

For information about	See
An overview of LiveCycle ES	*LiveCycle ES Overview*
The end-to-end process of creating a LiveCycle ES application that includes a form and human-centric processes, configuring services, and testing within Workspace ES	*Creating Your First LiveCycle Application*
Working with LiveCycle Workspace ES	*LiveCycle Workspace ES Help*

For information about	See
ActionScript classes and properties included with LiveCycle ES and Flex. This includes the Workspace API.	*Adobe LiveCycle ES Update 1 ActionScript Reference*
Rendering and deploying form guides using processes created in Workbench ES	*LiveCycle Workbench ES Help*
LiveCycle ES terminology	*LiveCycle ES Glossary*
Installing Flex Builder	*Flex Documentation*
Patch updates and technical notes on this product version	Adobe Technical Support

1 Introduction

LiveCycle Workspace ES is a Flex application that allows end users to initiate and participate in form-based business processes by using a web browser. Because users use Workspace ES to start and participate in automated processes, you may want to customize the user interface to be consistent with your organization's look and feel.

This document describes through tasks how to perform typical customizations. Examples illustrate typical customizations to help you understand how to customize the Workspace ES user interface and build Flex applications by reusing parts of the Workspace ES user interface and functionality.

Before you begin your customizations, it is recommended that you understand these aspects of Workspace ES:

- How files are organized in the deployed Workspace ES EAR and custom EAR files you create. (See Understanding the Workspace ES EAR file structure and Understanding the custom EAR file structure).
- How it is built and its component architecture. (See Understanding the Workspace API architecture.)
- How its components are used. (See Using Workspace API components.)
- The types of changes that are supported and described in this document. (See Understanding how to customize Workspace ES.)

You should understand the basic concepts of the Workspace API architecture and EAR files before you configure your development environment. It is recommended that you read through the following sections before performing your Workspace ES user interface customizations.

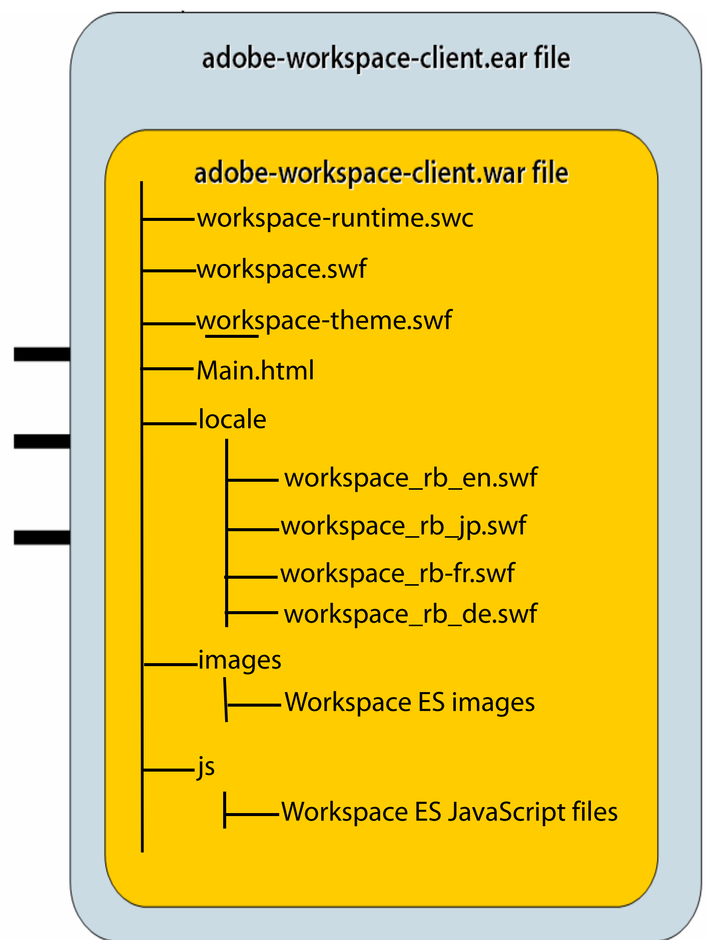
- Configuring the Development Environment - Installing the Flex SDK
- Installing Ant to Flex Builder
- Configuring the Workspace Project
- Compiling and Deploying the Workspace Project.)

After you complete the previously mentioned tasks, you can use this document to perform theme, localization, and layout customizations.

Understanding the Workspace ES EAR file structure

Workspace ES exists as an Enterprise Archive (EAR) file named adobe-workspace-client.ear that is deployed to an application server. You should not modify the adobe-workspace-client.ear file or the adobe-workspace-client.war file on your server. Instead, use the provided Workspace project to create your version of the EAR file and then deploy it to your server.

Understanding the contents of the EAR file will help you understand the customization process and how customizations are deployed. This illustration shows the contents of the adobe-workspace-client.ear file.



The adobe-workspace-client.war file contains several files and folders:

workspace-runtime.swc: The SWC file that contains the Workspace API that is necessary to access the ActionScript components that comprise Workspace ES.

workspace.swf: The main Workspace ES application, when loaded, searches for the correct localization file and the theme file to load. Modifying the workspace.swf file is not supported.

workspace-theme.swf: The theme file that contains the cascading style sheets (CSS) and any images that Workspace ES uses. The workspace.swf file dynamically loads the workspace-theme.swf file at startup.

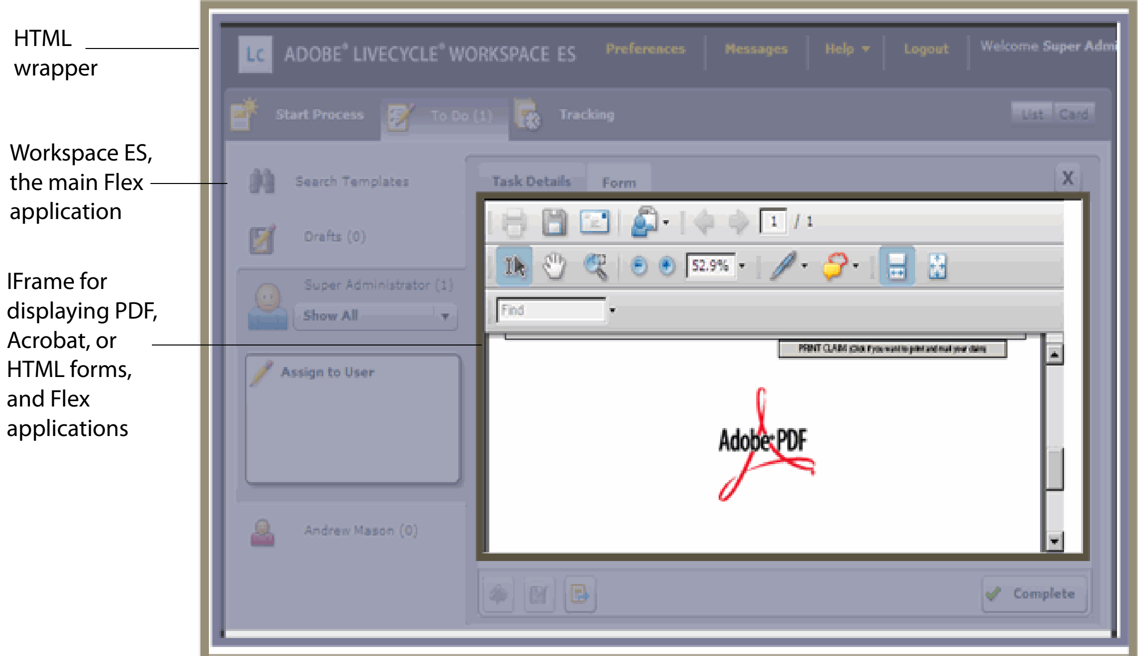
workspace_rb_*[locale]*.swf: The localization files for Workspace ES, where *[locale]* specifies the locale and region-specific settings. The appropriate localization file is dynamically loaded by the workspace.swf file. Workspace ES loads the appropriate localization file for each language in the order that is specified in the user's web browser settings. If a localization file that matches the web browser settings is not found, Workspace ES defaults to English. Workspace ES can load region-specific locales; however, if a region-specific locale is not found, the general locale is loaded instead. For example, Workspace ES loads ES if the browser settings specify ES-BR, but no matching localization is found.

Main.html: The HTML wrapper displays the main Workspace ES application and has iFrames that control the display of various forms (PDF, Acrobat, or HTML forms, and Flex applications). The Main.html

wrapper provides JavaScript™ and other support for applications that are using Workspace API components. The functionality provided is required to interact with the web browser and also provides the following functions:

- Determines the locale specified by the web browser for localization support
- Detects the version of Adobe Acrobat® or Adobe Reader®
- Controls the various iFrames required for displaying PDF forms, HTML forms, Acrobat forms, and Flex applications
- Handles the authentication of a user and manages the time-out period

This illustration shows the various parts of the Main.html wrapper.



Note: Modifying the Main.html file is not supported.

js folder: The folder of JavaScript files that the Workspace API invokes to interact with PDF, HTML, and Flex applications.

images folder: The folder of images that Workspace ES uses.

locale folder: The folder of localization files. Workspace ES is localized for English, French, German, and Japanese languages. Any localization file that you create must be added to this folder.

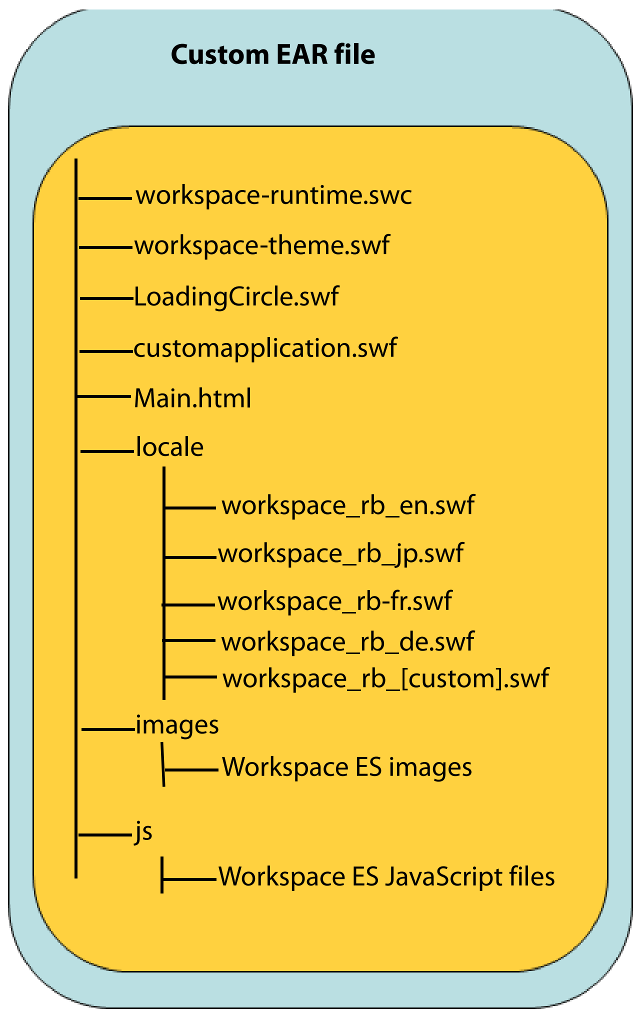
Understanding the custom EAR file structure

A Flex project is provided in the LiveCycle ES SDK and permits you to build custom EAR files for customizing the Workspace ES user interface. Using the Workspace project, you build your own instance of Workspace ES that runs on an application server.

The EAR file that you build is very similar to the EAR file that is provided for Workspace ES except that a Workspace ES SWF file is not provided. Instead, you create a Flex application by using the `lc:Desktop` component, which is the Workspace ES application provided as a Flex component. Compile the provided

Flex project against the provided adobe-workspace-runtime.swc file. The resultant EAR file is deployed on an application server.

The custom EAR file contains HTML files, SWF files, and JavaScript files, which are required to properly display a functioning Workspace ES. The items that are important to understand are shown in the following illustration.



The custom EAR file that you create for customizing the Workspace ES user interface consists of the WAR file that is created and an application.xml file. The application.xml file contains the web-uri (name of the WAR file) and the context-root of the location where the WAR is deployed to the web server.

The WAR file contains several files and folders:

js folder: The folder of JavaScript files that the Workspace API invokes to interact with PDF, HTML, and Flex applications. It is recommended that you do not modify any of these scripts.

images folder: The folder of images that Workspace ES uses. Any images that you add to your Workspace project are placed in this folder.

locale folder: The folder of localization files. Workspace ES is localized for English, French, German, and Japanese languages. Any localization file that you create is added to this folder by the build script.

.properties files: HTML files that contain text to indicate that Adobe Flash® Player is not installed on the users' web browser. Each file is for one of the localized languages provide by LiveCycle ES.

SWF files: SWF files are included in a custom EAR file. The following SWF files are important to know:

LoadingCircle.swf: The SWF file that indicates that an operation is in progress. This SWF file is included in the custom EAR file.

***[application.name]*.swf:** The main application for a custom Workspace ES application. The name of the SWF file is determined by modifying the build.xml file in the Workspace project. (See Modifying the build.xml file.)

workspace-theme.swf: The theme file that contains the CSS and any images that Workspace ES uses. The workspace.swf file dynamically loads the workspace-theme.swf file at startup. To customize the images and styles that Workspace ES uses, a new workspace-theme.swf file is created and included in the custom EAR file.

workspace_rb_*[locale]*.swf: The localization files for Workspace ES where *[locale]* specifies the locale and region-specific settings. Workspace ES dynamically loads the appropriate localization file. Each localization file that is created is stored in the locale folder.

HTML files: HTML files are included in a custom EAR file. The following HTML file is the only one that is important to know:

Main.html: The HTML wrapper displays the application. This name is the default used for all EAR files that you create. You can change this name by modifying the web.xml file located in the templates folder in the Workspace project.

Understanding the Workspace API architecture

Workspace ES is built using reusable visual and non-visual components that are exposed as the Workspace API. In Flex Builder, you may see other components that are available from the Workspace API that are not described in *LiveCycle ES Update 1 ActionScript Reference*. It is recommended that you do not use undocumented components from the Workspace API; they are not supported and are subject to change without notice in future releases.

For information about visual and non-visual components, see Using Workspace API components. The Workspace API architecture consists of three layers that are packaged as part of the workspace-runtime.swc file:

Presentation: This layer consists of visual components. Each visual component consists of a view component and presentation model component. *View components* provide the user interface for Workspace ES. *Presentation model components* store the data that will be displayed and implement the business logic by using components from the Domain Model layer. Therefore, the view and presentation model components must function as pairs.

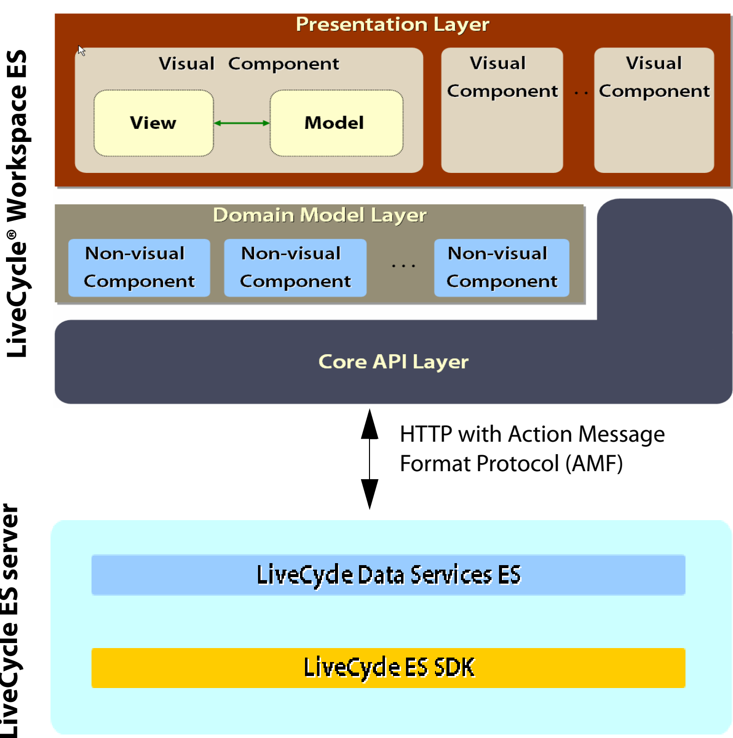
Note: Presentation model components have the same name as a view component but with the word *Model* appended as a suffix.

You can instantiate presentation model components, as necessary, to retrieve information for populating view components. Because the user interface component is separate from the component that holds the business logic and stores the data, you can code your own user interfaces to reuse the same presentation model components without needing to know the implementation details.

Domain Model: This layer consists of non-visual components that encapsulate LiveCycle ES logical business concepts for process management, such as tasks, endpoints, and queues. The components are used to implement business logic for an application. The components in this layer do not provide user interface functions. Instead, the components communicate with the Core API layer and are often used to work with LiveCycle ES business concepts to create your own user interface.

Core API: This layer consists of ActionScript components that implement the details to communicate with the LiveCycle ES server. To provide process management functions, the components in the Core API layer invoke the Task Manager Service API that is described in *Programming with LiveCycle ES*. Adobe LiveCycle Data Services ES facilitates the communication between the Workspace APIs and LiveCycle ES SDK. The messaging channel used between the Workspace API and Data Services ES is HTTP with Action Message Format (AMF) protocol, which permits Workspace ES to be dynamically updated with server changes. Workspace ES does not use LiveCycle Remoting, which requires that remote clients poll the server to receive notification changes.

The following illustration shows the layers in the Workspace API, and how AMF and Data Services ES are used to communicate with the LiveCycle ES SDK to provide process management functions. *AMF* is a binary data protocol designed to facilitate binary serialization of ActionScript objects and types.



The available Workspace API components are described in *LiveCycle ES Update 1 ActionScript Reference* and are prefixed with *lc* in the namespace. The `lc.domain` package specifically describes the components in the Domain Model layer, and the `lc.core` package describes the components in the Core API layer.

Typically, you use components in the Presentation layer to develop your own Flex application. The components from the Domain Model layer are useful for passing information between presentation layer components when you reuse the user interface components to create your Flex application. You can also use Domain Model components in situations where you intend to create your user interface and want to use the components to create your own business logic. You will seldom use components from the Core API layer; although, you may want to implement interfaces or use components to coerce other objects to retrieve information for your Flex application.

Using Workspace API components

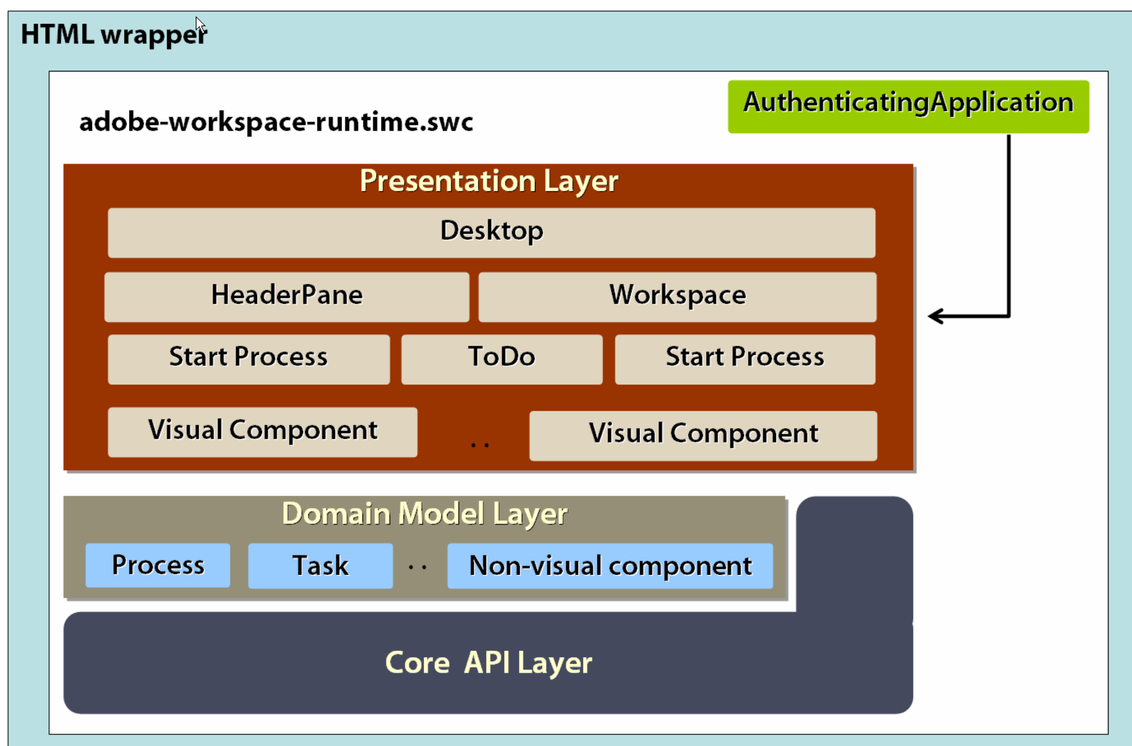
The Workspace API provides reusable visual and non-visual components written in MXML and ActionScript:

- *Visual components* are logical representations of a part of Workspace ES functionality that physically consist of a view component (typically defined in an MXML file) and a presentation model component (defined in an ActionScript file). By separating the visual components into view and presentation model components, it separates the user interface and events from the logical implementation details and behavior. This allows for a simplified view component that only needs to bind data to properties in the presentation model component. This design adheres to the Model-View-Controller (MVC) design pattern.

  Each visual component may be a grouping of one or more granular parts of Workspace ES functionality that includes both the user interface and the functionality. For example, the `lc:Desktop` component, which is the entire Workspace ES application available as a Flex component, consists of several other components, such as the `lc:Workspace` and `lc:HeaderPane` components. The `lc:Workspace` component consists of an `lc:StartProcess`, `lc:ToDo`, and `lc:Tracking` components. The `lc:StartProcess` component is composed of other more granular components that encompass both the user interface and functions such as `lc:EmbossedNavigator` and `lc:TaskInfo`, `lc:TaskForm`. Each of these visual components are used to build your own Flex application to perform a layout customization without having to be familiar with the details of the stand-alone.

- *Non-visual* components are logical representations of Workspace ES functionality but with no user interface exposed. Non-visual components encapsulate LiveCycle ES logical business concepts for process management, such as processes, tasks, endpoints, and queues.

  You can use non-visual components to provide the functionality for user interfaces that you create. The non-visual components in the Domain Model layer provide most of the functionality required for Flex applications you create.



To use any of the components in Workspace API, you must provide an authenticated session. The `lc:SessionMap` component stores an authenticated session and the information required for a Flex application to communicate with LiveCycle ES.

The `lc:SessionMap` component provides untyped access to all Workspace API objects. Almost all visual components require that you set the `session` attribute to an instance of the `lc:SessionMap` class. In your Flex application, you can access an instance of the `lc:SessionMap` class by using the `session` attribute from the `lc:AuthenticatingApplication` component. For access to an `lc:SessionMap` object from other MXML components, you can create an instance of an `lc:SessionMap` class.

It is recommended that you use the `lc:AuthenticatingApplication` component as the root tag in your default application file because the following features are automatically provided:

Dynamic loading of localization files: Lets you add a new localization file and deploy with your Flex application. It dynamically loads a matching locale based on the locale setting of the user's web browser. If no matching localization file is found, the default is English.

Dynamic loading of theme files: Lets you customize the display settings of Workspace ES, such as colors and images, in CSS that is compiled into a theme file. After you deploy the theme file with your Flex application, the display settings are automatically loaded.

Automatic authentication of the user: Ensures that the logic within your Flex application is not executed until the user is authenticated. If the session expires, the user is prompted to authenticate again without writing additional code in the Flex application.

If you choose not to use the `lc:AuthenticatingApplication` component in your Flex application, you can create an `lc:SessionManager` component and call the `authenticateUser` method. The `lc:AuthenticatingApplication` component creates an internal instance of a `lc:SessionManager` class. (See *LiveCycle ES Update 1 ActionScript Reference*.)

You also can use the `lc:SessionMap` object to access Domain Model components. Most of the components from the Domain Model layer are singletons that you access by calling the `getObject` method and providing the fully qualified name for the class. For example, to retrieve an instance of an `lc:QueuesManager` component, use the following code if you have an instance of an `lc:SessionMap` object named `mySession`:

```
var myQueue:QueuesManager;
myQueue = new QueuesManager(mySession.getObject("lc.core.QueuesManager"));
```

Understanding how to customize Workspace ES

Workspace ES is designed so that you can customize it to meet the requirements of your organization. Depending on the type of changes you make, you may need to replace files or add new files in your own copy of the adobe-workspace-runtime.ear file or even build a separate Flex application that reuses components from the Workspace API. You can customize the following aspects of Workspace ES:

Localization: Workspace ES is localized for the English, French, German, and Japanese languages. You can localize Workspace ES to another language by creating a new localization file and deploying it. (See Localization Customization - Localizing to Spanish.)

Localization files can also be used to change specific text, such as terminology, to better suite your organization.

You do not need to use the Workspace API to perform localization customizations.

Theme: Workspace ES has a CSS that specifies the colors, images, fonts, and other styles used in the user interface. You can customize the colors, images, fonts, and other styles that are available as part of the CSS specification. A basic understanding of CSS and its usage helps you to achieve the results you want.

For example, you can customize Workspace ES to brand it with your organization's logos and graphics or to modify colors to match those of your organization. (See Theme Customization - Replacing Images and Theme Customization - Modifying Colors.)

You must use the Workspace API to perform theme customization.

Layout: You can use visual components from the Workspace API to build your own Flex applications for the following purposes:

- To simplify Workspace ES by exposing only the components or functions that your organization requires. For example, you can customize Workspace ES to only start processes. (See Layout Customization - Simplifying the User Interface.)
- To change the layout of Workspace ES by building your own Flex application by using components that are available from the Workspace API. For example, you can build a three-pane view that has a pane that displays the tasks assigned to a user, a pane that describes the task details, and a pane that displays the form. (See Layout Customization - Creating a Three Pane View.)
- To replace certain components with your own. For example, you can replace the default login screen with a login screen that you create. (See Layout Customization - Creating a New Login Screen.)

User-interface redesign: Workspace ES is built by using visual-based and non-visual Workspace ES components. You can redesign and build your own user interface components as a separate Flex

application if the provided Workspace ES user interface does not meet your business requirements. It is recommended that you use components from the Domain Model layer (`lc.domain`) and model components from the Presentation layer to build a library of reusable components to provide the process management functionality for your Flex application.

Developing a completely new user interface is full Flex development, which is beyond the scope of this document. However, you may find it useful to read various sections of this document to understand how to design your own custom user interface and use various parts of the Workspace API.

Understanding the stand-alone

The stand-alone for Workspace ES is provided as a reference to help you understand how to use Workspace API components and as a basis for building your own Flex applications. You can compile the stand-alone to understand how Workspace ES is designed. (See Compiling and Deploying the Workspace Project.)

The stand-alone is required to build the adobe-workspace-client.ear file when you customize the Workspace ES user interface. Though you compile and deploy the provided stand-alone to production, any modification of the stand-alone on the a production server is not supported. Instead of modifying the stand-alone, it is recommended that you create your own subclasses by extending Workspace API components, which are described in *LiveCycle ES Update 1 ActionScript Reference*.

Upgrade considerations

When you apply a patch or upgrade your version of LiveCycle ES, Workspace ES customizations are not affected if you deployed the theme and localization in a separate EAR file by using a different context root and web URI as recommended in this document.

If you are upgrading from a previous release to LiveCycle ES Update 1 (8.2), you must recompile your customizations with the Flex 3.0.1 SDK, which is compatible with LiveCycle ES. (See Configuring the Development Environment - Installing the Flex SDK.) In general, it is recommended that you leverage any resolved product defects and additional features, and recompile your customizations with the recommended version of the Flex SDK when you upgrade to patch releases. Each time a patch is applied, the source code and workspace-runtime.swc file

Each time a patch is applied, the source code and workspace-runtime.swc file are updated in the LiveCycle ES SDK folder. To leverage any of the changes, import the updated source code and workspace-runtime.swc file from the LiveCycle ES SDK folder into your Flex project and recompile it.

2 Configuring the Development Environment - Installing the Flex SDK

You must use the same Flex SDK version (version 3.0.1) that is compatible with LiveCycle ES Update 1 (8.2) and include the necessary SWC file in your project to access the Workspace API. The steps described in this section must be performed on each computer that you use to customize LiveCycle Workspace ES.

To use the Workspace API, the Workspace ES SWC files must be configured in your build path in Flex Builder or included as part of the compile options when you use the command line. These SWC files are located in the LiveCycle ES SDK folder and let you use components, interfaces, and classes from the Workspace API. When you use Flex Builder, you simply build your project, as long as you included the workspace-runtime.swc file in your Library Path. However, when you build your Flex application using the Flex SDK, you must specify the workspace-runtime.swc file in your command line compile options each time you build.

Before you can configure your development environment, verify that the following steps are completed:

- The LiveCycle ES SDK folder is accessible and all the samples are installed. You can access the LiveCycle ES SDK folder on the LiveCycle ES server or from a computer where LiveCycle Workbench ES is installed.
- Flex Builder 3 on your own development environment is configured to work with the Flex SDK. Although using Flex Builder is not a requirement, it is recommended for use with steps that are described in this document. Alternatively, you can use the Flex SDK and command-line options to perform steps such as compiling.

Summary of steps

It is presumed that you are using Flex Builder for customizing the Workspace ES user-interface. The following summary outlines the steps to configure your computer as a development environment with Flex Builder:

1. Install the Flex SDK that is provided with LiveCycle ES. (See Installing the Flex SDK from LiveCycle ES.)

2. Configure Flex Builder to use the newly installed Flex SDK. (See Configuring Flex Builder to use the Flex SDK from LiveCycle ES.)

Installing the Flex SDK from LiveCycle ES

You must install the Flex SDK version that is compatible with LiveCycle ES. The Flex SDK version provided for LiveCycle ES includes localization SWC files and fixes that ensure that your Flex applications work properly within a LiveCycle ES environment and that provide proper localization support. The Flex SDK that is compatible with LiveCycle ES is available on the LiveCycle ES DVD.

Before you perform these steps, you must have Flex Builder 3 installed in your development environment and have access to the LiveCycle ES DVD.

➤ **To install the Flex SDK for LiveCycle ES in Flex Builder 3:**

1. Exit Flex Builder if it is running.

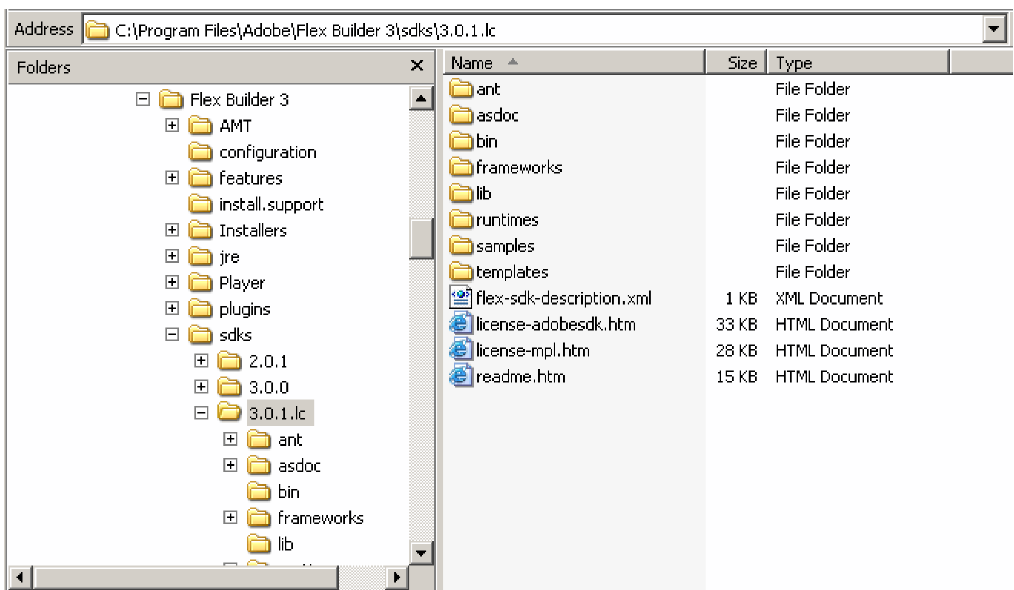
2. On the LiveCycle ES DVD, go to the livecycle_dataservices folder and copy the **flex_sdk_3.zip** file to your development computer.

3. Go to the sdks folder located in the root folder where you installed Flex Builder 3. For example, in a Microsoft® Windows® environment, go to the C:\Program Files\Adobe\Flex Builder 3\sdks folder.

4. Create a new folder, such as 3.0.1.lc, for extracting the flex_sdk_3.zip file that you copied from the LiveCycle ES DVD.

5. Using an archiving tool, extract the flex_sdk_3.zip file to the folder you created in step 4.

After you extract and install Flex SDK for LiveCycle ES, the folder you create should have the same contents as shown in the following illustration. In the illustration, the new folder is named 3.0.1.lc.



Configuring Flex Builder to use the Flex SDK from LiveCycle ES

You must configure Flex Builder to compile your Flex applications for use with Workspace ES.

➤ **To configure Flex Builder to use the Flex SDK for LiveCycle ES:**

1. Start Flex Builder.

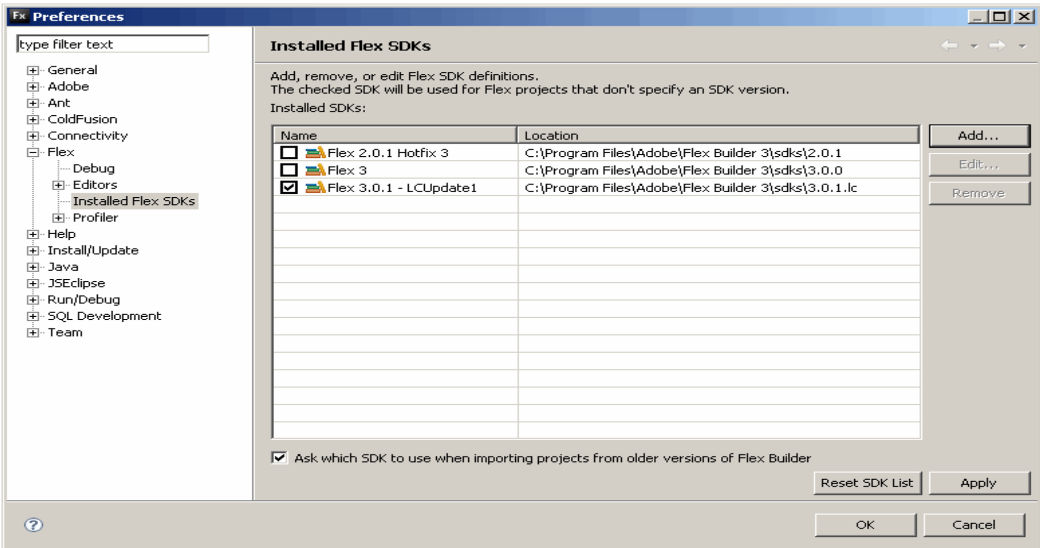
2. Select **Window** > **Preferences**.

3. In the Preferences dialog box, select **Flex** > **Installed Flex SDKs** and then click **Add**.

4. In the Add Flex SDK dialog box, perform the following steps and then click **OK**.

   - Click **Browse** and, in the Browse For Folder dialog box, go to and select the folder you created in step 4 of the procedure in Installing the Flex SDK from LiveCycle ES, and then click **OK**.
   - In the **Flex SDK name** box, type a name to reflect the specific Flex SDK version that is used for applications you develop for LiveCycle ES, such as `Flex 3.0.1 - LCUpdate1`.

5. Select the check box beside the Flex SDK you added to the list to make it the default version to use and then click **Apply**. After you perform this step, your installed list of Flex SDKs should look like the list in the following illustration.



6. In the Preferences dialog box, click **OK**.

3 Configuring the Development Environment - Installing Ant to Flex Builder

The Workspace ES stand-alone is a preconfigured Flex project that includes a set of build scripts, which require Ant and additional directives from the ant-contrib library. *Ant* is a Java-based build tool. The build scripts provide a convenient way for you to compile customizations to the user interface and package the compiled binaries to EAR files for deployment to an application server running LiveCycle ES.

When you are using Flex Builder (stand-alone), you must install both Ant and the ant-contrib library. However, when you are using the Flex Builder plug-in, you should not need to install Ant, but you may need to install the ant-contrib library (version ant-contrib-1.0b2) because Ant is usually included with the Eclipse IDE.

When you use the Flex SDK, you can install Ant and the ant-contrib library separately, and then run the build scripts from a command line by using the `ant` command.

Summary of steps

Complete these steps to install Ant to Flex Builder (stand-alone). Ant is required to use the build scripts that come with the Workspace ES stand-alone. You do not need to install Ant if you are using the Flex Builder plug-in and the Eclipse IDE.

1. Install the Ant plug-in to Flex Builder (stand-alone). (See Installing the Ant plug-in to Flex Builder.)
2. Install the Ant-contrib library. (See Installing the Ant-contrib library.)

Installing the Ant plug-in to Flex Builder

When you are using Flex Builder stand-alone instead of the plug-in version of Flex Builder, the Ant plug-in is not available. You must install Ant to use the build scripts that come with the Workspace ES stand-alone. Using the build scripts is necessary to build a custom EAR file. The EAR file is an instance of the Workspace ES application that you deploy on an application server that is running LiveCycle ES.

➤ **To Install Ant in Flex Builder:**

Note: Complete this procedure only if you are using Flex Builder stand-alone instead of the Flex Builder plug-in.

1. In Flex Builder, select **Help** > **Software Updates** > **Find and Install**.
2. In the Feature Updates dialog box, select **Search for new features to install** and click **Next**.
3. In the Update Sites to Visit dialog box, in the Sites to Include in Search pane, select **The Eclipse Project Updates**, deselect **Ignore features not applicable to this environment**, and then click **Finish**.

   Note: If you do not see **The Eclipse Project Updates** option, you must add it by performing these additional steps.

   - Click **New Remote Site**, type `The Eclipse Project Updates` in the **Name** box.
   - Type `http://update.eclipse.org/updates/3.3` in the **URL** box and click **OK**.

4. In the Update Site Mirrors dialog box, select **The Eclipse Project Updates** and click **OK**.
5. In the Search Results dialog box, select **The Eclipse Project Updates** > **Eclipse 3.3.2** > **Eclipse Java Development Tools 3.3.2**.***[version]*** check box, where *[version]* represents the current version of the plug-in, such as r33x_r20081029-7o7jE7_EDhYDiyVEnjb1pFD7ZGD7, and then click **Next**.
6. In the Feature License dialog box, accept the terms, and then click **Next**.
7. In the Installation dialog box, click **Finish** to start the download.
8. After the download completes, in the Feature Verification dialog box, click **Install All**.
9. In the Install/Update dialog box, click **Yes** to restart Flex Builder.

Installing the Ant-contrib library

The build scripts require special directives that are not part of the base Ant installation. Therefore, you must install the Ant-contrib library after you install Ant into Flex Builder. Version ant-contrib-1.0b2 is currently supported.

Caution: Version ant-contrib-1.0b3 does not work with the build scripts that are provided in the Workspace project.

➤ **To Install the Ant-contrib library:**

1. In a web browser, type `http://sourceforge.net/projects/ant-contrib/` to access the SourceForge.net site and download ant-contrib-1.0b2-bin.zip file.

   Note: To find ant-contrib-bin.zip, you must browse for older versions.
2. Extract the contents of the ZIP file that you downloaded to a folder on your computer, such as /ant.
3. In Flex Builder, select **Window** > **Preferences**.
4. Select **Ant** > **Runtime**. Ant is available only if you correctly installed Ant into Flex Builder.
5. In the Runtime properties pane, select **Ant Home Entries (Default)** and click **Add External JARs**.
6. In the Open dialog box, go to the folder that you extracted the ant-contrib-1.0b2-bin.zip file to, select **ant-contrib.jar**, and then click **Open**. The ant-contrib.jar file is located under *[your folder]*/ ant-contrib/lib, where *[your folder]* represents the folder you created in step 2.
7. When you return to the Preferences dialog box, click **Apply**, and then **OK**.

4 Configuring the Development Environment - Configuring the Workspace Project

In the LiveCycle ES SDK, a Flex project is provided to simplify the process of customizing the LiveCycle Workspace ES user interface. You must import the Flex project that is provided in the LiveCycle ES SDK to customize the Workspace ES user interface. After you import the Flex project, a project named *Workspace* appears in the Flex Navigator view. The Workspace project is packaged with build scripts that build custom EAR files to deploy a customized instance of Workspace ES; the Workspace project also includes the Workspace ES stand-alone for reference.

You must configure the provided build scripts to match the settings on your computer before you can compile the Flex project and create a custom EAR file. Deploying a custom EAR file is the recommended approach for deploying your Workspace user interface customizations in both a development and a production environment. However, in stand-alone development environments, you can choose to configure the Workspace project to deploy the compiled source directly to a web server so that you can step through the code and debug it in Flex Builder.

Summary of steps

Complete these high-level steps to configure the Workspace project. You must configure the project to match your computer settings so that you can compile a custom EAR file. The EAR file is an instance of the Workspace ES application that you deploy on an application server that is running LiveCycle ES. It is recommended that you use Flex Builder.

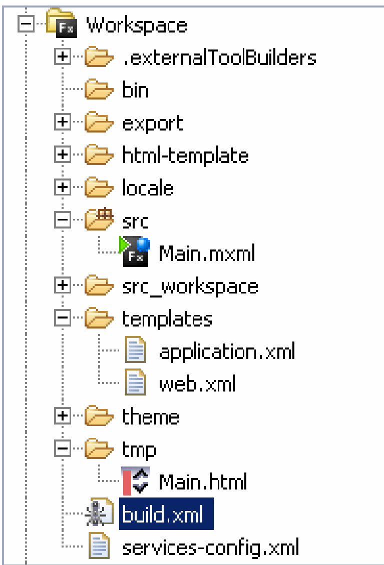
1. Configure your development environment. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Install Ant and the ant-contrib library to Flex Builder (stand-alone). (See Configuring the Development Environment - Installing Ant to Flex Builder.)
3. Import the Workspace ES project into Flex Builder. (See Importing the Workspace ES project.)
4. Modify the build.xml file to match the settings on your computer. (See Modifying the build.xml file.)
5. Create a new Ant builder for the Workspace project. (See Creating a new builder for compiling Workspace ES.)
6. Import and configure the Workspace API SWC file for the Workspace project to browse the Workspace ES stand-alone and use components from the Workspace API. (See Configuring the Flex project to browse Workspace ES stand-alone.)

Importing the Workspace ES project

The Workspace ES stand-alone is provided as an archived file in the LiveCycle ES SDK after you install LiveCycle ES. The stand-alone is provided as a Flex project that you compile and create a custom EAR file to deploy to an application server running LiveCycle ES. After you import the Flex project, a project named *Workspace* appears in the Flex Navigator view. In the Workspace project, the Workspace ES stand-alone is provided for you to understand how Workspace ES is designed and how to use the Workspace API. The provided Workspace ES source is not compiled into the EAR files that you deploy. Instead, any

customizations that you perform are compiled against the Workspace API, available from the workspace-client-runtime.swc file.

After you import the stand-alone into Flex Builder, the following folder structure appears in the Flex Navigator view.



Below are the descriptions for the folders and files in the previous illustration:

.externalToolBuilders: A folder that contains configuration files for the building environment for the project.

bin: The default folder when using the Flex Builder compiler. This folder is not used when you build using the Ant script.

export: A folder that contains the theme file, localization file, and EAR file. It is recommended that you deploy the EAR file because it contains the localization files and theme files that are compiled.

html-template: A folder that contains the html-template files, JavaScript scripts, and resources that are required for building the WAR and EAR files.

locale: A folder that contains a folder for each locale that is compiled. You can add additional folders to represent each locale you want to support.

src: A folder that contains the default Application file. In this folder, you can place custom code that uses the Workspace API for custom applications that you build.

src_workspace: A folder that contains the Workspace ES stand-alone. Do not change any files in this folder. Modifying any files and deploying the Workspace ES stand-alone to a production environment is not supported.

templates: A folder that contains the templates files that are used for creating the EAR and WAR files. You do not usually modify the files in this folder.

theme: A folder that contains an images subfolder and the CSS file that you modify to customize the images, colors, and fonts. Any custom images that you want to use in your customization must be added to the images subfolder.

tmp: A folder that contains temporary files that are created when compiling the localization files and custom EAR file.

build.xml: The file that contains the build scripts for compiling Workspace ES. Usually, you modify only the portion of the file that pertains to adapting the file to the settings on your computer.

services-config.xml: The file that contains the settings such as remote services, default locale, and polling for the Workspace ES. This file is required for compiling applications that you build by using the Workspace API and for compiling Workspace ES.

Before you import the Workspace ES stand-alone, ensure that you have access to the LiveCycle ES server that has the LiveCycle ES SDK folder installed on it. The LiveCycle ES SDK can be accessed directly on the LiveCycle ES server or on a computer where LiveCycle Workbench ES is installed and connected to the LiveCycle ES server.

Accessing from a computer where Workbench ES is installed: Go to the LiveCycle ES SDK folder in *[installdir]*\LiveCycle ES\Workbench ES\LiveCycle_ES_SDK\, where *[installdir]* represents where Workbench ES is installed on your computer.

Accessing the LiveCycle ES server directly: Go to the LiveCycle ES SDK folder in *[installdir]*\LiveCycle_ES_SDK\, where *[installdir]* represents where LiveCycle ES is installed on a server.

For example, for a JBoss turnkey installation in a Windows environment, go to the LiveCycle ES SDK folder in C:\Adobe\LiveCycle8\LiveCycle_ES_SDK\.

Note: You cannot run your Flex application locally. You must run it on the LiveCycle ES server.

Tip: To have multiple versions of the Workspace project, you can rename the project in Flex Builder.

➤ **To import stand-alone:**

1. Copy the contents of the LiveCycle ES SDK folder to a location on your computer and make note of its location. For example, you can copy it to /Adobe/LiveCycle8.2/LiveCycle_ES_SDK/.

2. In Flex Builder, select **File** > **Import** > **Flex Project**.

3. In the Import Flex Project dialog box, beside the **Archive file** box, click **Browse**.

4. In the Open dialog box, browse to the location you copied the LiveCycle ES SDK to, select the **adobe-workspace-src.zip** file, and then click **Open**. The adobe-workspace-src.zip file is located at *[LCSDK]*\misc\Process Management\ Workspace, where *[LCSDK]* represents the location on your computer where you copied the LiveCycle ES SDK folder to.

5. (Optional) To select an alternative location on your computer to store the project, deselect **Use default location**, and then click **Browse** beside the **Folder** box and select a new location on your system.

6. Click **Finish**. After the Workspace ES project is imported, a project named Workspace appears in your Flex Navigator view.

Caution: When you change the Flex SDK version in Flex Builder, the html-template may be overwritten. If this occurs, copy the html-template from the Workspace ES stand-alone to your project.

Modifying the build.xml file

The *build.xml* file contains the build scripts that are provided with the Workspace project. The following property values must be modified in the build.xml file:

flex.sdk.home: The location on your computer where you installed the Flex SDK that is compatible with LiveCycle ES, such as C:/Program Files/Adobe/Flex Builder 3/sdks/3.0.1.lc.

lc.sdk.dir: The location on your computer where you copied the LiveCycle ES SDK to.

application.title: The name that appears when Workspace ES is displayed in a web browser.

application.name: The name of the EAR file that is created and the namespace used to access the deployed application on the web server.

app.name: The name of the file that is set as the Default application in your Flex project. This value typically specifies the name of the MXML file (without the .mxml file name extension) to use as the default application. This value is important to identify the MXML file to compile your project.

➤ **To configure the build.xml file:**

1. In Flex Builder, in the Flex Navigator view, double-click the **build.xml** file to open it for editing.

2. In the editor, locate the following block of text and modify the values for **flex.sdk.home** and **lc.sdk.dir** to correspond to your development environment. Optionally, you can also modify the application.title, application.name, and app.name. The bold text below indicates the values that you can enter.

```
<!-- Modify the following properties to match your setup -->
   <property name="flex.sdk.home" location="C:/Program Files/Adobe/Flex
Builder 3/sdks/3.0.1.lc" />
  <property name="lc.sdk.dir"
location="C:/Adobe/LiveCycle8.2/LiveCycle_ES_SDK" />
  <property name="application.title" value="My Custom Application"/>
  <property name="application.name" value="myCustomApplication"/>
```

```
<property name="app.name" value="Main"/>
```

3. Select **File** > **Save** to save your changes.

Creating a new builder for compiling Workspace ES

By default, the Workspace project is configured with a builder named *Flex*. To use the build scripts provided with the Workspace project, you must create an Ant builder. After you create the new builder, the various build targets appear in the Ant view. These targets are used for compiling the Workspace project.

➤ **To create a new builder:**

1. Start Flex Builder.
1. In the Flex Navigator view, right click **Workspace** and select **Properties**.
2. Select **Builders** in the left pane.
3. In the Builders pane, click **New**.
4. In the Chose Configuration Type dialog box, select **Ant Builder** and click **OK**.
5. In the Properties for New_Builder dialog box, in the **Name** box, type a name such as `WorkspaceCustBuilder`.
6. Below the **Buildfile** box, click **Browse Workspace**.
7. In the Choose Location dialog box, specify the location of the build file:
   - In the left pane, select **Workspace**.
   - In the right pane, select the **build.xml** file and then click **OK**.
8. Below the **Base Directory** box, select the location of the base directory for the project:
   - Click **Browse Workspace**.
   - In the Folder Selection dialog box, select **Workspace** and then click **OK**.
9. Click **Apply** and then **OK**.
10. The new Builder you created should appear below Flex Builder. Confirm that the New Builder you created is selected, deselect the **default Flex Builder**, and then click **OK**.

    Note: If a Confirm Disable Builder dialog box appears, click **OK**.
11. (Optionally) Modify the display name, UI hint, and main HTML file by modifying the web.xml file located under the templates folder.

    The following text displays the default contents of the web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Adobe Workspace</display-name>
  <description>Adobe Workspace Application</description>
  <welcome-file-list>
```

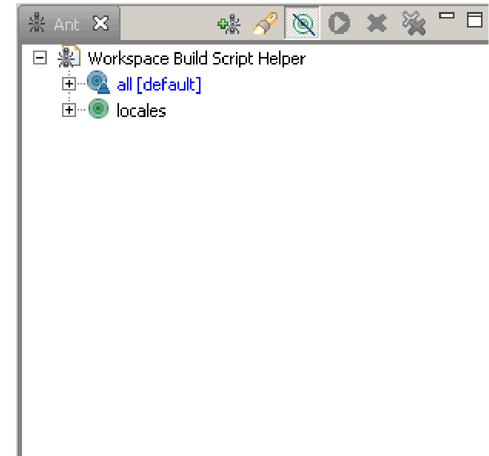
```
        <welcome-file>Main.html</welcome-file>
        <welcome-file>redirect.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

You can modify the value between the display-name, description, and first welcome-file tags. For example, you can make the following changes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Customized Workspace</display-name>
    <description>This Workspace was customized</description>
    <welcome-file-list>
        <welcome-file>mycustompage.html</welcome-file>
        <welcome-file>redirect.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

After you create a builder, you can verify that you can access the build scripts by displaying the Ant view. To display the Ant view in Flex Builder, select **Window** > **Other Views** and, in the Show View dialog box, select **Ant** and then click **OK**.

Note: You can toggle what you see in the Ant view by clicking the Hide Internal Targets button. You may need to click the Add Build Files button and select the build.xml file from the Workspace project to make the targets appear.



Tip: When Project > Build Automatically is selected, the Workspace project automatically compiles; otherwise, you can right-click the Workspace project and select Build Project. A successfully completed build indicates that you configured your Workspace project properly. You should also see the progress of the build in the Console view.

Configuring the Flex project to browse Workspace ES stand-alone

The stand-alone is not compiled with customizations that you compile and deploy to the LiveCycle ES server. After you complete the following steps, you can browse the code more easily by using the navigational functions, such as opening the source files for classes in the Workspace project by holding down the Control key and clicking the class name.

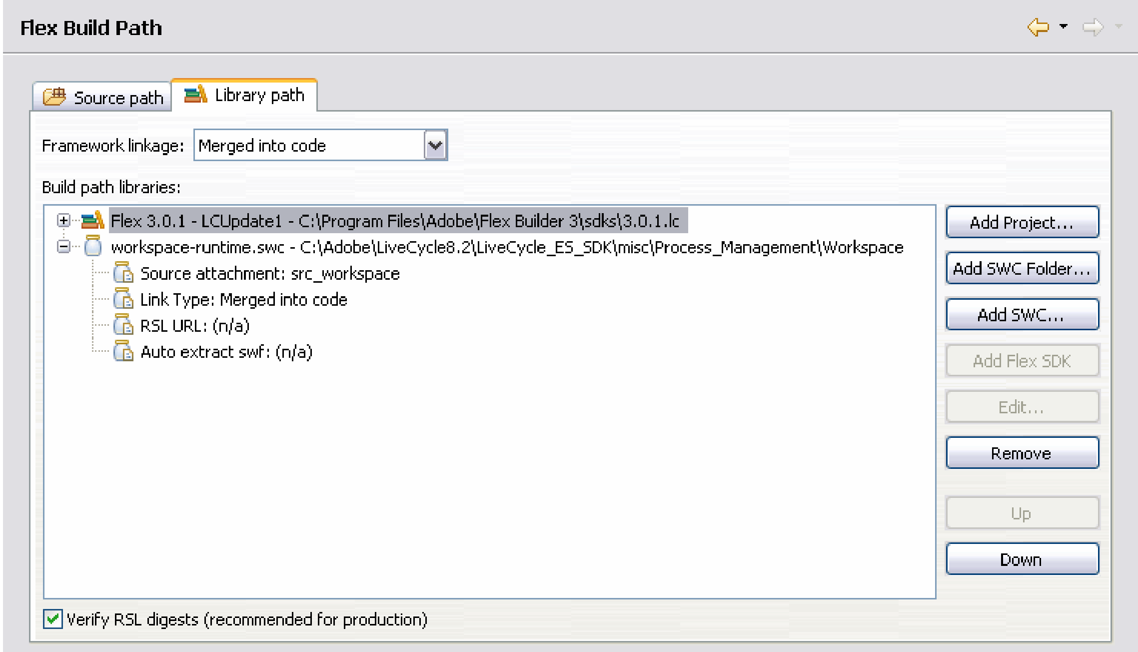
You need to configure your project to include the workspace-runtime.swc file that allows you browse the code and also provides auto completion for code when you use components from the Workspace API.

➤ **To configure the Flex project to browse the Workspace ES stand-alone:**

1. Start Flex Builder.
2. In the Flex Navigator view, right-click the **Workspace** and select **Properties**.
3. In the Properties for Workspace dialog box, in the left pane, select **Flex Build Path**.
4. In the Flex Build Path area, click the **Library Path** tab and then click **Add SWC**.
5. In the Add SWC dialog box, click **Browse**.
6. In the Choose a SWC File dialog box, navigate and select the **workspace-runtime.swc** file from the location on your computer that you copied the LiveCycle ES SDK to, click **Open**, and then click **OK**. (See step 1 in the procedure Importing the Workspace ES project.)

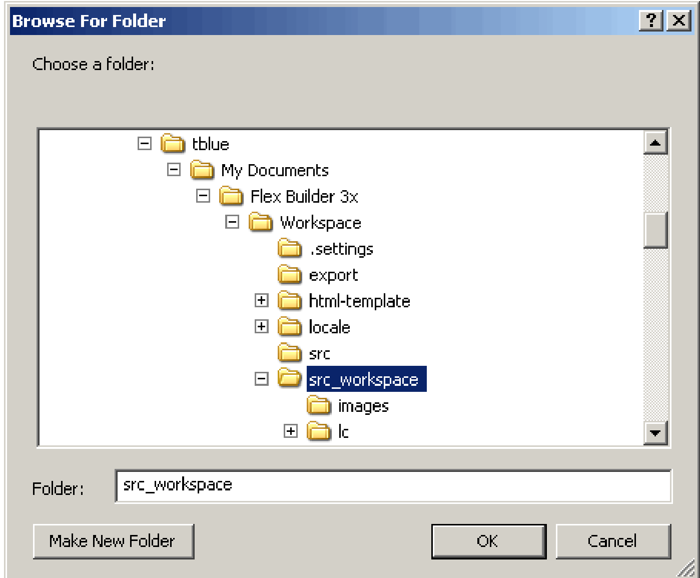
   The workspace-runtime.swc file is located in the *[LC_SDK]*/misc/Process_Management/Workspace folder where *[LC_SDK]* is the location of the LiveCycle ES SDK.

   After you complete this step, workspace-runtime.swc should appear in the build path as shown in the following illustration.



7. In the Build Path Libraries pane, click **workspace-runtime.swc** and double-click **source attachment**.
8. In the Edit Folder Path dialog box, click **Browse**.

9. In the Browse For Folder dialog box, navigate to the location on your computer where the Workspace project is located, select **src_workspace**, and then click **OK**.



Tip: The location where the Workspace project is stored where your Eclipse workspace is stored. For example, in a Windows installation of Flex Builder, it can be located at C:\Documents and Settings\tblue\My Documents\Flex Builder 3x\Workspace\src_workspace

10. When you return to the Edit Folder Path dialog box, click **OK**.

11. In the Properties for Workspace dialog box, click **OK**.

5 Compiling and Deploying the Workspace Project

After you configure your development environment, Ant in Flex Builder, and then the Workspace project, you are ready to compile the Workspace project. When you compile the Workspace project by using the provided build scripts, it creates a custom EAR file that you can deploy to the LiveCycle ES server for testing. The EAR file contains a custom Workspace ES, which is an instance of the Workspace ES application and any user interface customizations that you may have included.

The concepts described in this section are important for understanding how to compile and deploy all customizations that you perform to the Workspace ES user interface. The Workspace ES stand-alone is included as part of the Flex project for you to understand how to use the Workspace API and to understand the concepts behind the Workspace ES design. The stand-alone is not compiled into the EAR file.

Caution: When you compile Workspace ES, always use the latest version of the source code and the workspace-runtime.swc file. Each time a patch is applied, the source code and workspace-runtime.swc file are updated in the LiveCycle ES SDK folder. Import the updated source and workspace-runtime.swc file from the LiveCycle ES SDK folder before you compile.

Note: Modifying the Workspace ES stand-alone and deploying it to a production environment is not supported.

Summary of steps

You must complete these steps to compile the Workspace project and build a custom EAR file. It is recommended that you use Flex Builder to compile Workspace ES. Some of the steps below are from earlier sections in this document.

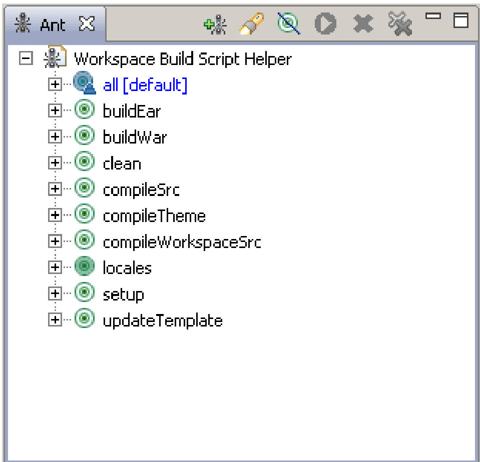
1. Configure your development environment. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Install ANT and the ant-contrib library to Flex Builder. (See Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Workspace Project.)
4. Compile Workspace ES. (See Compiling the Workspace project.)
5. Deploy your version of Workspace ES. (See Deploying a custom Workspace ES application for testing.)
6. Test your version of Workspace ES. (See Testing the Workspace ES application.)
7. (Optional) Configure your Flex project in debug mode. (See Configuring the Workspace project for debugging.)

Compiling the Workspace project

To compile the Workspace project, you must ensure that you are using an Ant builder that you created and you have your build.xml file configured to match the settings on your computer. (See Configuring the Workspace Project.)

After you create a builder, you can see the build scripts that are available to you. You should see build targets in the Ant view. You can hide the build scripts by clicking the Hide Internal Targets button.

Note: You can display the Ant view in Flex Builder from the Show View dialog box by selecting **Window** > **Other Views**.



You must run the **all[default]** target to build your EAR file and compile localization, color, or image customizations. You can also run each build script independently for the purpose of troubleshooting. The following targets are available:

clean: Deletes all the output from the export and tmp folders in the project that includes SWF files, such as localization files, the theme file, Workspace SWF file, and the WAR file and EAR files.

compileSrc: Compiles only the source file that is under the src folder in the project.

compileTheme: Compile only the workspace-theme.swf file.

compileWorkspaceSrc: Compiles only the files under the src_workspace folder in the project. The files in the src_workspace folder is the Workspace ES stand-alone that you can use as a reference. Deploying the compiled stand-alone in production is not supported.

Note: You must change the following `arg value` for the compileWorkspaceSrc target to compile the Workspace ES stand-alone:

- `${basedir}/src/namespace-manifest.xml` to `${basedir}/src_workspace/namespace-manifest.xml`
- `${basedir}/src` to `${basedir}/src_workspace`
- `${basedir}/export/${swf}` to `${basedir}/tmp/${swf}`
- `${basedir}/src/Main.mxml` to `${basedir}/src_workspace/Main.mxml`

```
<target name="compileWorkspaceSrc">
  <java taskname="mxmlc" jar="${flex.sdk.home}/lib/mxmlc.jar"
    dir="${flex.sdk.home}/frameworks" fork="true" failonerror="true">
    <jvmarg value="-Xmx512m" />
    <arg value="-headless-server=true"/>
    ...
    <arg value="${basedir}/src/namespace-manifest.xml"/>
    <arg value="-source-path"/>
    <arg value="${basedir}/src_workspace"/>
    ...
    <arg value="{basedir}/tmp/${swf}"/>
```

```
        <arg value="--"/>
        <arg value="${basedir}/src_workspace/Main.mxml"/>
      </java>
    </target>
    ...
```

locales: Creates a localization file for each locale in the project. The locales that are compiled are based on the name of the folders that you created under the locale folder in the Workspace project.

setup: Extracts the necessary files for compiling the theme and localization files.

updateTemplates: Creates the Main.html and sets the namespace and SWF file to use.

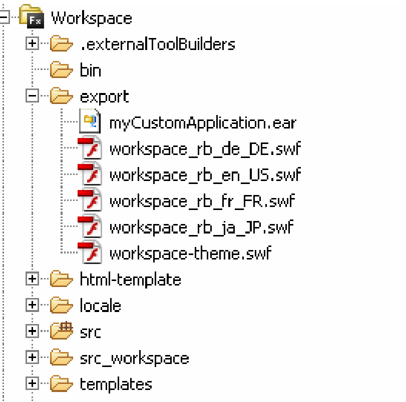
buildWar: Creates the WAR file that contains the compiled SWF file for Workspace ES, which includes the localization file, theme file, images, and related assets.

buildEar: Creates an EAR file for deployment to an application server that contains a WAR file.

➤ **To compile the Workspace project:**

- Use one of these methods to compile the Workspace project:
  - In Flex Builder, select **Project** > **Build Project** to compile the Workspace project. You must use the builder that you create to compile a custom EAR file. (See Creating a new builder for compiling Workspace ES.)
  - From the command prompt, navigate to the folder where the Workspace project is stored on your computer (where the build.xml file is located), type `ant`, and then press **Enter**. You must ensure that you have Ant configured properly on your computer.

By default, after a successful compile, the custom EAR file that contains your custom Workspace ES application is in the export folder in the Workspace project as shown in the following illustration.



Deploying a custom Workspace ES application for testing

After you compile Workspace ES, you must deploy to an application server that is running LiveCycle ES. You cannot run the Workspace project as a stand-alone Flex application. You must deploy the custom EAR file you created, which is built using the custom Ant builder you created. (See Creating a new builder for compiling Workspace ES.) The recommended method for deploying a custom Workspace ES application is a custom EAR file that is deployed to an application server. (See Deploying a custom Workspace ES application as an EAR file.)

You can also use the default builder that Flex Builder provides and deploy the compiled output directly to a web server. (See Configuring Flex Builder to deploy and run a custom Workspace ES application.) This is

useful for debugging in an development environment but should never be used in a production environment.

Deploying a custom Workspace ES application as an EAR file

To test the custom Workspace ES application, you must deploy it to an application server running LiveCycle ES. In the Workspace project configuration, a custom EAR file that contains the custom application is created in the export folder. Depending on the application server you deploy to, you may require different steps. See the documentation for the application server that you installed LiveCycle ES on.

For example, for a JBoss turnkey installation of LiveCycle ES, you perform these steps to deploy the custom EAR file:

1. On the application server, navigate to the location to deploy the EAR file. For example, for JBoss turnkey installation, navigate to *[installdir]*\jboss\server\all\ deploy\, where *[installdir]* is the location for an installation of the LiveCycle ES server.

2. In Flex Builder (or the location on your computer where your project is stored), go to the export folder and copy the custom EAR file to the deploy folder on the JBoss Application Server. The name of the EAR file depends on the value you configured for the `application.name` property in your build.xml file. (See Modifying the build.xml file.)

Note: Depending on the application server, you may need to restart it after you deploy the EAR file to access your newly deployed application

Configuring Flex Builder to deploy and run a custom Workspace ES application

You can configure Flex Builder to run the custom Workspace ES application if you can access the web server that is connected to your LiveCycle ES server. This is useful for deploying your compiled source automatically in a development environment for stepping through the code in the Flex Debugging perspective.

You may need to enable the default Flex builder provided with the Flex Builder application. You enable the builder for Flex by right-clicking the Workspace project and, in the Properties for Workspace dialog box, selecting **Builders**, selecting the **Flex** check box in the right pane, and then clicking **OK**.

➤ **To configure Flex Builder to run the Workspace project:**

1. In the Flex Navigator view, right-click **Workspace** and select **Properties**.

2. In the Properties for Workspace dialog box, select **Flex Build Path** and click the default **Source Path** tab if it is not selected.

3. In the **Output folder** box, click **Browse**.

4. Go to the location of the web server connected to your installation of LiveCycle ES. For example, in a JBoss Turnkey installation, the Tomcat web server is available at *[installdir]*/jboss/server/all/deploy/jbossweb-tomcat55.sar/ROOT.war/, where *[installdir]* is the location for an installation of the LiveCycle ES server.

5. Click **Make New Folder** to create a folder to store the output, type a name, such as `myCustomApplication`, and then click **OK**. The name of the folder you create is the context-root for the deployed application you compile.

   Note: Do not type the name `workspace` for the folder because the installed version of Workspace ES is already using it.

6. In the **Output folder URL** box, type the URL (excluding the Main.html file name extension), specifying where you will access the application on your web server, and then click **OK**. For example, type `http://`*`[servername]`*`:8080/myCustomWorkspace/`, where *`[servername]`* is the name of your computer. The context root is myCustomWorkspace.

Testing the Workspace ES application

You can test your custom version of Workspace ES by logging on to the server where LiveCycle ES is installed and deploying the EAR file to the application server.

For a JBoss turnkey installation of LiveCycle ES, you may have deployed your custom Workspace ES application as an EAR file, which deploys the WAR file to Tomcat, the JBoss turnkey web server.

To access the custom Workspace ES application, type `http://`*`[servername]`*`:`*`[port]`*`/`*`[name space]`*`/`*`[html page]`*, where

- *`[servername]`* is the name of the computer where LiveCycle ES is installed.
- *`[port]`* is the default port port for accessing the web server. For example, for a JBoss turnkey installation, the default port is 8080.
- *`[name space]`* is the context-root used to access your compiled version of Workspace ES. The name depends on the value you configured for the `application.name` property in your build.xml file. (See Modifying the build.xml file.)
- *`[html page]`* is the HTML home page. The default value is Main.html and is configured in the templates folder.

For example, if the server name is lcserver, the port is 8080, and application.name is myCustomApplication, you type `http://lcserver:8080/myCustomApplication/Main.html`.

The following table contains some recommended testing considerations for verifying that you customized your Workspace ES application properly.

Customization	Testing considerations
Theme	• Verify that any images you changed appear properly and are sized appropriately for the component. • If you modified colors, verify that their appearance is consistent. • For fonts, verify that they appear correctly and do not overlap or cross any boundaries.
Localization	• Configure the web browser to access the new locale for Internet Explorer and Firefox. (See Testing the localization file.) • Verify that message notifications are sent in the proper language. • Verify all strings in all areas are displayed properly and are not truncated. • Verify that the default language is correct when the browser settings do not find a matching locale. • Verify that text within images are not localized using the .properties file. You may need to modify the images if the text you want to modify is part of an image. • Verify that the correct language is displayed when a user is accessing Workspace ES from a browser
Layout	• Verify that, if you are using Workspace ES components, the components interact properly. • Verify that you can display PDF and HTML forms, form guides, and Flex applications properly. • Verify that each Workspace API component that you use appears visually correct and is sized properly.

Testing the localization file

After you deploy the EAR file that contains your custom localization file, you must test to ensure that your changes are properly displayed. When you modify the text in the alc_wks_client_ui.properties file, it is a good idea to verify that the strings you provided are not truncated. The localization file contains the resources that your custom Workspace ES application needs to display for the text in the user interface, error messages, and tracing messages.

The custom Workspace ES application dynamically determines the localization file to use. Each time your custom application loads within a web browser, it searches for a localization file that matches the locales that the web browser settings specify. The languages, which are selected, are based on individual web browser settings and the available SWF files in the WAR file that is packaged as part of the custom EAR file. For example, your web browser settings may specify that Spanish is the first preferred language; however, if a matching localization file is not available, the next language that is selected is based on the web browser settings. If no matching localization file is found, the default locale that is used is U.S. English (or the default locale you specified in your custom application).

Because you created your own version of the EAR file with a modified context-root, log in to your custom Workspace ES application with a different URL. For example, in a JBoss turnkey installation, if you deployed an EAR file with a context-root named *customworkspace*, you can access your custom application by typing the URL `http://`*`[servername]`*`:8080/`*`customworkspace`*/ in a web browser, where *[servername]* represents the name of the LiveCycle ES server.

Caution: To see your changes, you may need to delete localization files that are cached by your web browser.

➤ **To test the new localized SWF file using Mozilla FireFox:**

1. Start FireFox and select **Tools** > **Options**.
2. In the Options dialog box, click **Advanced**.
3. On the **General** tab, click **Choose**.
4. In the Languages dialog box, in the **Select a language to add** list, select the language that corresponds to the new localized SWF file and then click **Add**. For example, if you created a localization file for international Spanish, select **Spanish [es]**.
5. In the Languages In Order Of Preference area, click **Move Up** or **Move Down** to the location of your preference, and then click **OK**
6. In the Options dialog box, click **OK**.
7. Restart FireFox and start Workspace ES.
8. Navigate to the Workspace ES screens that you changed in the new localized SWF file and verify your localization changes.

➤ **To test the new localized SWF file using Microsoft Internet Explorer 6:**

1. Start Internet Explorer and select **Tools** > **Internet Options**.
2. On the **General** tab, click **Languages**.
3. In the Languages Preference dialog box, click **Add**.
4. In the Add Language dialog box, select the language that corresponds to the new localized SWF file and click **OK**.
5. In the Language area, select the new language, click **Move Up** or **Move Down** to modify the locale order of preference, and then click **OK**. For example, if you created a localization file for international Spanish, select **Spanish (International Sort) [es]**.
6. In the Internet Options dialog box, click **OK**.
7. Restart Internet Explorer and start Workspace ES.
8. Navigate to the Workspace ES screens that you changed in the new localized SWF file and verify localization changes.

Configuring the Workspace project for debugging

To run your custom Workspace ES application and debug it, compile a custom EAR file by using debug mode. Flex Builder can debug the deployed EAR even though it is deployed on a separate computer.

➤ **To configure the Workspace project for debugging:**

1. In Flex Builder, perform these s:

- Right-click the **Workspace** project and select **Properties**.
- In the Properties for Workspace dialog box, select **Flex Build Path**.
- In the **Main source folder** box on the **Source path** tab, type the appropriate text:
  - To debug and step through the Workspace ES stand-alone, `workspace_src`.
  - To debug and step through code that you add to the src\Main.xml, type `src`.
- Click **OK**.

2. Open the build.xml from the Workspace project.

3. In the editor, find the `compileSrc` build target, and change the argument `-debug` from `false` to a value of `true`, and then save your changes. The value to change is shown in the bold text below:

```
<target name="compileSrc">
      <java name="mxmlc" jar="${flex.sdk.home}/lib/mxmlc.jar"
dir="${flex.sdk.home}/frameworks" fork="true" failonerror="true">
          <jvmarg value="-Xmx512m" />
          <arg value="-headless-server=true"/>
          <arg value="-locale="/>
          <arg value="-debug=true"/>
          <arg
value="-library-path+=${workspace.sdk.home}/workspace-runtime.swc"/>
          <arg value="-services"/>
          <arg value="${basedir}/services-config.xml"/>
          <arg value="-o"/>
          <arg value="${basedir}/tmp/${swf}"/>
          <arg value="--"/>
          <arg value="${basedir}/src/${app.name}.mxml"/>
      </java>
   </target>
```

4. In the Flex Navigator view, right-click the **Workspace** project, and select **Build Project**.

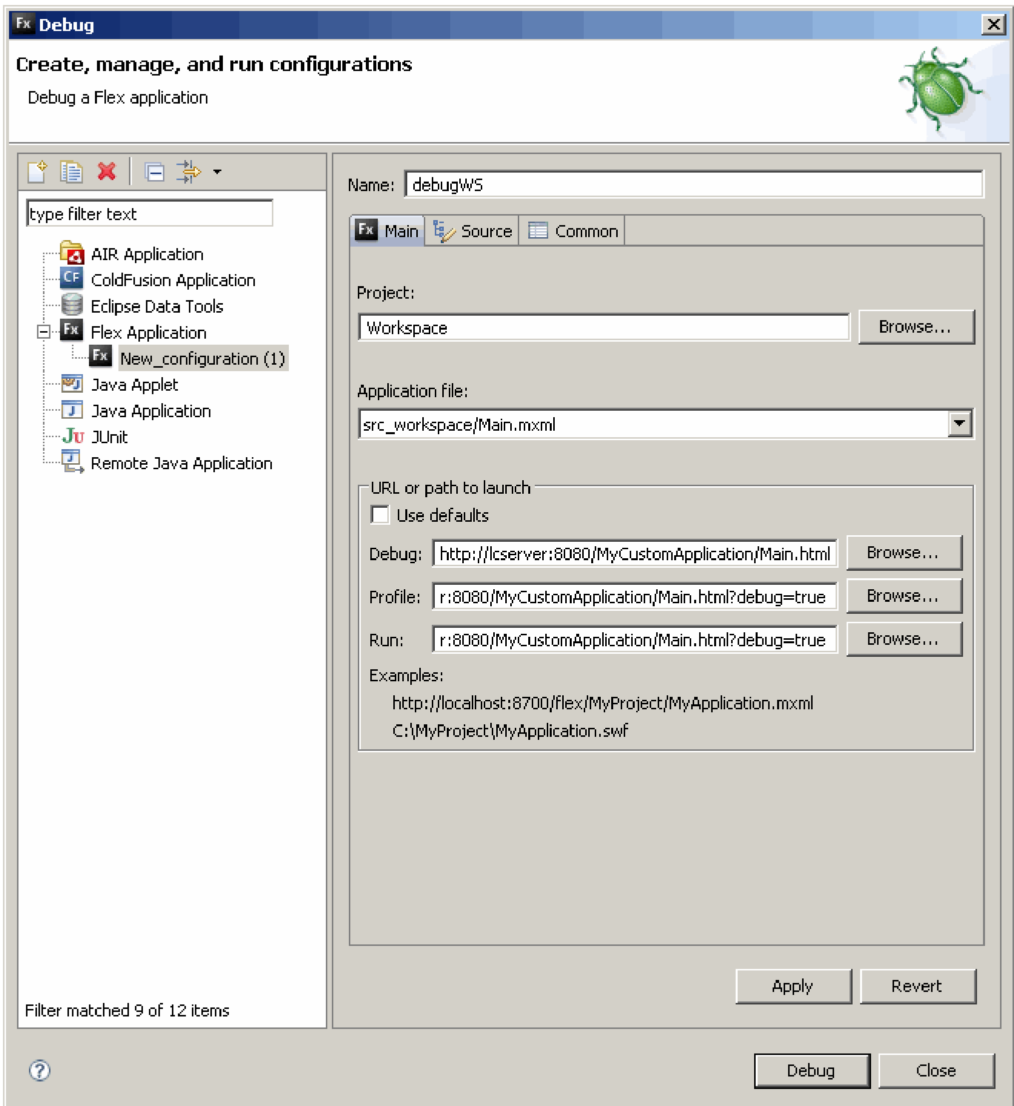
5. Deploy the EAR file from the export folder in the Workspace project. (See Deploying a custom Workspace ES application for testing.)

6. After you deploy the EAR file, in the Flex Navigator view in Flex Builder, right click **Workspace** and select **Debug As** > **Open Debug Dialog**.

7. In the Debug dialog box, select **Flex Application**.

8. Click **New** to create a new configuration and, on the **Main** tab, perform the following steps:
   - In the **Name** box, type a name for the configuration, such as **debugWS**.
   - Beside the **Project** box, click **Browse** and select the **Workspace** project.
   - In the **Application File** box, verify that the value **workspace_src\Main.mxml file** or **src\Main.mxml** is selected.
   - Under URL or Path to Launch, deselect **Use defaults** and, in the Debug, Profile, and Run boxes, type the URL to access your application on the server, appending `?debug=true` to the end of the URL. (See Testing the Workspace ES application.) For example, for a JBoss turnkey installation, type `http://lcserver:8080/MyCustomApplication/Main.html`**`?debug=true`**.

After you complete the above procedure, your Debug dialog box should look like the following illustration.



- Click **Apply** and then click **Debug**.

Note: You must have the default Flex builder disabled and you must create an Ant builder that uses the provided build.xml with the Workspace project.

The web page you typed appears. Depending on your settings, a Flash Player 9 dialog box may appear.

9. (Optional) If a Flash Player 9 dialog box appears, perform one of these steps and then click **Connect**:
   - If Flex Builder and the LiveCycle ES server are on the same computer, you select **LocalHost**.
   - If Flex Builder and the LiveCycle ES server are on separate computers, select **Other Machine** and, in the **Enter IP address** box, type the IP address of your computer.

In Flex Builder, the Console view displays debugging information. Before you start debugging, you can configure breakpoints in the code so that you can step through your code.

Compiling Workspace ES to another locale

When you compile Workspace ES to another locale, you must change the compiler options. When you are compiling, use the default builder provided by Flex Builder. This is useful when you want to debug the Workspace ES application in a different language. (See Configuring Flex Builder to deploy and run a custom Workspace ES application.) The default compiler options that are provided compile in English.

Note: Perform the following procedure only when you are compiling Workspace ES to a locale other than U.S. English.

➤ **To compile Workspace ES to anther locale:**

1. In Flex Navigator, right-click **Workspace** and select **Properties**.
2. In the left pane, click **Flex Compiler** to modify the locale option to compile Workspace ES to a language other than US English.

   Note: You must also add the localization file that you create to the html-template\locale folder in your Workspace project (see Localization Customization - Localizing to Spanish) and use proper localized versions of the Flex SDK files to match the locale that you want to compile to. The localization file contains the translated text for your locale.
3. Use the following default compile options to compile the Workspace ES using Flex Builder (stand-alone) successfully:

   ```
   -services ../services-config.xml -locale en_US -namespace
   http://www.adobe.com/2006/livecycle namespace-manifest.xml -theme lc.css
   ```

   Change `en_US` to the code that represents your locale. For example, to compile to French, type the value of `fr_FR` for the `-locale` parameter and change the compile options to this text:

   ```
   -services ../services-config.xml -locale fr_FR -namespace
   http://www.adobe.com/2006/livecycle namespace-manifest.xml -theme lc.css
   ```
4. After you make the changes, click **Apply** and then click **OK**.

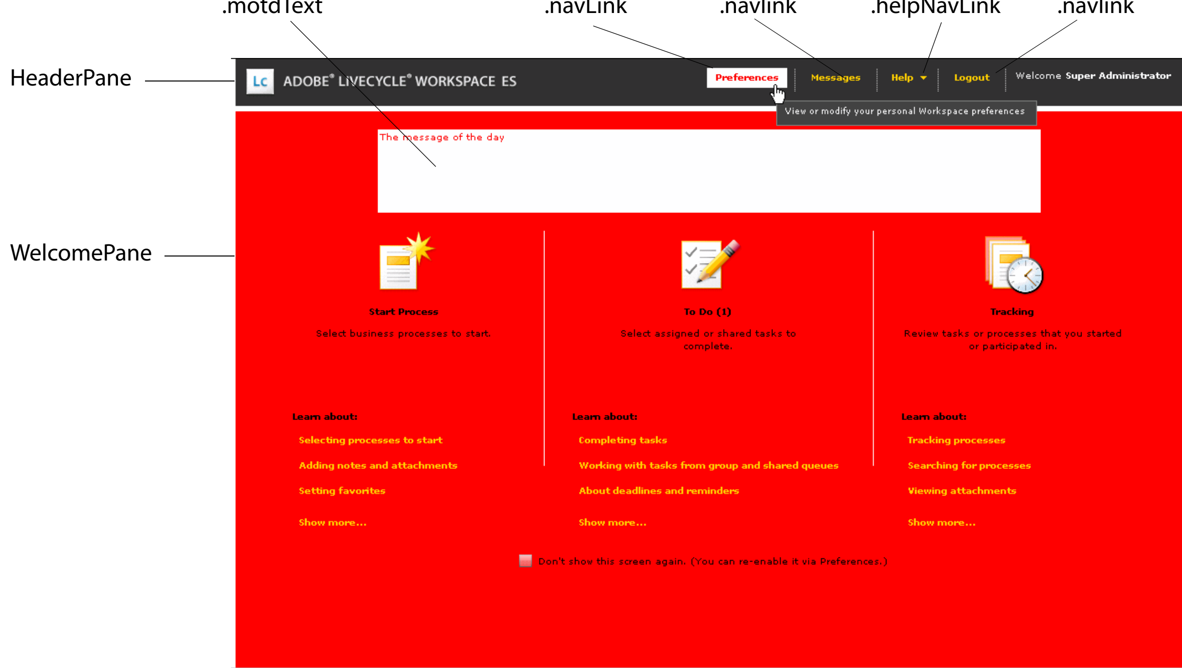
6 Theme Customization - Modifying Colors

LiveCycle Workspace ES uses a set of standard colors that are embedded within a theme file. A *theme file* contains the CSS and the images displayed within Workspace ES. This section describes how to modify the default colors.

You can modify the colors to reflect the branding of your organization or use colors that work best for your users. All colors displayed in Workspace ES are modified by changing the workspace-theme.swf theme file. The theme file allows the workspace-runtime.swf file to support embedding the run-time format of CSS. When the Workspace ES run time starts, it searches for the workspace-theme.swf file and loads it to support dynamic themes. You compile and deploy your own version of the theme file in the custom EAR file that is created when you compile the Workspace project

Workspace ES is built by using multiple visual components that are described in *LiveCycle ES Update 1 ActionScript Reference*. You can determine the styles' and components' entries to modify in the CSS file by looking at the Workspace ES stand-alone in the Workspace project under the src_workspace folder. (See Importing the Workspace ES project.) For example, to determine the components that comprise the Welcome screen, look at the Welcome.mxml file.

Each component uses the CSS `global` property, container-defined or component-specific colors. Therefore, to achieve a consistent look, you may need to change the color in the CSS file for multiple entries for various styles and classes. As shown in the illustration below, if you do not change all the relevant entries for a screen to the same color, you get a segmented appearance. In the illustration, the `lc:AuthenticatingApplication` component is the root component and is displayed as the surrounding component (white). The header kept the default color, the message-of-the-day area has changed to white, the rollover and text rollover for the `.navLink` style has changed to red text and white background, and the `lc:WelcomePane` component changed the background color to red. The illustration also shows the associated CSS properties to modify to change each component.



Summary of steps

You must complete these high-level steps to modify the colors in Workspace ES. The example that accompanies this procedure describes how to change the style and class entries in the CSS file to provide a consistent color of red.

Note: The setup, compile, and deploy steps are the same for all customizations; therefore, you may need to reference an earlier section as indicated below. Depending on your customization, the testing steps are similar.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Configure Ant in your development environment. (See Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Workspace Project.)
4. Modify CSS file colors. (See Modifying colors in the CSS.)
5. Compile the theme file into the EAR file. (See Compiling the Workspace project.)
6. Deploy the theme file into the EAR file. (See Deploying a custom Workspace ES application for testing.)
7. Test the theme customization. (See Testing the Workspace ES application.)

Modifying colors in the CSS

To modify the colors, compile a new theme file. A theme folder is provided in the Workspace project where a CSS file named lc.css is available. The theme file is compiled with all the other pieces, including the workspace-runtime.swc file to create an EAR file that you deploy on the server. Modify the entries in the lc.css file to change the colors displayed in Workspace ES.

The *lc.css* file contains the list of types of classes and style names that the visual components use. Class names refer to specific visual components that Workspace ES uses. For example, to customize the image that the ToDo component uses, modify the style for `ToDo`. Some visual components are customized by using a style name. Style names are prefixed with a period (.). You can determine the style names that are used by looking at the stand-alone; however, you can usually determine the purpose of most styles by its name.

To modify the general colors that are displayed in Workspace ES, modify the entries for the Application component and global style in the CSS file. The class name you modify is `Application` in the CSS file and the `.global` style. However, for the header on each page, it is referred to as `HeaderPane`. As you modify the colors displayed in Workspace ES, you may discover other areas to customize that you did not notice at first. For example, after you modify the general colors, you will notice that the header and an area that displays messages need to be modified. After you modify the header, you may discover that the rollover colors need to be modified. An iterative approach is required to achieve the results you want.

In general, the following CSS attributes are used to modify color:

- `backgroundColor`
- `backgroundGradientColors`
- `color`
- `textSelectedColor`

- `textRollOverColor`
- `rollOverColor`
- `selectionColor`

Caution: Depending on the colors you choose, you can make some text invisible. For example, if you choose white as the background color, the white text displayed below each image on the Welcome screen is no longer visible.

➤ To change the colors of a screen:

1. Open Flex Builder and the Workspace project.
2. In the Flex Navigator view, select the theme folder, and then open the lc.css file for editing.
3. Locate and modify the appropriate attributes for the classes and styles. You need to determine which components need to modified.

   For example, you may want to change the color of the Welcome screen to red. To give it a consistent appearance, you need to modify the `Application` container, the `HeaderPane` component, and the `WelcomePane` component. Therefore, you need to make changes to the following classes or styles in the lc.css file:

   `Application` class: `backgroundGradientColors` attribute from a value of `#333333, #333333` to `#B22222,#B22222`

   `HeaderPane` class: `backgroundColor` attribute from a value of `#333333` to `#B22222`

   `WelcomePane` class: `backgroundColor` attribute from a value of `#333333` to `#B22222`

   `.motdText` style: `backgroundColor` attribute from a value of `#333333` to `#B22222`

4. Save the lc.css file.

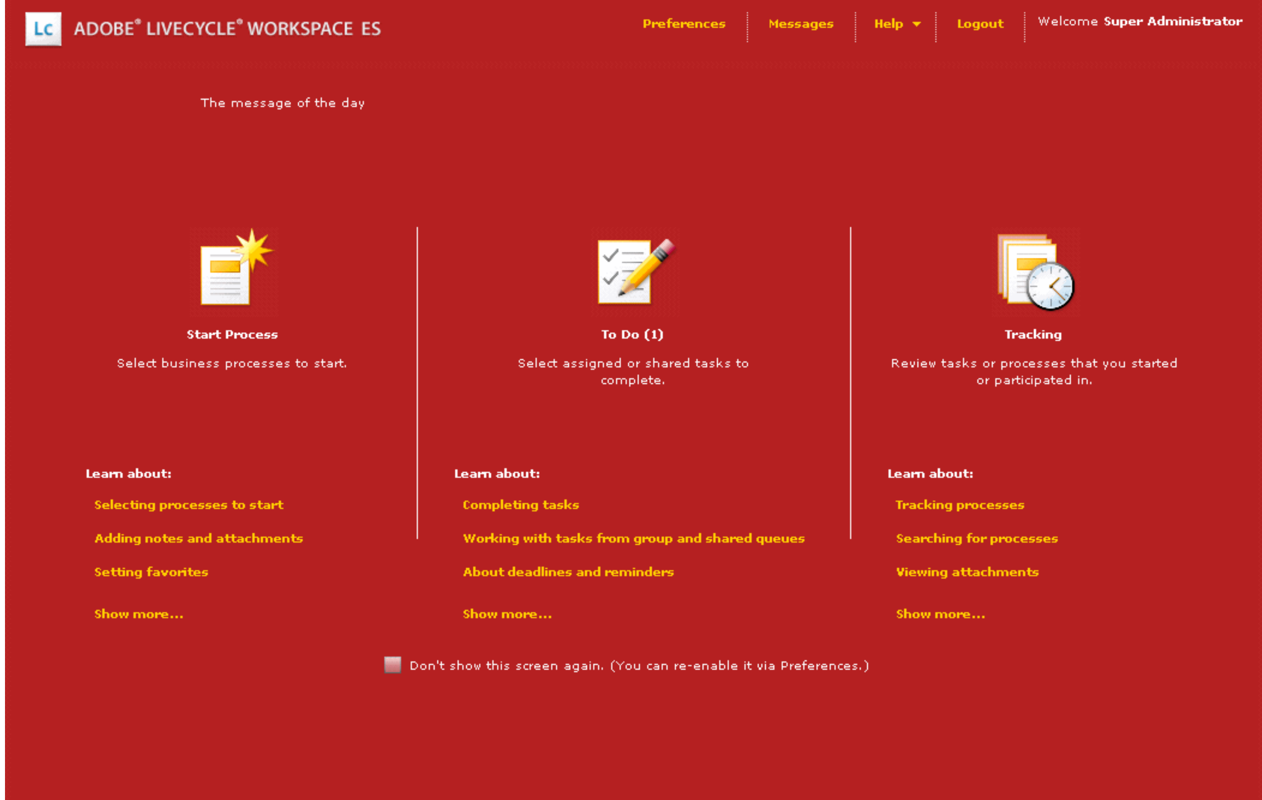
You changed the basic colors for the Welcome screen. The rollover colors for some of the links remain the original colors. You can modify these colors to give Workspace ES the same appearance.

Example: ***Changing the colors of the Welcome screen to red in the lc.css file***

```
...
...
Application {
   backgroundGradientColors:  #B22222, #B22222;
   backgroundColor:           #ededed;
}
...
...
HeaderPane {
   logo:                      Embed('images/corp-logo.png');
   color:                     #ffffff;
   backgroundColor:           #B22222;
}
...
...
WelcomePane {
   color:                     #ffffff;
   textRollOverColor:         #ff0000;
   textSelectedColor:         #ff0000;
```

```
   rollOverColor:              #ffffff;
   selectionColor:             #ffffff;
   backgroundColor:            #B22222;
}
...
...
.motdText {
   color:        #ffffff;
   backgroundColor: #B22222;
   borderThickness:"0";
}
...
...
```

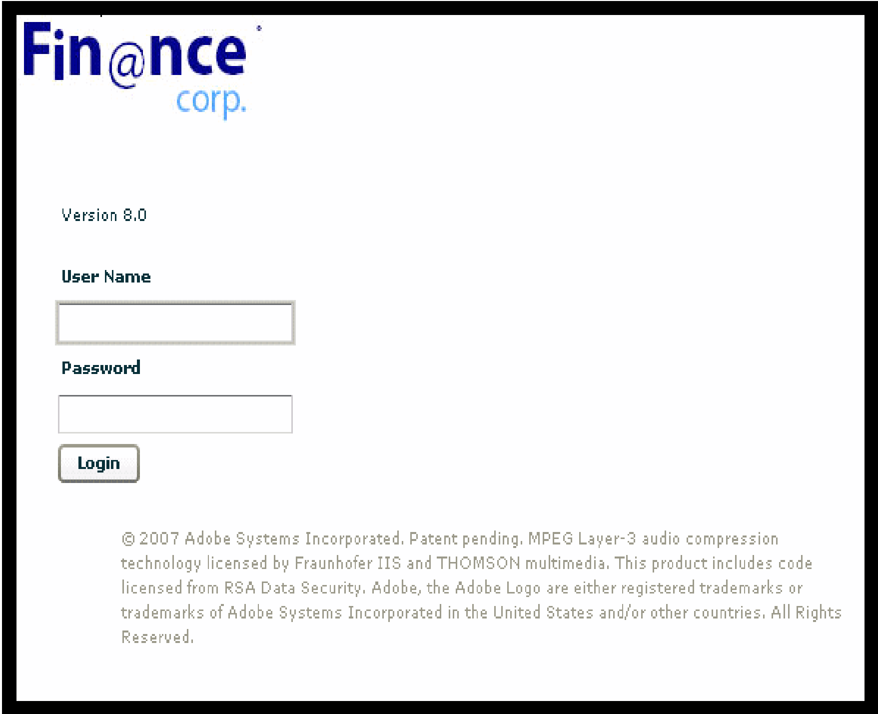
After you compile your EAR file and deploy it the LiveCycle ES server, you can test it by accessing the server from a web browser. When you are testing your changes, it is recommended that you also verify that the screens look consistent. For example, if you changed the background color to red, none of the screen sections or components should be using the original color. If you modified the colors as described in the example, your Welcome screen should look like this illustration.



Note: To view your changes, you may need to delete the online and offline files that are cached by your web browser.

7 Theme Customization - Replacing Images

LiveCycle Workspace ES is displayed with default images that are embedded within a theme file. A *theme file* contains a CSS and the images that are displayed within Workspace ES. This section describes how to replace images in Workspace ES with your own images. Most of the images in Workspace ES can be replaced by modifying the workspace-theme.swf theme file. The theme file allows the workspace-runtime.swf file to support embedding the run-time format of CSS. When a custom Workspace ES application starts, it searches for the workspace-theme.swf file and loads it to support dynamic themes. You can compile and deploy your own version of the theme file for your custom Workspace ES application to display the images you want. For example, you can display the logo for your organization, as shown in this illustration, instead of the default Workspace ES image for the logon page.



Summary of steps

You must complete these high-level steps to replace an existing image with a custom image. The example that accompanies these steps describes how to modify the images that are displayed on the login screen and banner (`lc:HeaderPane` component).

Note: Because the setup, compile, and deploy steps are the same for all customizations, you may need to refer to an earlier section as indicated below. Depending on your customization, the testing steps are similar.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Configure Ant in your development environment. (See Configuring the Development Environment - Installing Ant to Flex Builder.)

3. Import and configure the Workspace project. (See Configuring the Development Environment - Configuring the Workspace Project.)

4. Add image files to your project. (See Adding custom images to the project.)

5. Change the appropriate areas in the CSS file to use the image. (See Modifying the CSS to replace images.)

6. Compile the theme file. (See Compiling the Workspace project.)

7. Deploy the theme file. (See Deploying a custom Workspace ES application for testing.)

8. Test the theme file changes. (See Testing the Workspace ES application.)

Adding custom images to the project

You can add custom images to the theme > images folder in the Workspace project. The images are compiled into the new theme file. It is recommended that you scale the images to the same size as the image you plan to replace in order to provide a consistent appearance in Workspace ES

Note: It is recommended that you do not overwrite existing images that are provided with Workspace ES; otherwise, it will be difficult to revert to a previous image.

➤ **To add an image to your project:**

- Either drag your JPG, GIF, or PNG image file directly to the images folder or copy it to the location on your computer.

Modifying the CSS to replace images

After you copy the image to the lc/images folder in the Workspace project, you can modify the CSS file called lc.css. Modify the entries in the lc.css file to reference images that you want displayed in Workspace ES. The lc.css file is compiled with the images into a theme file that the Workspace ES application uses by at run time.

The *lc.css* file contains the types of classes and style names that the visual components use. Class names refer to specific visual components that Workspace ES uses. For example, to replace the image that the `lc:ToDo` component uses, modify the style for `ToDo`.You can determine the style names that are used by looking at the stand-alone; however, you can usually determine the purpose of most styles by its name.

Therefore, for example, to modify the login screen (which uses the `.loginSplash` style), change the `backgroundImage` property to reference a different file. By default, it references a file called login_splash.png, as shown in the following snippet from the lc.css file:

```
.loginSplash {
  backgroundImage: Embed('images/login_splash.png');
}
```

Tip: You can determine the CSS property to modify by navigating to the images folder that you copied to your project, view the images in a web browser, and find the name of the file that matches the image you want to modify. Then, in the lc.css file, use that name as a search key to find the CSS property to modify.

Depending on the image, you may also want to modify the colors to provide a consistent appearance in Workspace ES. (See Theme Customization - Modifying Colors.)

Caution: Most images that are displayed in Workspace ES can be replaced by changing the lc.css file. The following images cannot be replaced with Workspace ES because they are not embedded in the lc.css file.

Images that are not directly referenced in the CSS are dynamically loaded at run time from the images folder by the Workspace ES application and can be modified without recompiling the workspace-theme.swf file.

➤ **To modify the login screen and header pane logos:**

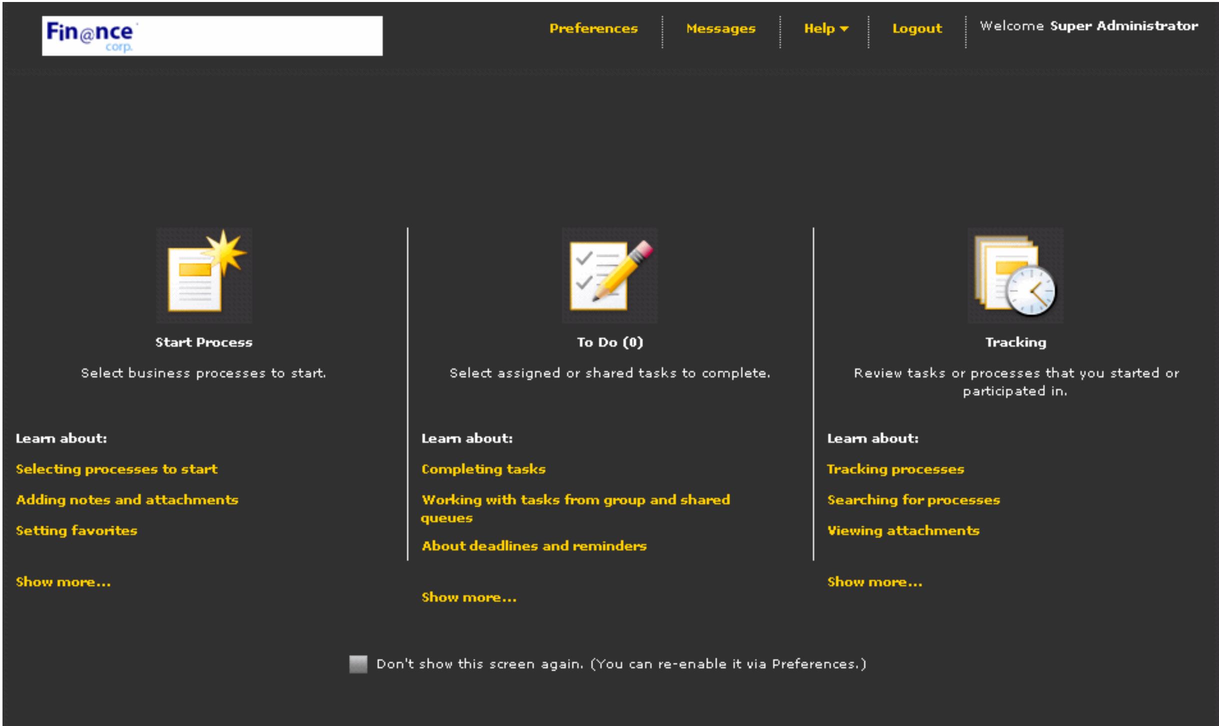
1. Open Flex Builder and the Workspace project.
2. In the Flex Navigator view, select the **theme** folder, and then open the **lc.css** file for editing.
3. Find the style name or class name to modify. For example, to change the logo on the screen and the main header, modify the `backgroundImage` attribute for the `.login` style name and the `logo` attribute for the `HeaderPane` class name.

   Note: To determine the CSS style to modify, look through the stand-alone.
4. Type the name of the new image you copied to your project to replace the existing image name. The property you modify is specific to each class or style name. For example, for the `.login` style name, modify the `backgroundImage` property; however, for the `HeaderPane` class name, modify the `logo` property.
5. Save the lc.css file.

Example: ***Modifying the images used for the login screen and header pane logos***

```
.headerLogo {
  icon:        Embed('images/financeCorpBanner.png');
  rollOverColor:#333333;
  selectionColor:#333333;
}

...
/*
---------------------------------------------------------------------------
   The following section describes the Workspace class selectors.

---------------------------------------------------------------------------
*/
.copyright {
  color:       #999988;
}

.loginSplash {
  backgroundImage: Embed('images/financeCorpLogo.png');

}
```

After you build and create a custom EAR file, deploy it to the LiveCycle ES server and test it by accessing the server from a web browser. When testing your changes, it is recommended that you also verify that the screens look consistent. For example, if you changed the background color to red, none of the screen sections or components should be using the original color. If you modified the images as described in this section, your Welcome screen should look like this illustration.

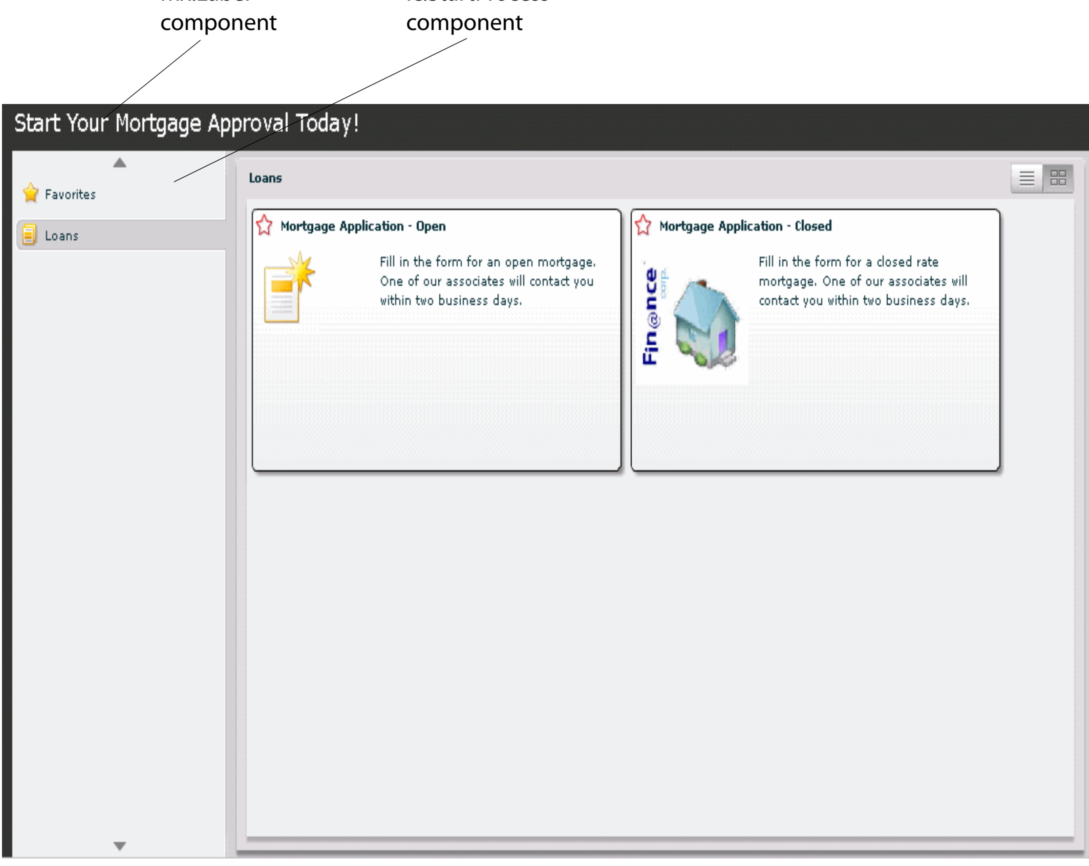


8 Layout Customization - Simplifying the User Interface

LiveCycle Workspace ES is built using self-contained visual components and non-visual components collectively called the *Workspace API*. The Workspace API provides process management functions as described in *LiveCycle ES Update 1 ActionScript Reference*. Various Workspace ES components expose different parts of the Workspace ES user interface and its functionality. Some components provide all the functions that are required for a specific task, including the user interface. This section describes how to reuse one Workspace ES component to build a simplified version of Workspace ES in a few simple steps.

For example, your organization may have users who only start processes and do not participate in business processes. These users can be customer service representatives who start business processes in your company or even external users, such as applicants requiring loans. You may not want these users to see the To Do tab.

You can create a separate Flex application that provides users access to only a limited portion of the Workspace ES user interface to start the processes. In this case, you must create a Flex application that uses the `lc:StartProcess` component, which contains all the functionality and user interface necessary to start a process in LiveCycle ES. The following illustration shows an `mx:Label` component and `lc:StartProcess` component in a simple Flex application that is used to start an approval process for a mortgage.



Summary of steps

You must complete these high-level steps to create a Flex application to build a simplified version of the Workspace ES user interface and functionality. The example that accompanies these steps describes how to use a Workspace API component to create a Flex application that starts processes.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Configure Ant in your development environment. (See Configuring the Development Environment - Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Development Environment - Configuring the Workspace Project.)
4. Create the application logic. (See Creating the application logic for a simplified Workspace ES.)
5. Modify the build.xml file to compile your files. (SeeConfiguring the build.xml file to compile a custom MXML.)
6. Compile the Workspace project file. (See Compiling the Workspace project.)
7. Deploy the EAR file. (See Deploying a custom Workspace ES application for testing.)
8. Test the localization changes. (See Testing the Workspace ES application.)

Creating the application logic for a simplified Workspace ES

After you create and configure your Flex project, you are ready to create the application logic by using the Workspace API. By default, a Flex project is created with an `mx:Application` container. The Workspace API provides the `lc:AuthenticatingApplication` container, which works in a similar manner to the `mx:Application` container, except it handles the display of the Workspace ES screen when necessary and stores the authenticated session information that is required to communicate with the LiveCycle ES server.

You create the application for a simplified Workspace ES by completing the following procedure:

➤ **To create the application logic:**

1. In Flex Builder, in the Workspace project, right-click **src** and select **File > MXML Application**.
2. In the New MXML Application dialog box, in the **Filename** box, type a name for your application, such as `CustSimpleStart.mxml`.
3. In the MXML file you created in the previous step, in the default `<mx:Application>` tag, add the LiveCycle ES namespace by typing `xmlns:lc="http://www.adobe.com/2006/livecycle"` as an attribute to use the Workspace API.
4. Replace the `<mx:Application>` tag with `lc:AuthenticatingApplication`.

5. To add instructions to the screen, perform these s:
   - Add an `mx:VBox` container.
   - Under the opening `<mx:VBox>` tag, add an `mx:Label` component.
6. After the `<mx:Label>` closing tag, add an `lc:StartProcess` component and set the `session` attribute to bind to the `session` property from the `lc:AuthenticatingApplication` object. The `lc:StartProcess` component encapsulates the functionality and user interface for starting a process.
7. Select **File** > **Save** to save the MXML application file.

Example: ***An application that only starts a process***

```
<lc:AuthenticatingApplication
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:lc="http://www.adobe.com/2006/livecycle"
  paddingBottom="5" paddingLeft="5" paddingRight="5" paddingTop="5"
  horizontalAlign="center" verticalAlign="middle"
  width="100%" height="100%" layout="absolute"
  horizontalScrollPolicy="off" verticalScrollPolicy="off">

  <mx:VBox id= "myID" width="100%" height="100%" x="10">
    <mx:Label text="Start Your LiveCycle ES Processes here!" color="white"
fontSize="18"/>
    <lc:StartProcess session="{session}" height="100%" width="100%"/>
  </mx:VBox>
</lc:AuthenticatingApplication>
```

Configuring the build.xml file to compile a custom MXML

Because you created your own MXML file, you can configure the build.xml file in the Workspace project to compile your MXML file as the default application instead of the default Main.mxml file. Typically, in Flex Builder, in your project, you right-click the MXML file and select Set as Default Application. However, because build scripts are used, you must modify the build.xml file. Before you perform these steps, ensure that you created an MXML file.

➤ **To configure the build.xml file to compile a custom MXML file:**

1. In the Workspace project, double-click the **build.xml** file to open it for editing.
2. Make the following changes to the build.xml file:
   - For the `application.name` property, change the value to `CustSimpleWS`.
   - For the `application.title` property, change the value to `Simple Customization WS`.
   - For the `app.name` property, change the `value` to the name of the MXML file that you created in Creating the application logic for a simplified Workspace ES. For example, if you created an MXML file named CustSimpleStart.mxml file, change the value to `CustSimpleStart`.

```
<!-- Modify the following properties to match your setup -->
<property   name="flex.sdk.home"
            location="C:/Program Files/Adobe/Flex Builder 3/sdks/3.0.1.lc"
/>
<property name="lc.sdk.dir"
```

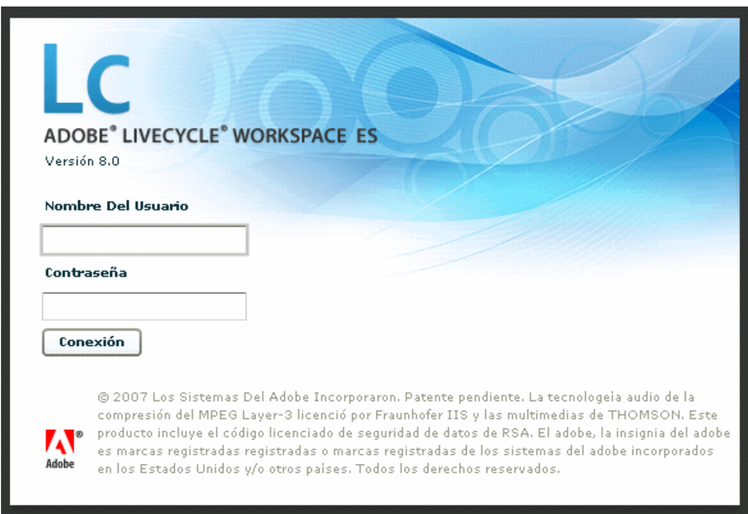
```
 location="C:/Adobe/LiveCycle8.2/LiveCycle_ES_SDK" />
<property name="application.title" value="Simple Customization WS"/>
<property name="application.name" value="CustSimpleWS"/>
<property name="app.name" value="CustSimpleStart"/>
```

3. Save your changes.

After you compile and deploy your changes, your simplified version of the Workspace ES application should appear. For example, to access the newly deployed simplified Workspace ES application in a JBoss turnkey installation, in a web browser, type `http://`*`[servername]`*`:8080/CustSimpleWS`, where *`[servername]`* is the name of your computer. The context root is `CustSimpleWS`.

9 Localization Customization - Localizing to Spanish

This section describes how to localize LiveCycle Workspace ES (end-user client application) to another language, such as Spanish or Dutch, and describes how the LiveCycle Workspace ES Help can be localized. For example, you can customize the Workspace ES screen so that the text for the copyright, user name, and password are displayed in Spanish, as shown in the following illustration.



Workspace ES is localized for the English (US), French, German, and Japanese languages. The following languages are also translated for various portions of LiveCycle Data Services ES and Flex SDK to more easily localize Workspace ES.

- Danish
- Spanish
- Finnish
- Italian
- Korean
- Norwegian Bokmål
- Dutch
- Brazilian Portuguese
- Swedish
- Simplified Chinese
- Traditional Chinese

The portions of the Flex SDK that Data Services ES localizes are required to localize Workspace ES user-interface or a version of the language. For example, you can have different versions of Spanish, such as Chilean Spanish.

Note: When you localize Workspace ES, only the user interface is localized. Any forms (PDF, XML) or Flex applications that are used in processes must be localized as a separate activity. For the message of

the day, you can localize the text in Workspace ES Administration, which is available in the LiveCycle Administration Console.

Caution: The queuesharing.swf file is not localized as part of this customization, and the source is not available for you to customize.

In Workspace ES, support for the English, French, German, and Japanese languages is provided in individual localization files that store the specific strings for a language. Each *localization file* is a SWF file that is named workspace_rb_*[locale]*.swf, where *[locale]* is the locale code for a specific language. For example, the localization file for Japanese is named workspace_rb_jp.swf.

The localization text is loaded from localization files that are part of the custom EAR file. The custom Workspace ES application dynamically loads the appropriate localization file based on the locale specified in the user's web browser settings. When multiple locales are specified, Workspace ES searches for a matching localization file based on the order of the language preference that is specified in the web browser settings. When no matching localization file is found, a default localization file is used, which is configured to be English.

You can provide support for additional languages by creating a new localization file and deploying it as part your custom EAR file. However, due to Flash Player limitations, you can only localize languages that are displayed from left to right and from top to bottom.

Tip: When localizing Workspace ES to another language, you may also want to localize the text that is embedded in an image. (See Theme Customization - Replacing Images.)

Tip: Localization files can also be used to change specific text, such as terminology, to better suit your organization. For example, to modify strings in English, you can modify the corresponding folder for en_US under the locale folder in the Workspace project.

Summary of steps

You must complete the following high-level steps to compile a new localization file to display different languages in Workspace ES. The example that accompanies these steps describes how to modify and change Workspace ES to another locale, such as Spanish.

Note: The setup, compile, and deploy steps are the same for all customizations; therefore, you may need to refer to an earlier section as indicated below. Depending on your customization, the testing steps are similar.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Configure Ant in your development environment. (See Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Workspace Project.)
4. Add a new locale folder in the Workspace project. (See Adding a new folder to create a new localization file.)
5. Localize the .properties file in the Workspace project. (See Modifying the Workspace ES properties files.)
6. (Optional) Localize LiveCycle Workspace ES Help. (See Localizing LiveCycle Workspace ES Help.)
7. Create an HTML file in case the user does not have Flash Player installed. (See Creating an error file.)

8. Configure the Workspace project to support the new locale. (See Configuring support for the new locale.)

9. Compile the Workspace project file. (See Compiling the Workspace project.)

10. Deploy the EAR file. (See Deploying a custom Workspace ES application for testing.)

11. Test the localization changes. (See Testing the Workspace ES application.)

Adding a new folder to create a new localization file

For a localization customization, you must create a new folder under the locale folder in the Workspace project and copy .properties files that represent various user-interface strings. The folder name must be locale code that is formatted as *[language code]_[country code]* where *[language code]* is a two-letter code that represents the language you want to localize to, and *[country code]* is a two-letter code that specifies the variant of the language. The build script that is provided with the Workspace project creates a new localization based on the name of the folder you create.

LiveCycle ES provides translated libraries for the various Flex and Data Services messages that you may receive. You must create a file that specifies which language you want to base your localization customization on. The following languages are translated and are listed with their locale codes in addition to en_US (US English).

Locale code	Language	Locale code	Language
da_DK	Danish	Ko_KR	Korean
de_DE	German	nb_NO	Norwegian Bokmål
es_ES	Spanish	nl_NL	Dutch
fi_FI	Finnish	pt_BR	Brazilian Portuguese
fr_FR	French	sv_SE	Swedish
it_IT	Italian	zh_CN	Simplified Chinese
ja_JP	Japanese	zh_TW	Traditional Chinese

➤ **To add a new folder to a new localization file:**

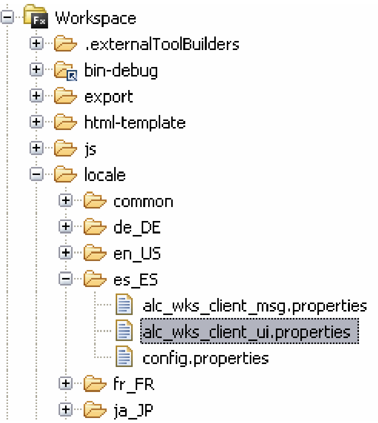
1. In Flex Builder, in the Flex Navigator view, right-click **Workspace** > **locale** and select **New** > **Folder**.

2. In the **Folder name** box, type the locale code, such as `es_ES` for Spanish, and click **Finish**.

3. Right-click the folder you created in step 2, and select **New** > **File**.

4. From one of the existing locale folders, such as en_US or fr_FR, copy the alc_wks_client.msg.properties and alc_wks_ui.properties files to the folder you created in step 2.

   The locale folder you choose depends on which language best suites your translation workflow. For example, if you translate from English to Spanish, copy the .properties files from the en_US locale folder. Your new folder should look like the following illustration:



5. (Optional). Perform these steps only if you are creating a localization file based on an existing language. Do not perform these steps when you create a localization file based on one of the existing fifteen locale codes. For example, if you were localizing to Chilean Spanish, you base your localization customization on Spanish and specify es_CL. However, if you are basing your customization on es_ES, the Spanish locale code (or any other existing locale code), do not create a config.properties file.

   - Right-click the folder you created in step 2, and select **New** > **File**.
   - In the **File name** box, type `config.properties`, and then click **Finish**.
   - Double-click the config.properties file to open it for editing, type `based.on=` *`[locale]`* where *`[locale]`* is the locale code, and then save the changes. For example, if you were localizing to Chilean Spanish, you base your localization customization on Spanish, and the local code would be `based.on=es_ES`.

   Note: The `based.on` value that you provide must be one of the fifteen locale codes, including en_US, that are provided at the beginning of this section.

Modifying the Workspace ES properties files

After creating a project, you must translate the strings within the properties files that Workspace ES uses to to display error messages and text in the user interface. These properties files are used to create new localization files.

In each properties file, a list of keys are paired with each text string. The keys are used internally by Workspace ES to determine which string to display in the user interface. Each pair is structured in the format `[key] = [string]`.

`[key]`: An internal value that Workspace ES and the Workspace API use. You must not modify this value.

`[string]`: A string that consists of the text you want to display to a user and token values, which are surrounded by braces ({}). The *tokens* are zero-based representations of the value that is inserted at run time. If a string contains these tokens, you must include them in your localized text.

The following properties files are available for Workspace ES:

alc_wks_client_msg.properties: Workspace ES uses this file to display error messages from the server to the user. You can modify this file to localize any of the messages. For example, the following key and string are used to display a login error message:

```
ALC-WKS-007-001 = Invalid user name or password.
```

alc_wks_client_ui.properties: Workspace ES uses this file to display the text for a button, field, or tab within the user interface. You can determine which string maps to which key by looking at the Workspace ES user interface. For example, the screen uses the following keys:

```
# login
login.build=Build Number 8.0.0000.000000.0
login.legal=\u00A9 2007 Adobe Systems Incorporated. Patent pending. MPEG
Layer-3 audio compression technology licensed by Fraunhofer IIS and
THOMSON multimedia. This product includes code licensed from RSA Data
Security. Adobe, the Adobe Logo are either registered trademarks or
trademarks of Adobe Systems Incorporated in the United States and/or other
countries. All Rights Reserved.
login.password=Password
login.submit-label=Login
login.username=User Name
login.version=Version 8.0
```

Caution: Do not modify or delete any of the keys in the properties files; only the strings that are associated with a key can be modified. The keys are critical for the operation of Workspace ES.

Tip: For special characters, use the unicode values. For example, when *password* is translated from English to Spanish, it becomes *contraseña*. The letter *ñ* must be represented by the unicode value of \u00F1 to appear with the proper accent in a web browser. In the properties file, you must type `Contrase\u00F1a`. You can also use also use Java's native2ascii translation tool to translate letters to unicode values.

➤ **To modify a properties file:**

1. Open the alc_wks_client_msg.properties file and find the following key and string:

   ```
   ALC-WKS-007-001 = Invalid user name or password
   ```

2. Replace the text `Invalid user name or password` with the new localized text.

3. Save the properties file.

Example: ***Changing the failed login error message to another language in the alc_wks_client_msg.properties file***

```
ALC-WKS-007-001 = [Translated text, such as Spanish, for an invalid name or
password message]
```

➤ **To modify the text that appears on the screen:**

1. Open the alc_wks_client_ui.properties file, find the `# login` area, and change the strings for the following keys to the localized text:

   - `login.build`
   - `login.legal`
   - `login.password`

- `login.submit-label`
- `login.username`
- `login.version`

2. Save the properties file.

Example: *Changing the copyright and text to another language in the alc_wks_client_ui.properties file*

```
# login
login.build=[Translated text, such as Spanish, for the build number label]
login.legal=[Translated text, such as Spanish, for the legal information]
login.password=[Translated, such as Spanish, for password label]
login.submit-label=[Translated text, such as Spanish, for login label]
login.username=[Translated text, such as Spanish, for user name label]
login.version=[Translated text, such as Spanish,for version label]
```

Localizing LiveCycle Workspace ES Help

When you localize Workspace ES, you can also localize the Workspace ES Help. The Workspace ES help files are stored in a separate EAR file called workspace_help.ear. The EAR file is located on the LiveCycle ES server and is deployed on your application server.

You can create your own version of the help, localize the contents of the LiveCycle Workspace ES Help and put them into your own version of the help, and then deploy the help onto the LiveCycle ES server. You can do this by building your own EAR file that uses the workspace-help ear file as a template.

Caution: It is not recommended you localize the HTML files directly. You should instead build your own version of the help. Localizing the HTML files directly causes the indexing in the help to not function.

➤ **To localize the Workspace ES Help:**

1. Create a folder on your computer, such as a *EarBuild*, and copy the workspace_help.ear file to your computer. The workspace-help.ear file is located on the LiveCycle ES in the *[installdir]*\deploy folder where *[installdir]* is the folder where LiveCycle ES is installed.

2. Navigate to the folder that you copied the workspace-help.ear file to in step1, and perform the following steps:
   - Extract the EAR file by using an archiving utility.

     For example, in the command prompt, navigate to the EarBuild folder and type `jar -xvf workspace_help.ear`. After you enter the `jar` command, you will see a META-INF folder, application.xml file, and four WAR files. Each WAR file represents the help files for each language
   - Delete the workspace_help.ear file from the folder. You no longer require it because you will be creating a new EAR file.

3. Create another folder to store the contents from the WAR file that you will use for localizing the Workspace ES Help content.

   Note: The folder you create must be a separate folder that is not located under the folder you created in step 1.

4. Create your own help system. You can copy the help contents using any tool provided the output is HTML and can be accessed from a web server. The contents of the help system are bundled into a WAR file that is deployed on the web server that is connected to LiveCycle ES.

   - Go to the folder you created in step 2 and, using an archiving tool, extract the contents of the WAR file.

     For example, in the command prompt, go to the folder and type the following command:

     ```
     jar -xvf workspace_help_en.war
     ```

     After you enter the `jar` command, a number of folders appear, such as JavaScript script, META_INF, and HTML files. This is the Workspace ES Help, which is a collection of HTML files.

   - Delete the WAR file you copied to the folder. You no longer require it because you will create a new WAR file.

   - Delete all the extracted files with the exception of the WEB_INF folder and META_INF folders. You will replace the help system contents with your own help system.

5. In the folder where you removed all files in the previous step, add your localized help system. Your help system should function on a web server as a stand-alone system and must include an index.html file. It is recommended that you include any script files or images in this folder.After you localize the Help text in the HTML file, perform the following steps to complete the edits to the files:

   - In the WEB_INF folder, edit the web.xml file and change all references of `workspace_help_`*`[locale]`*, where *`[locale]`* is the language of the file that you are localizing from to match the localization code that you are localizing to, and then save the file.

     For example, if you are localizing to Spanish using the HTML files within the workspace_help_en.war file, in the web.xml file, change each of the `workspace_help_en` entries to `workspace_help_es` as shown in the following code:

     ```
     <?xml version="1.0"?>
     <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
               2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
     <web-app id="WebApp_workspace_help_es">
        <display-name>workspace_help_es;</display-name>
        <welcome-file-list>
           <welcome-file>index.html</welcome-file>
     <!-- <welcome-file>additionalrootfile.htm</welcome-file> -->
        </welcome-file-list>
     </web-app>
     ```

   - In the WEB_INF folder, edit the ibm-web-bnd.xml file and change all references of `workspace_help_`*`[locale]`*, where *`[locale]`* is the language of the file that you are localizing from to match the localization code that you are localizing to, and then save the file.

     For example, if you are localizing to Spanish the HTML files within the workspace_help_en.war file, in the ibm-web-bnd.xml file, change each of the `workspace_help_en` entries to `workspace_help_es` as shown in the following code:

     ```
     <?xml version="1.0" encoding="UTF-8"?>
     <com.ibm.ejs.models.base.bindings.webappbnd:WebAppBinding
          xmi:version="2.0"
          xmlns:xmi="http://www.omg.org/XMI"
          xmlns:com.ibm.ejs.models.base.bindings.webappbnd="webappbnd.xmi"
          xmi:id="WebAppBinding_$GET_GLOBAL(BindingID);">
        <webapp href="WEB-INF/web.xml#WebApp_workspace_help_es"/>
     </com.ibm.ejs.models.base.bindings.webappbnd:WebAppBinding>
     <?xml version="1.0" encoding="UTF-8"?>
     ```

Note: The ibm-web-bnd.xml file is used only when the Workspace ES Help is deployed on a WebSphere Application Server.

6. In the folder where you created your localized help system in step 5, create a WAR File by using an archiving utility and name the file workspace_help_*[locale]*.war, where *[locale]* is the localization code.

   For example, if you are localizing to Spanish, name the WAR file workspace_help_es.war, navigate to the folder that stores the files you localized and the XML files you modified in the previous step and, in the command prompt, type the following command:

   ```
   jar.exe -cvfm workspace_help_es.war META-INF\MANIFEST.MF *
   ```

7. Copy the WAR file that you created in step 6 to the folder that you extracted the contents of the workspace_help.ear file to in step 2.

8. In the same folder, delete all the old WAR files, except for the one you copied and replaced in the folder from the previous step.

9. Go to the folder that stores the contents of the workspace_help.ear file that you created in step 2, and perform the following steps:

   - Edit the application.xml file located under the META-INF folder.
   - In the application.xml file, delete the `<module>` tag and its children except for the first entry.
   - Modify the `<web_uri>` and `<context-root>` tags to reference the WAR file that you created in step 6 and put in place a unique value in the context-root to extract the localized Help files on the web server.

   For example, for a WAR file to localize to Spanish, your application.xml file would look like the following illustration:

   ```
   <!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
   Application 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

   <application id="LiveCycle_Workspace_ES_Help">
     <display-name>LiveCycle Workspace ES Help</display-name>
       <module id="Workspace_ES_Help">
         <web>
           <web-uri>workspace_help_es.war</web-uri>
           <context-root>/workspace_help_es</context-root>
         </web>
       </module>
   </application>
   ```

10. In the folder where you stored the EAR contents, create an EAR file by using an archiving utility and name the file workspace_help_*[uniqueid]*.ear where *[uniqueid]* is a unique identifier for the EAR file.

    For example, in the command prompt, type the following command:

    ```
    jar.exe -cvfm workspace_help_myCustom.ear META-INF\MANIFEST.MF *
    ```

11. Copy the EAR File that you created in the previous step and deploy it to the application server that runs LiveCycle ES.

    For example, on a JBoss Turnkey installation of LiveCycle ES, you can copy the workspace_help_myCustom.ear file to the JBoss deploy server at /Adobe/LiveCycle8.2/jboss/server/all/deploy.

Creating an error file

You can create an error file to notify users when Flash Player is not installed on their web browsers. The error file is an HTML file that contains text for the language that you are localizing to, with a message about Adobe Flash Player not being installed.

➤ **To create an error file:**

1. In Flex Builder, in the Flex Navigator view, navigate to **Workspace** > **html-template**.
2. Right-click one of the files that is prefixed with **alc_wks_client_html** and select **Copy**. You should choose the file with the localization code that is suitable for you to translate from. For example, if you are translating from English to Spanish, right-click and copy the **alc_wks_client_html_en.properties** file.
3. Right-click **html-template** and select **Paste**.
4. In the Name Conflict dialog box that appears, in the box, type `alc_wks_client_html_`*`[locale]`*`.properties`, where *`[locale]`* is the localization code of the language you want to localize to, and then click **OK**. For example, if you are localizing to Spanish, type `es`.
5. Double-click the name of the new file to edit the contents. The new file was created under the html-template folder.
6. Translate the text to the right of the equal sign, with the exception of the {0} token, to the language you are localizing to. For example, if you are translating to Spanish, replace the text as follows:

```
browser.document.title=Adobe LiveCycle Workspace ES
no.flash.player=[Translated to Spanish] {0}.
flash.here.linktext=[Translated to Spanish]
```

7. Save your file.

Configuring support for the new locale

To add support for the locale you create, you must modify the workspace-config.js file. You must add the new locale file that you want to support and a localized version of the Help system. Optionally, you can also change the default locale to use if the user's web browser settings do not specify a supported locale. The default configured language is en_US.

➤ **To configure support for the new locale:**

1. In Flex Builder, in the Flex Navigator view, click **Workspace** > **html-template** > **js**.
2. Double-click **workspace-config.js** to open the file for editing.
3. For your new locale, type the following text immediately after the first locale entry. Each supported locale entry is delimited by a comma:

```
:[locale_country]: {flex: "locale/workspace_rb[locale_country].swf", html:
"alc_wks_client_html_[locale].properties", help:
```

```
"/workspace_help_[locale]" )},
```

where `[locale code]` is the language code and country code of the language you are localizing to and the variant of the language, respectively.

For example, if you are localizing to Spanish and the locale code is `es` and the country code is `ES`, type the text that follows in bold type. This example shows a localized version of the Workspace ES Help in Spanish that is deployed as a separate EAR file. (See Localizing LiveCycle Workspace ES Help.)

```
var supportedLocales =
{
     de_DE:  { flex: "locale/workspace_rb_de_DE.swf", html:
       "alc_wks_client_html_de.properties", help: "/workspace_help_de" },
     es_ES:  { flex: "locale/workspace_rb_es_ES.swf", html:
       "alc_wks_client_html_es.properties", help: "/workspace_help_es" },
     en_US:  { flex: "locale/workspace_rb_en_US.swf", html:
       "alc_wks_client_html_en.properties", help: "/workspace_help_en" },
     fr_FR:  { flex: "locale/workspace_rb_fr_FR.swf", html:
       "alc_wks_client_html_fr.properties", help: "/workspace_help_fr" },
     ja_JP:  { flex: "locale/workspace_rb_ja_JP.swf", html:
       "alc_wks_client_html_ja.properties", help: "/workspace_help_ja" }
};
```

4. (Optional) Change the locale to use when none of the locales that are specified by a user's browser match any of the locale codes. The locale you specify must exist, and the default is English (US).

```
var fallbackLocale = "en_US";
```

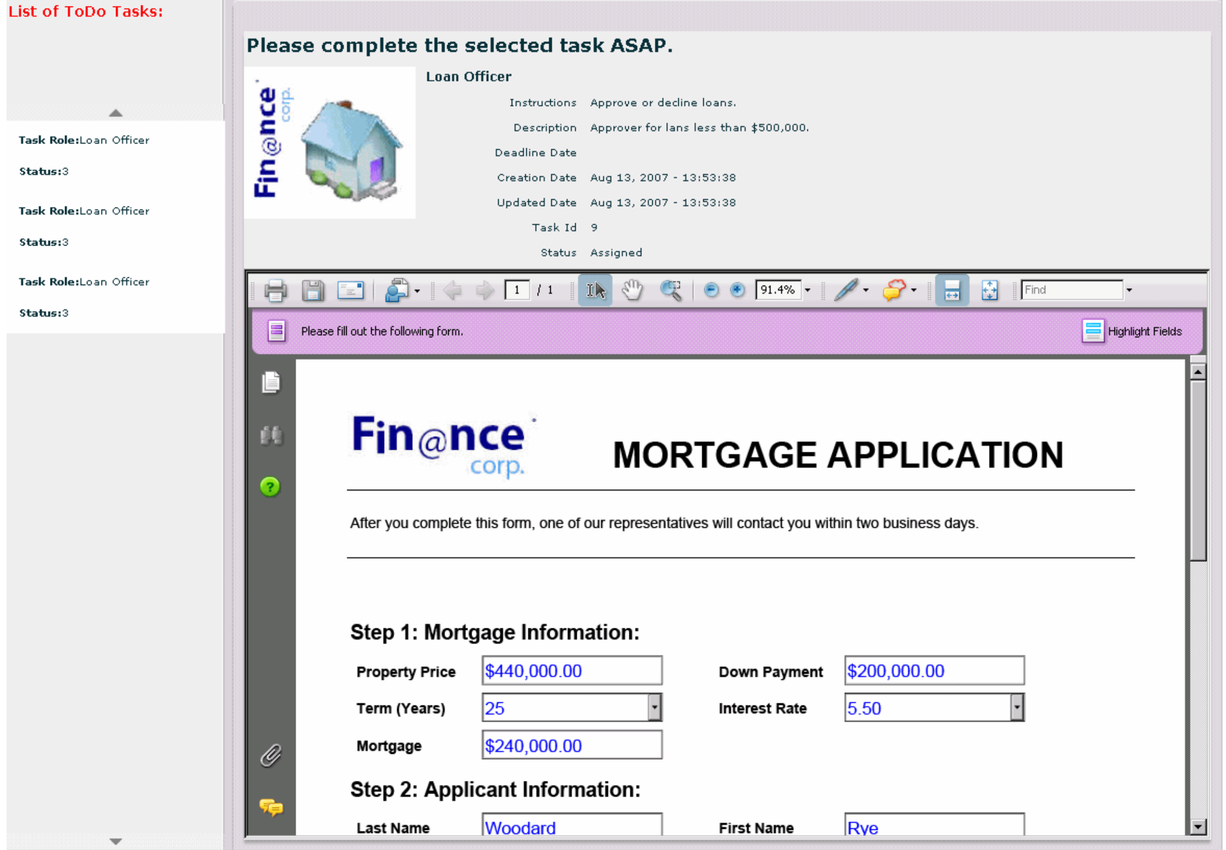
After you compile and deploy your changes, the localization customization is displayed in the custom Workspace ES application.

10 Layout Customization - Creating a Three Pane View

LiveCycle Workspace ES is built using self-contained visual components and non-visual components, collectively called the *Workspace API*, which provides process management functions as described in *LiveCycle ES Update 1 ActionScript Reference*.

This chapter describes how to use multiple visual and non-visual Workspace ES components to build your own Flex application to create a new layout in Workspace ES. You may want to create a different layout from the default Workspace ES. Organizations may create custom applications that have a similar appearance to applications that their users are familiar with to leverage their users' existing knowledge and reduce training costs.

For example, you may require only the task management function of Workspace ES because your users do not start processes. The `lc:ToDo` component's user interface is not what your users are accustomed to or want; however, they are accustomed to using email systems that display a three-pane view. You can build your own Flex application by using the Workspace API to create a three-pane view that shows a user's assigned tasks, task image and details, and the form that is associated with the selected task, as shown in the following illustration.



Summary of steps

You must complete these high-level tasks to create a Flex application that uses Workspace API components to create a custom layout. The example that accompanies these tasks describes how to use a

combination of components from the Presentation layer (both view and presentation model components), Domain Model layer, and Core API layer to create a three-pane interface. The three-pane interface consists of an area to navigate the tasks that are assigned to a user and a content area that shows the details and associated form for the selected task.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)
2. Configure Ant in your development environment. (See Configuring the Development Environment - Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Development Environment - Configuring the Workspace Project.)
4. Create the application logic. You may need to create custom view and presentation model components. (See Creating the application logic for a custom layout.)
5. Compile the Workspace project file. (See Compiling the Workspace project.)
6. Deploy the EAR file. (See Deploying a custom Workspace ES application for testing.)
7. Test the localization changes. (See Testing the Workspace ES application.)

Creating the application logic for a custom layout

Flex projects are created with a default application file that contains an empty `mx:Application` container. To use the Workspace API, replace the `<mx:Application>` tag with the `lc:AuthenticatingApplication` container. The `lc:AuthenticatingApplication` component functions similar to the `mx:Application` container, except it displays the Workspace ES login screen, as necessary, and stores the session information that is passed to other Workspace ES components.

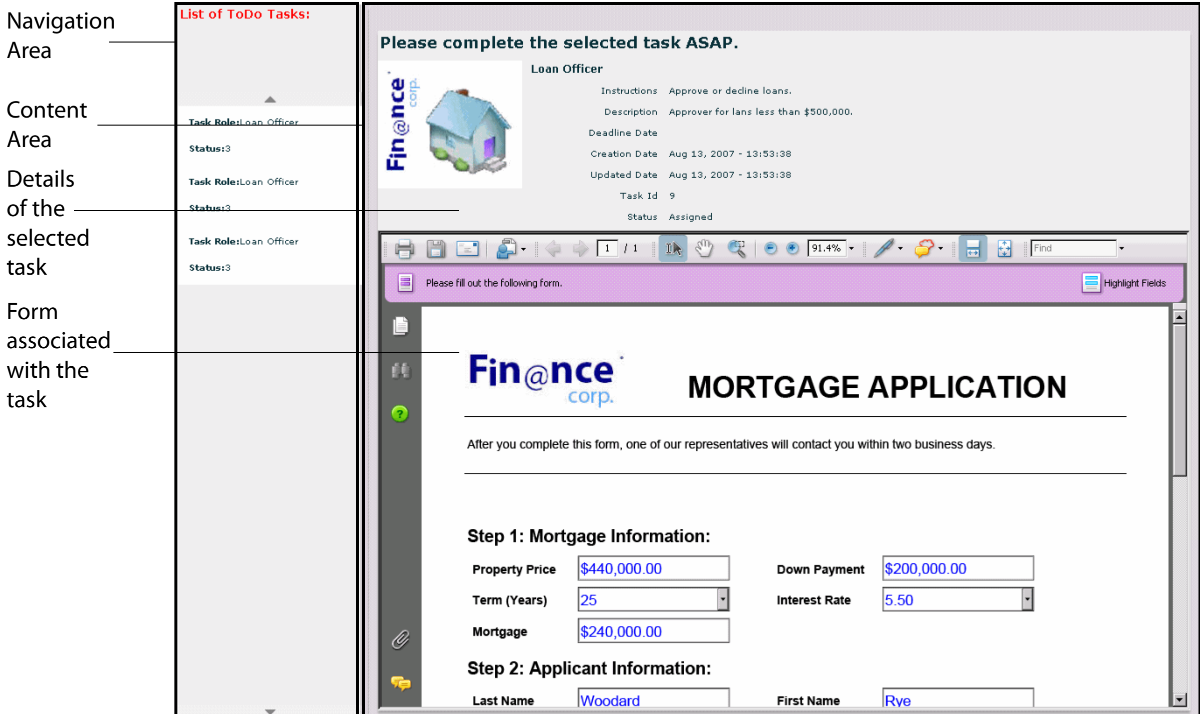
To create a new layout, you need to use components from the Presentation layer, Domain Model layer, and Core API layer. For example, to work with tasks, use the `lc:Task` component (Domain Model layer component); to work with the navigation area, use the `lc:EmbossedNavigator` component (Presentation layer, view component). When you create a custom layout, you can create additional components. Depending on the layout changes, you my need to create several components and use them together to exchange information.

For example, to make layout changes to manage tasks, create a new layout by using multiple custom visual components that consist of view and presentation model components that your default application file uses, which is the same as any other component in the Presentation layer. Organizing your code into separate view and presentation model components promotes future code reuse and ease in testing the code. The new layout is a three-pane interface that contains the following areas:

Navigation: Represents a vertical listing of all the tasks that are assigned to the user. This area is for navigating between a user’s uncompleted tasks. It also controls what appears in the content area.

Content: Displays the selected item from the navigation area. For example, in the following illustration, the details of the task are in the uppermost area, and the form that is associated with the task area is displayed in the bottom area. Both areas are referred to as the *content area*. For example, you can create

a content area that displays the details of the task that is assigned to the user and the form that is associated with the task.



To create your own layout, create two custom visual components that consists of view and presentation model components before you use them in the default application file.

The navigation area is for selecting from a list of tasks, and the content area is for displaying the image associated with the process that the task belongs to, the details of the task, and the form associated with the task. Perform these tasks to create a three-pane layout that displays a navigation area and content area:

1. Creating a folder for custom components
2. Creating a presentation model component for the content area
3. Creating a view component for the content area
4. Creating the presentation model component for the navigation area
5. Creating a view component for the navigation area
6. Creating the default application for a custom layout

Creating a folder for custom components

It is recommended that you create a folder for the components that you create to reuse.

To create a folder for custom components:

1. In Flex Builder, in the Workspace project, right-click **src**, and select **New** > **Folder**.

2. In the New Folder dialog box, in **Folder name** box, type a name for your folder, such as `custComp`, and then click **Finish**.

Creating a presentation model component for the content area

You create a presentation model component to store the data that will be displayed in the content area. The content area will use other presentation models from the Workspace API to retrieve information about details of the selected task and also the form that is associated with the task.

When you create a presentation model, make the properties and methods that store and access data, respectively, bindable. The bindable properties and methods allow view components to be updated dynamically when the data changes in the presentation model component. You need to extend the `lc:PresentationModel` class to implement a presentation model component.

Before you write the application logic for the presentation model, create an ActionScript class in Flex Builder based on the `lc:PresentationModel` class.

➤ **To create an ActionScript class for the content area:**

1. In Flex Builder, in the Navigator view, right-click the folder that you created in your Flex project to store custom components and select **New** > **ActionScript Class**. For example, right-click **custComp**.
2. In the New ActionScript Class dialog box, in the **Package** box, type a new name for the package or accept the default, which is the folder name. For example, use the default `custComp`, which is provided because the folder is named custComp.
3. In the **Name** box, type a name for the class. For example, type `CustTaskFormDisplayModel`.
4. Beside the **Superclass** box, click **Browse** and select **PresentationModel - lc.presentationmodel**.
5. Click **OK** and then click **Finish**.

After you create your ActionScript class, complete these steps in Flex Builder to write the application logic code for the presentation model component:

1. In the editor for the new ActionScript class, below the `import lc.presentationmodel.PresentationModel` statement, add `import` statements for the following classes:
   - `lc.core.Token`
   - `lc.core.events.WorkspaceFaultEvent`
   - `lc.domain.Task`
   - `lc.domain.TaskImageModel`
   - `lc.domainTaskInfoModel`
   - `lc.task.form.TaskForm`
   - `flash.external.ExternalInterface`

   Each class you import that is prefixed with `lc` is using a component from the Workspace API. (See *LiveCycle ES Update 1 ActionScript Reference*.) The `ExternalInterface` class is a `flash` class you need to work with return values.

   ```
   package custComp
   {
   import lc.presentationmodel.PresentationModel;
   ```

```
import lc.core.Token;
import lc.core.events.WorkspaceFaultEvent;
import lc.domain.Task;
import lc.task.TaskImageModel;
import lc.task.TaskInfoModel;
import lc.task.form.TaskForm;
import flash.external.ExternalInterface;

public class CustTaskFormDisplayModel extends PresentationModel
```

2. Add the `[Bindable]` tag immediately above the class definition to specify that all members in the class are bindable. You need to make data members specifically bindable in order to update the data in the user interface when you specify the data bindings in the view component.

```
[Bindable]
public class CustTaskFormDisplayModel extends PresentationModel
```

3. Create public variables of the following classes from the Domain Model layer:
   - `Task`: An object that stores the currently selected task. For example, create a variable named `currTask`.
   - `TaskForm`: An object that handles the retrieval and display of the form that is currently associated with the selected task. For example, create a variable named `currTaskForm`.
   - `String`: Object that stores any associated errors. For example, create a variable named `errorString`.
   - `TaskInfoModel`: An object that accesses the details of a task. For example,create an instance of the Model component named `currTaskInfoModel`.
   - `TaskImageModel`: An object that stores the image associated with the process. For example, create an instance of the presentation model component named `currTaskImageModel`.

```
public class CustTaskFormDisplayModel extends PresentationModel
{
    public var currTask:Task;
    public var errorString:String;
    public var currTaskForm:TaskForm;
    public var currTaskInfoModel:TaskInfoModel;
    public var currTaskImageModel:TaskImageModel;
```

4. Create a private function that takes a parameter of type `WorkspaceFaultEvent` and returns `void`. This function handles any faults that occur while loading the form. In a private function, build an error message by using the error information from the `WorkspaceFaultEvent` class and copy it to the `String` instance you created in step 3. For example, the function may look like the following code:

```
public function  handleTaskInfoError(event:WorkspaceFaultEvent):void
{
     errorString = "A <font color='#ff0000'>problem</font> occurred
                    trying to load the form\n Detailed Message: "
                  + event.errorMessage.formattedMessage;
}
```

5. Create a public setter method that handles receiving an object of type `Object`. This setter method also handles the currently selected task from the navigation area, and retrieves the information for the selected task and the associated form. The following tasks are required for the setter method:
   - Cast the `Object` value as an `lc:Task` object and assign it to the variable you created to store the `currTask`.

- Using the `ExternalInterface` function, call the `getAcrobatVersion` function to retrieve the current Acrobat version.
- Using the `lc:TaskForm` object, display the form that is associated with the selected task by using the `load` method, passing both the Acrobat version and the `Task` object as parameters. If necessary, the form will be loaded with Acrobat.
- Add the fault handler that you created in step 4 by using the `lc:Token` that is returned from calling the `load` method.

For example, if you created a setter method named `currData`, it looks like the following code:

```
public function set currData(value:Object):void
{
   if(value)
   {
      currTask = Task(value);

      var acrobatVer:Object  = new Object();
      var acroVer:String  =
                          ExternalInterface.call("getAcrobatVersion", []);
      acrobatVer["acrobatVersion"] = acroVer;

      var token:Token = currTaskForm.load(acrobatVer, currTask, 0);
      token.addFaultHandler(handleTaskInfoError);
   }
}
```

6. Create a public function named `initialize` that has no parameters, returns `void`, and overrides the `initialize` method of the parent class. In the function, create the following data bindings to update the task details, image, and associated form when a new task is selected in the navigation area:

   Note: In ActionScript, use the `BindingUtils.bindProperty` method. For convenience, you can use the `bindProperty` method, which is inherited from the `lc:PresentationModel` class.

   - When the `lc:Task` object is updated in the current presentation model component, the task property from the `lc:TaskInfoModel` object is updated.
   - When the `lc:Task` object is updated in the current presentation model component, the task property from the `lc:TaskImageModel` object is updated.

   For example, the function looks similar to the following code if you name the information in the presentation model with the names described in the example in step 3:

```
override public function initialize():void
{
   bindProperty(currTaskInfoModel,  "task", this, "currTask");
   bindProperty(currTaskImageModel, "task", this, "currTask");
}
```

7. Select **File** > **Save** to save the presentation model component file.

Example: ***The presentation model component for the content area***

```
package custComp
{
   import lc.presentationmodel.PresentationModel;
   import lc.core.Token;
   import lc.core.events.WorkspaceFaultEvent;
   import lc.domain.Task;
```

```
import lc.task.TaskImageModel;
import lc.task.TaskInfoModel;
import lc.task.form.TaskForm;
import flash.external.ExternalInterface;

//Specify that all members of the class support data binding
[Bindable]
public class CustTaskFormDisplayModel extends PresentationModel
{
   //The currently selected task
   public var currTask:Task;
   //The error message.
   public var errorString:String;
   //The form associated with the selected task.
   public var currTaskForm:TaskForm;
   //The presentation model for the task information details.
   public var currTaskInfoModel:TaskInfoModel;
   //The presentation model for the image associated with the process of the
   //selected task.
   public var currTaskImageModel:TaskImageModel;

   //Populates the errorString property with any errors while trying
   //to display the form.
   public function  handleTaskInfoError(event:WorkspaceFaultEvent):void
   {
      errorString = "A <font color='#ff0000'>problem</font> occurred
                     trying to load the form\n Detailed Message: "
            + event.errorMessage.formattedMessage;
   }

   //The setter method to populate data for this class that includes
   // the currently selected task and the form associated with the task.
   public function set currData(value:Object):void
   {
      if(value)
      {
         currTask = Task(value);

         var acrobatVer:Object  = new Object();
         var acroVer:String  =
                         ExternalInterface.call("getAcrobatVersion", []);
         acrobatVer["acrobatVersion"] = acroVer;

         var token:Token = currTaskForm.load(acrobatVer, currTask, 0);
         token.addFaultHandler(handleTaskInfoError);
      }
   }

   //Binds information stored by this presentation model component to
   //data from other data models.This ensures that when data is changed, it
   //is updated everywhere.
   override public function initialize():void
   {
      bindProperty(currTaskInfoModel,  "task", this, "currTask");
      bindProperty(currTaskImageModel, "task", this, "currTask");
```

```
        }

    }
}
```

Creating a view component for the content area

After you create the presentation model component to store the information for the content area, you can use the component when you write the application logic for the view component. The view component represents the user interface that is displayed to users and is typically an MXML component.

Before you write the application logic for the view component, create an MXML component in Flex Builder.

➤ **To create an MXML component for the content area:**

1. In Flex Builder, in the Navigator view, right-click the folder you created in your Flex project to store custom components and select **New** > **MXML Component**. For example, right-click **custComp**.
2. In the New MXML Component dialog box, in the **Filename** box, type a name. For example, type `CustTaskFormDisplay`.
3. In the **Based On** list, select a root container. For example, select **Canvas**.
4. In the **Width** and **Height** boxes, type `100%`, and then click **Finish**.

After you create your MXML component in Flex Builder, complete these tasks to write the application logic code for the view component:

1. In Flex Builder, in the MXML file you created, add the following attributes to the root container in order to use the Workspace API components and custom components that you created in the current Flex project:
   - `xmlns:lc="http://www.adobe.com/2006/livecycle"`
   - `xmlns:cc="custComp.*"`

   For example, if your root container for the MXML component is `mx:Canvas` and you created a folder named custComp in which to store the components you created in the project, your code may appear as follows:

   ```
   <mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
             xmlns:lc="http://www.adobe.com/2006/livecycle"
             xmlns:cc="custComp.*"
             width="100%" height="100%">
   ```
2. Add a `lc:SessionMap` object under the opening tag for your root component with an `id` attribute for accessing the authenticated session information that is required to use the Workspace API components. For example, if your root container for the MXML component is `mx:Canvas`, your code may appear as follows:

   ```
   <mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
             xmlns:lc="http://www.adobe.com/2006/livecycle"
             xmlns:cc="custComp.*"
             width="100%" height="100%">
     <lc:SessionMap id="session"/>
   ```
3. Design the user interface by performing the following tasks after the closing `<lc.SessionMap>` tag:

Note: In this user interface design, the top pane in the content area includes an image that is associated with the process for the task and the details of the selected task. The bottom pane displays the form that is associated with the task.

- Add an `mx:Panel` component with the `top`, `bottom`, and `right` attributes set to a value of `5`, and the left attribute set to a value of `1`.
- After the opening `<mx:Panel>` tag, add an `mx:Label` component and set the `text` attribute to a value of `Please complete the selected task ASAP.`, set `fontSize` to a value of `18`, and set `fontWeight` to a value of `bold`.
- After the closing `<mx:Label>` tag, add an `mx:HBox` component and set the `width` attribute to a value of `100%`.
- After the opening `<mx:HBox>` tag, add an `lc:TaskImage` component, which displays the image associated with the process of the selected task, set the `id` attribute to a name, and bind the `session` attribute to the `lc:SessionMap` object you created in step 2. For example, for the `id` attribute, type `myTaskImage`.
- After the closing `<lc:TaskImage>` tag, add an `lc:TaskInfo` component, which displays the details of the selected task, set the `id` attribute to a name, and bind the session attribute to the `lc:SessionMap` you created in step 2. For example, for the`id` attribute, type `myTaskInfo`.
- After the closing `<mx:HBox>` tag, add an `lc:TaskForm` component and set the `width` and `height` attributes to a value of `100%`. This component displays the form that is associated with the selected task.

Your code may appear as follows:

```
<mx:Panel>
<mx:Label text="Please complete the selected task."
          fontSize="18" fontWeight="bold"/>
  <mx:HBox width="100%">
    <lc:TaskImage id="myTaskImage" session="{session}"/>
    <lc:TaskInfo  id="myTaskInfo" session="{session}" width="100%"/>
  </mx:HBox>

  <!-- The form associated with the selected task. -->
  <lc:TaskForm id="myTaskForm" width="100%" height="100%" />
</mx:Panel>
```

4. Create an instance of the presentation model object that you want to retrieve, and bind the user interface components to the properties in the presentation model component that you want to display. For example, after the closing `<lc:Panel>` tag, create an instance of the presentation model component that you created for your content area (see Creating a presentation model component for the content area) and, for each of the following attribute types, bind each attribute to a view component:
   - `session`: Bind this attribute to the `lc:SessionMap` component you created in step 2.
   - `currTaskImageModel`: Bind this attribute to the `model` attribute of the `lc:TaskImage` object you created in step 3.
   - `currTaskInfoModel`: Bind this attribute to the `model` attribute of the `lc:TaskImage` object you created in step 3.
   - `currTaskForm`: Bind this attribute to the to the `lc:TaskForm` component you created in step 3.

- `currData`: Binds to `data`, which updates all the information in the presentation model component. For example, the setter method for the presentation model you created in Creating a presentation model component for the content area was `currData`.

For example, if the objects you created have the following names, your code may appear as the code follows:

- `lc:SessionMap: session`
- `lc:TaskImageModel: myTaskImage`
- `lc:TaskInfoModel: myTaskInfo`
- `lc:TaskForm: myTaskForm`

```
<cc:CustTaskFormDisplayModel id="myModel"
                             session="{session}"
                             currTaskImageModel="{myTaskImage.model}"
                             currTaskInfoModel="{myTaskInfo.model}"
                             currTaskForm="{myTaskForm}"
                             currData="{data}"/>
```

5. Select **File** > **Save** to save the view component.

Example: ***The view component for the content area***

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
       xmlns:lc="http://www.adobe.com/2006/livecycle"
       xmlns:cc="custComp.*"
       width="100%" height="100%">

  <!-- Retrieve the authenticated session information required to communicate
       with LiveCycle ES. -->
  <lc:SessionMap id="session"/>

  <!-- Display the details and form associated with the selected task. -->
  <mx:Panel id="panel" top="5" left="1" right="5" bottom="5">
    <mx:Label text="Please complete the selected task ASAP."
              fontSize="18" fontWeight="bold"/>

    <!-- The details of the selected task and also the image associated
         with the process for which the process belongs. -->
    <mx:HBox width="100%">
      <lc:TaskImage id="myTaskImage" session="{session}"/>
      <lc:TaskInfo  id="myTaskInfo" session="{session}" width="100%"/>
    </mx:HBox>

    <!-- The form associated with the selected task. -->
    <lc:TaskForm id="myTaskForm" width="100%" height="100%" />
  </mx:Panel>
```

```
<!-- The custom presentation model component used to access the data for
     display. Bind the presentation model to the view. -->
<cc:CustTaskFormDisplayModel id="myModel"
                             session="{session}"
                             currTaskImageModel="{myTaskImage.model}"
                             currTaskInfoModel="{myTaskInfo.model}"
                             currTaskForm="{myTaskForm}"
                             currData="{data}"/>
</mx:Canvas>
```

Creating the presentation model component for the navigation area

You create a presentation model component to retrieve the list of tasks that appear in the navigation area. The list of tasks can be all tasks (completed, active, and draft) that are assigned to a user or only the active tasks that are assigned to that user.

When you create a presentation model component, make the properties and methods that store and access data, respectively, bindable. The bindable properties and methods allow view components to be updated dynamically when data is changed in the presentation model component. You need to extend the `lc:PresentationModel` class to implement a presentation model component.

Before you write the application logic for the presentation model, create an ActionScript class based on the `lc:PresentationModel` class.

➤ **To create an ActionScript class for the presentation model component:**

1. In Flex Builder, in the Navigator view, right-click the folder you created in your Flex project to store custom components and select **New** > **ActionScript Class**. For example, right-click **custComp**.
2. In the New ActionScript Class dialog box, in the **Package** box, type a new name for the package or accept the default, which is the folder name. For example, use the default `custComp`, which is provided because the folder is named custComp.
3. In the **Name** box, type a name for the class. For example, type `CustNavigatorModel`.
4. Beside the **Superclass** box, click **Browse** and select **PresentationModel - lc.presentationmodel**.
5. Click **OK** and then click **Finish**.

After you create your ActionScript class in Flex Builder, complete these steps to write the application logic code for the presentation model component:

1. In Flex Builder, in the editor for the new ActionScript class you created, locate the `import lc.presentationmodel.PresentationModel` statement, and add `import` statements for the following classes:
   - `lc.domain.QueuesManager`
   - `mx.collections.ListCollectionView`

   Each class you import that is prefixed with `lc` is using a component from the Workspace API. (See *LiveCycle ES Update 1 ActionScript Reference*.) The `ExternalInterface` class is a `flash` class you need to work with return values. Your code may appear as follows:

   ```
   package custComp
   {
      import lc.presentationmodel.PresentationModel;
   ```

```
import mx.collections.ListCollectionView;
import lc.domain.QueuesManager;
```

2. Add the `[Bindable]` tag immediately above the class definition to specify that all members in the presentation model component that are bindable in order that the user interface is updated. Your code may appear as follows:

```
[Bindable]
public class CustNavigatorModel extends PresentationModel
```

3. Create a public variable of type `mx:ListCollectionView` to store the list of `lc:Task` objects in the presentation model component. For example, you can create a variable named `currTaskList` that stores the list of assigned tasks to the user. Your code may appear as follows:

```
public var  currTaskList:ListCollectionView;
```

4. Create a public function named `initialize` that has no parameters, returns `void`, and overrides the `initialize` method of the parent class. In the function, perform the following tasks:
   - Invoke the `initialize` method from the parent class.
   - Retrieve an instance of an `lc:QueuesManager` class by using the inherited `session` property from the `lc:PresentationModel` class. Using an `lc:QueuesManager` object retrieves all active tasks that are assigned to the user.
   - Retrieve the list of tasks that is assigned to the user by using the `lc:QueuesManager` object using the `defaultQueue.tasks` property and populating the `mx:ListCollectionView` object that you created in step 3. Your code may appear as follows:

```
override public function initialize():void
{
  //Call the initialize method from PresentationModel class
  super.initialize();
  //Retrieve the collection of tasks as collection tokens
  var queuesManager:QueuesManager =
            QueuesManager(session.getObject("lc.domain.QueuesManager"));
  currTaskList = queuesManager.defaultQueue.tasks;
}
```

5. Select **File** > **Save** to save the presentation model component.

Example: ***The presentation model component for the navigation area***

```
package custComp
{
  import lc.presentationmodel.PresentationModel;
  import mx.collections.ListCollectionView;
  import lc.domain.QueuesManager;

  //Specify that all members of the class support data binding
  [Bindable]
  public class CustNavigatorModel extends PresentationModel{

    public function CustNavigatorModel()
    {
      super();
    }

    public var currTaskList:ListCollectionView;
```

```
override public function initialize():void
     {
        //Call the initialize method from PresentationModel class
        super.initialize();
        //Retrieve the collection of tasks as collection tokens
        var queuesManager:QueuesManager =
                  QueuesManager(session.getObject("lc.domain.QueuesManager"));
        currTaskList = queuesManager.defaultQueue.tasks;
     }
   }
}
```

Creating a view component for the navigation area

After you create the presentation model component to store the information for the navigation area, you use it to write the application logic for the view component. The view component represents the user interface that is displayed to users and is typically an MXML component.

For the navigation area, the `lc:EmbossedNavigator` component is used, which can be modified to display multiple pieces of data for each entry in the list. You can define additional information in the navigation area by using the `<lc:header>` tag.

For example, you may want to retrieve all tasks that are assigned to the user who is currently logged in to your application and display the name and status of each task as an entry. You may also want to provide information that instructs the user to select one of the tasks in the navigation area.

The view component that you create displays both the navigation pane and the content pane. Write the application logic in the view component to connect the navigation area and content areas.

Before you write the application logic for the view component, you need to create an MXML component in Flex Builder.

➤ **To create a an MXML component for the navigation area:**

1. In Flex Builder, in the Navigator view, right-click the folder that you created in your Flex project to store custom components and select **New** > **MXML Component**. For example, right-click **custComp**.
2. In the New MXML Component dialog box, in the **Filename** box, type a name. For example, type `CustNavigator`.
3. In the **Based on** list, select a root container. For example, select **Canvas**.
4. In the **Width** and **Height** boxes, type `100%`, and then click **Finish**.

After you create your MXML component in Flex Builder, complete these tasks to write the application logic code for the view component:

1. In Flex Builder, in the MXML file, add the following attributes to the root container in order to use the Workspace API components and custom components that you created in the current Flex project:
   - `xmlns:lc="http://www.adobe.com/2006/livecycle"`

- `xmlns:cc="custComp.*"`

For example, if your root container for the MXML component is a `mx:Canvas` component, your code may appear as follows:

```
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
            xmlns:lc="http://www.adobe.com/2006/livecycle"
            xmlns:cc="custComp.*"
            width="100%" height="100%">
```

2. Add an `lc:SessionMap` object under the opening tag for your root component with an `id` attribute to access the authenticated session information that is required to use the Workspace API components. For example, if your root container for the MXML component is an `mx:Canvas` component, your code may appear as follows:

```
<mx:Canvas mlns:mx="http://www.adobe.com/2006/mxml"
           xmlns:lc="http://www.adobe.com/2006/livecycle"
           xmlns:cc="custComp.*"
           width="100%" height="100%">
  <lc:SessionMap id="session"/>
```

3. After the `<lc:SessionMap>` tag, create an instance of the presentation model component for the navigation area and set the `id` attribute to bind to the `session` attribute from the `lc:SessionMap` object you created in step 2. (See Creating the presentation model component for the navigation area.) Your code may appear as follows:

```
<cc:CustNavigatorModel id="myModel" session="{session}"/>
```

4. Add an `lc:EmbossedNavigator` component and set the `dataProvider` attribute to bind to the property that stores the list of tasks. For example, you can create the following data binding to an instance of a presentation model component with a property that stores the list of tasks called `currTaskList`:

```
<lc:EmbossedNavigator dataProvider="{myModel.currTaskList}"
                      width="100%" height="100%">
```

5. After the opening `<lc:EmbossedNavigator>` tag, add an `lc:header` tag. By adding it as a tag, you can add other components, such as `mx:Label` or `mx:Image`. For example, add an `mx:Label` component and add the following attributes:

   - `text` and set it to a value of `Please review loan applications`
   - `color` and set it to a value of `#FF0000` for red
   - `fontSize` and set it to a value of `10`
   - `width` and set it to a value of `200`

```
<lc:header>
  <mx:Label text="Please review loan applications" color="#FF0000"
            fontSize="10" width="200" />
</lc:header>
```

6. After the closing `<lc:header>` tag, add an `lc:listItemRenderer` tag, which is used to create each entry in the navigation area by adding an `mx:Component` tag. By adding it as a tag, you can specify a component directly in the same MXML file.

   For example, you can add two labels that appear vertically and are aligned to the left of the navigation area (without scroll bars) that displays the name of the task and the status of each task. You can retrieve the information for a task because you can access `lc:Task` objects that are passed from the `data` attribute of the `lc.listItemRenderer` component as specified by the presentation model

component for the `dataProvider` attribute of the `lc:EmbossedNavigator` component in step 4. Perform the following tasks to complete this step.

- After the opening `<lc:listItemRenderer>` tag, add an `mx:Component` tag. `lc:listItemRenderer` is an attribute of the `EmbossedNavigator` component that you would typically point to the component. In this case, specify `lc:listItemRenderer` as a tag to function like a container tag.
- After the opening `<mx:Component>` tag, add an `mx:VBox` container and then configure the `height` attribute to a value of `50`; the `paddingLeft`, `paddingRight`, `paddingTop`, and `paddingBottom` attributes to a value of `10`; the `width` and `height` attributes to a value of `100%`; the `horizontalAlign` attribute to a value of `left`; and the `verticalScrollPolicy` attribute to a value of `off`.
- After the opening `<mx:VBox>` tag, add an `mx:Label` component to display the name of the step from the `lc:Task` object. To access the data, use the `data` property from the `lc:listItemRenderer` object, which is an `mx:ListItemRender` component, in order to display the name of the step by using the `stepName` property from the `lc:Task` object.
- After the closing `<mx:Label>` tag, add an `mx:Spacer` component to separate the items.
- After the closing `<mx:Spacer>` tag, add an `mx:Label` component to display the status of the task. To access the data, use the `data` attribute again but display the `status` property from the `lc:Task` object.

Note: The properties for the `lc:Task` object are in the `lc.domain.Task` namespace of the *LiveCycle ES ActionScript Reference*.

```
<lc:listItemRenderer>
   <mx:Component>
       <mx:VBox horizontalAlign="left"
                verticalScrollPolicy="off"
                paddingTop="10"
                paddingBottom="10"
                paddingLeft="10"
                paddingRight="10"
                width="100%"
                height="100%">
            <mx:Label htmlText="{'&lt;b&gt;Task Role: &lt;/b&gt;' +
                                  data.stepName}"
                  width="100%"/>
            <mx:Label htmlText="{'&lt;b&gt;Status: &lt;/b&gt;' +
                                  data.status}"
                  width="100%"  />
      </mx:VBox>
   </mx:Component>
</lc:listItemRenderer>
```

7. After the closing `<lc:itemRenderer>` tag, add an `<lc:contentItemRenderer>` tag to specify the component that will handle the display of information in the content area.

8. After the opening `<lc:contentItemRenderer>` tag, add an `<mx:Component>` tag.

9. After the opening `<mx:Component>` tag, add the view component you created for the content area. (See Creating a view component for the content area.) For example, add `cc:CustTaskFormDisplay` and pass the session information.

```
<lc:contentItemRenderer>
```

```
<mx:Component>
   <cc:CustTaskFormDisplay session="{session}"
                           width="100%" height="100%" />
</mx:Component>
```

10. Select **File** > **Save** to save the view component for the navigation area.

Example: *The view component for the navigation area*

```
<?xml version="1.0" encoding="utf-8"?>
mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
        xmlns:lc="http://www.adobe.com/2006/livecycle"
        xmlns:cc="custComp.*"
        width="100%" height="100%">

  <!-- Retrieve the authenticated session information required to
       communicate with LiveCycle ES. -->
  <lc:SessionMap id="session"/>

  <!-- Create an instance of the custom presentation model component to
       retrieve the data to display. -->
  <cc:CustNavigatorModel id="myModel" session="{session}"/>

  <!-- Add navigation component from the Workspace API -->
  <lc:EmbossedNavigator dataProvider="{myModel.currTaskList}"
                        width="100%" height="100%">

    <!-- Using the listItemRenderer property, retrieve each Task object and
         retrieve information using the 'data' object. -->
    <lc:header>
      <mx:Label text="Please Review loan applications" color="#FF0000"
                fontSize="10" width="200" />
    </lc:header>

    <!-- Specify the task information that is displayed for each task entry
        in the navigation pane. -->
    <lc:listItemRenderer>
      <mx:Component>
         <mx:VBox horizontalAlign="left"
                  verticalScrollPolicy="off"
                  paddingTop="10"
                  paddingBottom="10"
                  paddingLeft="10"
                  paddingRight="10"
                  width="100%"
                  height="100%">
            <mx:Label htmlText="{'&lt;b&gt;Task Role: &lt;/b&gt;' +
                                data.stepName}"
                                width="100%"/>
            <mx:Label htmlText="{'&lt;b&gt;Status: &lt;/b&gt;' +
                                 data.status}"
                                 width="100%"  />
         </mx:VBox>
      </mx:Component>
    </lc:listItemRenderer>

    <!-- Specify that the custom content renderer component
        that displays the details and form for the selected task. -->
    <lc:contentItemRenderer>
      <mx:Component>
        <cc:CustTaskFormDisplay session="{session}"
                                width="100%" height="100%" />
```

```
        </mx:Component>
      </lc:contentItemRenderer>
    </lc:EmbossedNavigator>
</mx:Canvas>
```

Creating the default application for a custom layout

After you create view and presentation model components for the navigation and content areas, you can use them in your default application file. To create the default application for a custom layout, perform the following steps:

1. In Flex Builder, in the Workspace project, right-click **src** and select **File > MXML Application**.
2. In the New MXML Application dialog box, in the **Filename** box, type a name for your application, such as `CustomLayoutWS.mxml`.
3. In Flex Builder, in the MXML file you created in the previous step, in the default `<mx:Application>` tag, add the following attributes to include the Workspace API and the location of your custom components:
   - `xmlns:lc="http://www.adobe.com/2006/livecycle"`
   - `xmlns:cc="custComp.*"`

   Note: `custComp.*` is the folder where you saved the components to display the navigation and content area that you created when you created your Flex project. (See Creating the application logic for a custom layout.)
4. Type `lc:AuthenticatingApplication` to replace the `<mx:Application>` tag.
5. After the opening `<lc:AuthenticatingApplication>` tag, add the component that you created to display the navigation area and bind the `session` attribute from the `lc:AuthenticatingAppliation` component to your `session` attribute for your custom component that you created in Creating a view component for the navigation area.

   For example, if you created a view component named `CustNavigator`, add the `cc:CustNavigator` component and set the `id` attribute to the `session` attribute from the `lc:AuthenticatingApplication` component.

   ```
   <cc:CustNavigator session="{session}" width="100%" height="100%"/>
   ```
6. Select **File** > **Save** to save the default application file.

Example: ***Creating a custom layout to view tasks assigned to the current user***

```
<?xml version="1.0" encoding="utf-8"?>
<lc:AuthenticatingApplication
          xmlns:mx="http://www.adobe.com/2006/mxml"
          xmlns:lc="http://www.adobe.com/2006/livecycle"
          xmlns:cc="custComp.*"
          layout="absolute">

  <cc:CustNavigator session="{session}" width="100%" height="100%"/>

</lc:AuthenticatingApplication>
```

Configuring the build.xml file to compile a custom MXML

Because you created your own MXML file, you can configure the build.xml file in the Workspace project to compile your MXML file as the default application instead of the default Main.mxml file. Typically, in Flex Builder, in your project, you right-click the MXML file and select Set as Default Application. However, because build scripts are used, you must modify the build.xml file. Before you perform these steps, ensure that you created an MXML file.

➤ **To configure the build.xml file to compile a custom MXML file:**

1. In the Workspace project, double-click the **build.xml** file to open it for editing.
2. Make the following changes to the build.xml file:
   - For the `application.name` property, change the value to `CustomLayout`.
   - For the `application.title` property, change the value to `Simple Customization WS`.
   - For the `app.name` property, change the `value` to the name of the MXML file that you created in Creating the default application for a custom layout. For example, if you created an MXML file named CustomLayoutWS.mxml, change the value to `CustomLayoutWS`.

   ```
   <!-- Modify the following properties to match your setup -->
   <property name="flex.sdk.home"
                location="C:/Program Files/Adobe/Flex Builder 3/sdks/3.0.1.rc1"
   />
   <property name="lc.sdk.dir"
    location="C:/Adobe/LiveCycle8.2/LiveCycle_ES_SDK" />
   <property name="application.title" value="Simple Customization WS"/>
   <property name="application.name" value="CustomLayout"/>
   <property name="app.name" value="CustomLayoutWS"/>
   ```

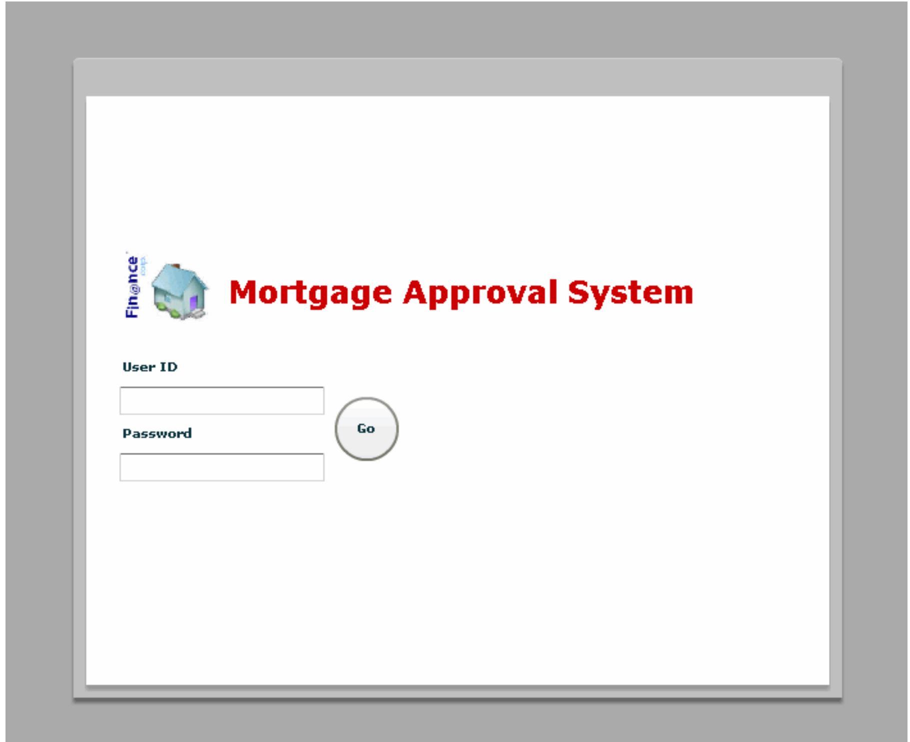
3. Save your changes.

After you compile and deploy your changes, the custom layout that you created can be seen. For example, in a JBoss turnkey installation, in a web browser, type `http://`*`[servername]`*`:8080/CustomLayout`, where *`[servername]`* is the name of your computer. The context root is CustomLayout.

11 Layout Customization - Creating a New Login Screen

LiveCycle Workspace ES is built using self-contained visual components and non-visual components collectively called the Workspace API, which provides process management functions as described in *LiveCycle ES Update 1 ActionScript Reference*. You can configure the `lc:AuthenticatingApplication` component, which handles displaying the login screen, to use a custom login screen that you create. You can easily replace the default login screen with your own in Workspace ES. This section describes how to implement a custom login screen.

When you need to go beyond changing the image and colors on the login screen, you can create your own custom component to handle the login process. For example, you may want to add additional functionality to the login screen or completely change its layout if the default login screen does not meet the requirements of your organization. As shown in the following illustration, you can replace your login screen so that it looks completely different from the Workspace ES default login screen.



Summary of steps

You must complete these high-level steps to create a Flex application that replaces the login screen for Workspace ES. The example that accompanies these steps describes how to implement the necessary interface for a login screen, create a new user interface for the login screen, and then use the custom login screen to start the Workspace ES application.

1. Configure your development environment for customizing Workspace ES. (See Configuring the Development Environment - Installing the Flex SDK.)

2. Configure Ant in your development environment. (See Configuring the Development Environment - Installing Ant to Flex Builder.)
3. Import and configure the Workspace project. (See Configuring the Development Environment - Configuring the Workspace Project.)
4. Create the application logic for a custom login screen. (See Creating the application logic for a custom login screen.)
5. Compile the Workspace project file. (See Compiling the Workspace project.)
6. Deploy the EAR file. (See Deploying a custom Workspace ES application for testing.)
7. Test the localization changes. (See Testing the Workspace ES application.)

Creating the application logic for a custom login screen

Flex projects are created with a default application file that contains an empty `mx:Application` container component. To use the Workspace API, replace the `<mx:Application>` tag with the `lc:AuthenticatingApplication` tag. The `lc:AuthenticatingApplication` component, which is a container, functions similarly to the `mx:Application` container. However, it displays the Workspace ES login screen as necessary and stores the session information required to use Workspace API components. You can modify the `loginPage` attribute to display a custom login screen instead of the default login screen.

After you log in to Workspace ES from the custom login screen, you use the `lc:Desktop` component, which is the Workspace ES application exposed as a component.

To create the application logic, complete the following high-level steps:

1. Creating a folder for custom components
2. Creating an ActionScript class to implement the lc:ILogin interface
3. Creating the user interface for the login screen
4. Creating the application logic in the default application file

Creating a folder for custom components

It is recommended that you create a folder for the components that you create to reuse.

➤ **To create a folder for custom components:**

1. In Flex Builder, in the Workspace project, right-click **src** and select **New** > **Folder**.
2. In the New Folder dialog box, in **Folder name** box, type a name for your folder, such as `custComp`, and then click **Finish**.

For the purpose of this example, create a Folder named *custImages* under the custComp folder and copy the houseImage.jpg file from the LiveCycle ES SDK from one of these locations:

Workbench ES installation: Navigate to the Workspace folder in *[installdir]*/LiveCycle_ES_SDK/samples/

LiveCycleES/MortgageLoan-Prebuilt/Images, where *[installdir]* is the location where Workbench ES is installed on your computer.

LiveCycle ES server: Navigate to the Images folder in *[installdir]*/LiveCycle_ES_SDK/samples/LiveCycleES/MortgageLoan-Prebuilt/Images, where *[installdir]* is where LiveCycle ES is installed on your server.

For example, for a JBoss turnkey installation, go to the Workspace folder in /Adobe/LiveCycle8/LiveCycle_ES_SDK/samples/LiveCycleES/MortgageLoan-Prebuilt/Images.

Creating an ActionScript class to implement the lc:ILogin interface

Before you write the application logic to implement the `lc:ILogin` interface from Workspace ES, you need to create an ActionScript class. This instance is an example of extending a Core API layer component.

➤ **To create an ActionScript class:**

1. In Flex Builder, in the Navigator view, right-click the folder you created to store the custom components that you create for your project. For example, right-click **custComp** and select **New** > **ActionScript Class**.
2. In the New ActionScript Class dialog box, in the **Package** box, type a new name for the package or accept the default, which is the folder name. For example, use the default `custComp`, which is provided because the folder is named custComp.
3. In the **Name** box, type a name for the class. For example, type `CustLoginPageAdapter`.
4. Beside the **Superclass** box, click **Browse** and select a user interface component that you will use for the custom screen. For example, select **Box - mx.containers** and then click **OK**.
5. In the Interfaces area, click **Add**, select **ILoginPage - lc.core**, and then click **OK**.
6. Confirm that only **Generate functions inherited from interfaces** is selected and then click **Finish**.

A new ActionScript file is created that automatically includes the required import statements and a template for inherited functions from the inherited interfaces.

After you create your ActionScript class in Flex Builder, complete these steps to write the application logic code that implements the `lc:ILogin` interface:

1. In Flex Builder, in the ActionScript class you created, locate the class definition (`public class CustLoginPageAdapter extends Box implements ILoginPage`).
2. Below the class definition, add bindable properties of the same type to mirror the `errorMessage`, `password`, `relogin`, `serverUrl`, and `userid` properties for the `lc:ILogin` interface. You must create your own bindable versions of the properties.

   For example, create protected variables named `myErrorMessage` of type `Message`, `myPassword` of type `String`, `myRelogin` of type `Boolean`, `myServerUrl` of type `String`, and `myUserid` of type `String`.

```
public class CustLoginPageAdapter extends Box implements ILoginPage
{

    [Bindable]
   protected var myErrorMessage:Message;
    [Bindable]
```

```
protected var myPassword:String;
[Bindable]
protected var myRelogin:Boolean;
[Bindable]
protected var myServerUrl:String;
[Bindable]
protected var myUserid:String;
```

Note: Verify that you added the `[Bindable]` tag to the beginning of each property you create.

3. Add a new function to handle dispatching a login message to Workspace ES. You must dispatch the `WorkspaceEvent.LOGIN` event as a bubbling event that cannot be canceled.

   For example, you can create a protected function called `doLogin` that returns `void` and dispatches an instance of the `WorkspaceEvent.LOGIN` event in an instance of the `lc:WorkspaceEvent` class.

```
protected function doLogin():void
{
   var myLoginEvent:WorkspaceEvent =
          new WorkspaceEvent(WorkspaceEvent.LOGIN, true,false, null);
   dispatchEvent(myLoginEvent);
}
```

Note: You may need to add an `import lc.core.events.WorkspaceEvent` to your existing list of import statements.

4. For the `serverUrl` property, perform the following steps:

   - For the getter method, change the default `return null;` statement to return the value of the property you created to store the server URL information in step 1.
   - For the setter method, assign the value that is passed to the function, which is `serverUrl`, to the property you created to store the userid information in step 1.

For example, the setter and getter methods will look like the following code if you create a property named `myErrorMessage`.

```
public function get serverUrl():String
{
   return myServerUrl;
}

public function set serverUrl(serverUrl:String):void
{
   myServerUrl = serverUrl;
}
```

5. For the `userid` property, perform the following s:
   - For the getter method, change the default `return null;` statement to return the value of the property you created to store the userid information in step 1.
   - For the setter method, assign the value that is passed to the function, which is `userid`, to the property you created to store the userid information in step 1.

   For example, the setter and getter methods will look like the following code if you create a property named `myUserid`:

```
public function get userid():String
{
   return myUserid;
}

public function set userid(userid:String):void
{
   myUserid = userid;
}
```

6. For the `errorMessage` property, perform the following s:
   - For the getter method, change the default `return null;` statement to return the value of the property you created to store the error message in step 1
   - For the setter method, assign the value that is passed to the function, which is `error`, to the property you created to store the error message information in step 1.

   For example, the setter and getter methods will look like the following code if you create a property named `myErrorMessage`:

```
public function get errorMessage():Message
{
   return myErrorMessage;
}

public function set errorMessage(error:Message):void
{
   myErrorMessage = error;
}
```

Note: To determine whether additional error messages exist (called *nested messages*), cast the `Message` object as a `CompositeMessage` object. It is recommended that you display nested error messages separately. For example, you can display nested errors in a tool tip or in a separate logging file.

7. For the `password` property , perform the following s:
   - For the getter method, change the default `return null;` statement to return the value of the property you created to store the password in step 1.
   - For the setter method, assign the value that is passed to the function, which is `password`, to the property you created to store the userid information in step 1.

   For example, the setter and getter methods will look like the following code if you create a property named `myPassword`:

```
public function get password():String
{
   return myPassword;
}

public function set password(password:String):void
{
   myPassword = password;
}
```

8. For the `relogin` property, perform the following s:
   - For the getter method, change the default `return null;` statement to return the value of the property you created to store the `relogin` flag in step 1.
   - For the setter method, assign the value that is passed to the function, which is `relogin`, to the property you created to store the userid information in step 1.

   For example, the setter and getter methods will look like the following code if you create a property named `myRelogin`:

```
public function get relogin():Boolean
{
   return myRelogin;
}

public function set relogin(relogin:Boolean):void
{
   myRelogin = relogin;
}
```

9. Delete all other functions that are added to your class; otherwise, an error occurs when you compile the project.
10. Select **File** > **Save** to save the ActionScript class.

Example: ***A custom ActionScript class that implements the lc:ILoginPage interface***

```
package custComp
{
   import lc.core.ILoginPage;
   import flash.geom.Transform;
   import flash.geom.Rectangle;
   import flash.display.DisplayObjectContainer;
   import mx.containers.Box;
   import flash.events.Event;
   import mx.managers.ISystemManager;
   import flash.display.Sprite;
   import lc.core.Message;
```

```
import flash.display.DisplayObject;
import lc.core.CompositeMessage;
import lc.core.events.WorkspaceEvent;

public class CustLoginPageAdapter extends Box implements ILoginPage
{

   [Bindable]
   protected var myErrorMessage:Message;
   [Bindable]
   protected var myPassword:String;
   [Bindable]
   protected var myRelogin:Boolean;
   [Bindable]
   protected var myServerUrl:String;
   [Bindable]
   protected var myUserid:String;

   //Sends the authentication credentials to the server.
   protected function doLogin():void
   {
      var myLoginEvent:WorkspaceEvent =
                new WorkspaceEvent(WorkspaceEvent.LOGIN, true,false, null);
      dispatchEvent(myLoginEvent);
   }

   //The location of the server as a URL.
   public function get serverUrl():String
   {
      return myServerUrl;
   }

   //The location of the server as a URL.
   public function set serverUrl(serverUrl:String):void
   {
      myServerUrl = serverUrl;
   }

   //The user id.
   public function get userid():String
   {
      return myUserid;
   }

   //The user id.
   public function set userid(userid:String):void
   {
      myUserid = userid;
   }

   //The error message if a problem occurs while logging in.
   public function get errorMessage():Message
   {
      return myErrorMessage;
   }
```

```
        //The error message if a problem occurs while logging in.
        public function set errorMessage(error:Message):void
        {
            myErrorMessage = error;
        }

        //The password.
        public function get password():String
        {
            return myPassword;
        }

        //The password.
        public function set password(password:String):void
        {
            myPassword = password;
        }

        //A flag that specifies whether this is another attempt to login.
        public function get relogin():Boolean
        {
            return myRelogin;
        }

        //A flag that specifies whether this is another attempt to login.
        public function set relogin(relogin:Boolean):void
        {
            myRelogin = relogin;
        }
    }
}
```

Creating the user interface for the login screen

After you create your new ActionScript class that implements the `ILogin` interface, you can start creating the user interface for your login screen. Creating a separate class that implements the details of implementing the `ILogin` class allows for easier reuse when you create the user interface.

Before you write the application logic for the new login screen, you need to create an MXML component.

➤ **To create the user interface for the login screen as an MXML component:**

1. In Flex Builder, in the Flex Navigator view, right-click the folder you created earlier to store custom components and select **New** > **MXML Component**. For example, right-click **custComp**.

2. In the New MXML Component dialog box, in the **Filename** box, type a name (for example, type `CustLoginPage`).

3. In the **Based on** list, select **CustLoginPageAdapter**, which is the ActionScript class you created by following the steps in Creating an ActionScript class to implement the lc:ILogin interface.

4. In the **Width** and **Height** boxes, type `100%`, and then click **Finish**.

After you create your MXML component in Flex Builder, complete these steps to write the application logic code for the new login screen:

1. In the CustLoginPage.mxml file, for the `cc:CustLoginPageAdapter` component, add `http://www.adobe.com/2006/livecycle` as the `lc` namespace to access the Workspace API components and other attributes to help you customize your user interface.

   For example, to specify that the login screen is centered in the middle of the screen, set the `verticalAlign` attribute to a value of `middle`, set the `horizontalAlign` attribute to a value of `center`, and set the `backgroundColor` attribute to "#AAAAAA" for the `cc:CustLoginPageAdapter` component.

```
<cc:CustLoginPageAdapter xmlns:mx="http://www.adobe.com/2006/mxml"
                         xmlns:cc="custComp.*"
                         xmlns:lc="http://www.adobe.com/2006/livecycle"
                         verticalAlign="middle" horizontalAlign="center"
                         backgroundColor="#AAAAAA"
                         width="100%" height="100%">
```

2. Define the user interface for your component. You are not limited to what you use to create the user interface, and it is dependent on your requirements. The only requirement is that you bind the userid and password information to the user interface and then invoke the method to dispatch your login information to LiveCycle ES. For example, the user interface may include the following components:

   - An `mx:Panel` container that contains a Form container that contains fields for the userid and password.
   - A round button with the word *Go* for logging in to Workspace ES. This button is located beside the userid and password labels.
   - A different image and title that appears at the top of the panel.
   - Any error messages displayed immediately below the field where users type their password.

   For example, to create a layout similar to the illustration displayed in Layout Customization - Creating a New Login Screen, perform the following steps:

   - Add an `mx:HBox` container, set the `backgroundcolor` attribute to a value of `#FFFFFF` (white), and set the `verticalAlign` attribute to a value of `middle`.
   - After the opening `<mx:HBox>` tag, add an `mx:Image` component, set the `source` attribute to a value of `@Embed(source='../../custImages/houseImage.jpg')",` and set both the `height` and the `width` attributes to a value of `75`. You must embed the image to include it in the application
   - After the `<mx:Image>` tag, add an `mx:Label` component, set the text attribute to a value of `Mortgage Approval System`, set the `fontSize` to a value of `24`, set the `fontWeight` to a value of `bold`, and set the color set to a value of `#CCCCCC` (red).
   - After the closing `<mx:HBox>` tag, add another `mx:HBox` component and set the `verticalAlign` attribute to a value of `middle`.
   - After the second opening `<mx:HBox>` tag, add an `mx:VBox` component.
   - After the opening `<mx:VBox>` tag, add an `mx:Label` component, set the `text` attribute to a value of `User ID`, and set the `fontWeight` attribute to a value of `bold`.
   - After the closing `<mx:Label>` tag, add an `mx:TextInput` component, set the `id` attribute to a value of `username`, set the `tabIndex` attribute to a value of `1`, set the `change` event to update the property that stores the userid from the ActionScript class you are extending from the current value in the text field, and bind the `text` attribute to the property that stores the userid from the ActionScript class you are extending. For example, use the value of `myUserid`.

- After the `<mx:TextInput>` tag, add an `mx:Label` component, set the `id` attribute to a value of `Password`, and set the `fontWeight` attribute to a value of `bold`.
- After the closing `<mx:Label>` tag for the password, add another `mx:TextInput` component, set the `id` attribute to a value of `password`, set the `tabIndex` attribute to a value of `2`, set the `change` event to update the property that stores the password from the ActionScript class you are extending from the current value in the text field, and bind the `text` attribute to the property that stores the password from the ActionScript class you are extending. For example, use the value of `myPassword`.
- After the closing `<mx:TextInput>` tag for the password, add an `mx:Text` component, set the `color` attribute to a value of `#000000` (black), set the `width` attribute to a value of `100%`, and bind the `text` attribute to display the value of the property that stores the error messages, if any. For example, use the value of `myErrorMessage`.
- After the closing `<mx:VBox>` tag, add an `mx:Button` component, set the `id` attribute to a value of `login`, set the `tabIndex` to a value of `3`, set both the `height` and the `width` attributes to a value of `50`, set the `label` attribute to a value of `Go`, set the `cornerRadius` attribute to a value of `28`, and set the `click` event to call the function to authenticate your credentials from the ActionScript class you are extending. For example, call `doLogin`.

```
<mx:Panel paddingBottom="10" paddingRight="10"
          paddingLeft="10" paddingTop="100"
          width="600" height="500" backgroundColor="#FFFFFF">
     <mx:HBox backgroundColor="#FFFFFF" verticalAlign="middle">
        <mx:Image source =
                    "@Embed(source='../../custImages/houseImage.jpg')"
                 height="75" width="75"/>
        <mx:Label text="Mortgage Approval System" fontSize="24"
                 fontWeight="bold" color="#CC0000" />
     </mx:HBox>
     <mx:Spacer/>
     <mx:HBox verticalAlign="middle">
        <mx:VBox>
           <mx:Label text="User ID" fontWeight="bold"/>
           <mx:TextInput  id="username" text="{myUserid}" tabIndex="1"
                          change="myUserid=username.text"
                          enabled="{!myRelogin}"/>
           <mx:Label text="Password"  fontWeight="bold"/>
           <mx:TextInput  id="passwordInput" tabIndex="2"
                          displayAsPassword="true" text="{myPassword}"
                          change="myPassword=passwordInput.text"/>
           <mx:Spacer/>
           <mx:Text  color="#000000" width="100%"
                    text="{myErrorMessage == null ? null :
                          myErrorMessage.formattedMessage}"/>
        </mx:VBox>
        <mx:Button id="login" tabIndex="3" width="50" height="50"
                  label="Go" cornerRadius="28" click="doLogin()"/>
     </mx:HBox>
</mx:Panel>
```

3. Select **File** > **Save** to save your user interface screen.

Example: *Creating the user interface component that implements the lc:ILoginPage interface*

```
<?xml version="1.0" encoding="utf-8"?>

<cc:CustLoginPageAdapter
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="custComp.*"
  xmlns:lc="http://www.adobe.com/2006/livecycle"
  verticalAlign="middle"
  horizontalAlign="center"
  backgroundColor="#AAAAAA"
  width="100" height="100%">

  <mx:Panel paddingBottom="10" paddingRight="10"
            paddingLeft="10" paddingTop="100"
            width="600" height="500" backgroundColor="#FFFFFF">
      <mx:HBox backgroundColor="#FFFFFF" verticalAlign="middle">
        <mx:Image source=
                    "@Embed(source='../../custImages/houseImage.jpg')"
                  height="75" width="75"/>
        <mx:Label text="Mortgage Approval System" fontSize="24"
                  fontWeight="bold" color="#CC0000" />
      </mx:HBox>
      <mx:Spacer/>
      <mx:HBox verticalAlign="middle">
        <mx:VBox>
          <mx:Label text="User ID" fontWeight="bold"/>
          <mx:TextInput  id="username" text="{myUserid}" tabIndex="1"
                         change="myUserid=username.text"
                         enabled="{!myRelogin}"/>
          <mx:Label text="Password"  fontWeight="bold"/>
          <mx:TextInput  id="passwordInput" tabIndex="2"
                         displayAsPassword="true" text="{myPassword}"
                         change="myPassword=passwordInput.text"/>
          <mx:Spacer/>
          <mx:Text  color="#000000" width="100%"
                    text="{myErrorMessage == null ? null :
                         myErrorMessage.formattedMessage}"/>
        </mx:VBox>
        <mx:Button id="login" tabIndex="3" width="50" height="50"
                   label="Go" cornerRadius="28" click="doLogin()"/>
      </mx:HBox>
  </mx:Panel>

</cc:CustLoginPageAdapter>
```

Creating the application logic in the default application file

After you create the component for a custom login screen, you are ready to use it in your Flex applications. You can replace the default screen and use the entire Workspace ES application by adding the `lc:Desktop` component in your code. The `lc:Desktop` component exposes the Workspace ES application as a reusable component.

To create the application to use a custom login screen and then start the Workspace ES application, complete the following steps:

1. In Flex Builder, in the Workspace project, right-click **src**, and select **File > MXML Application**.

2. In the New MXML Application dialog box, in the **Filename** box, type a name for your application, such as `CustLoginPageWS.mxml`.

3. In Flex Builder, in the MXML file you created in the previous step, add the following attributes to the default `<mx:Application>` tag to include the Workspace API and the location of your custom components:

   - `xmlns:lc="http://www.adobe.com/2006/livecycle"`
   - `xmlns:cc="custComp.*"`

   Note: `custComp.*` is the folder where you saved the components to display the navigation and content area.

4. Type `lc:AuthenticatingApplication` in place of the `<mx:Application>` tag.

5. Add the `loginPage` attribute and set the value to the component you created by following the steps in Creating the user interface for the login screen.

```
<lc:AuthenticatingApplication
                    xmlns:mx="http://www.adobe.com/2006/mxml"
                    xmlns:lc="http://www.adobe.com/2006/livecycle"
                    xmlns:cc="custComp.*"
                    layout="absolute"
                    loginPage="custComp.CustLoginPage" >
```

6. After the closing `<lc:AuthenticatingApplication>` tag, add the `lc:Desktop` component and bind the session attribute to the session attribute from the `lc:AuthenticatingApplication` component. The Workspace ES application is available as the `lc:Desktop` component.

7. Add the `lc:HelpContext` component to integrate the Workspace help to your application, add the `creationComplete` to the `AuthenticatingApplication` component, and set the `creationComplete` to a value of `"HelpContext.helpContextRoot = parameters.helpURL;"`. This allows you to call the LiveCycle Workspace ES Help that is deployed with LiveCycle ES.

8. Select **File** > **Save** to save your default application file.

9. Select **Project** > **Build Project** to compile the project.

   Note: The Flex application you build cannot be run locally. It must be deployed to a web server.

Example: ***Using a custom login screen component for Workspace ES***

```
<?xml version="1.0" encoding="utf-8"?>
<lc:AuthenticatingApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:lc="http://www.adobe.com/2006/livecycle"
    xmlns:cc="custComp.*"
    layout="absolute"
    loginPage="custComp.CustLoginPage"
    creationComplete="HelpContext.helpContextRoot = parameters.helpUrl;">

    <!-- Connect the application to the deploy LiveCycle Workspace ES Help -->
    <lc:HelpContext/>
```

```
    <!-- Workspace ES application as a component -->
    <lc:Desktop session="{session}" />

</lc:AuthenticatingApplication>
```

Configuring the build.xml file to compile a custom MXML

Because you created your own MXML file, you can configure the build.xml file in the Workspace project to compile your MXML file as the default application instead of the default Main.mxml file. Typically, in Flex Builder, in your project, you right-click the MXML file and select Set as Default Application. However, because build scripts are used, you must modify the build.xml file. Before you perform these steps, ensure that you created an MXML file.

➤ **To configure the build.xml file to compile a custom MXML file:**

1. In the Workspace project, double-click the **build.xml** file to open it for editing.
2. Make the following changes to the build.xml file:
   - For the `application.name` property, change the value to `CustLoginPage`.
   - For the `application.title` property, change the value to `Simple Customization WS`.
   - For the `app.name` property and change the `value` to the name of the MXML file that you created in Creating the application logic in the default application file. For example, if you created an MXML file named CustSimpleStart.mxml file, change the value to `CustLoginPage`.

```
<!-- Modify the following properties to match your setup -->
<property   name="flex.sdk.home"
            location="C:/Program Files/Adobe/Flex Builder 3/sdks/3.0.1.lc"
/>
<property name="lc.sdk.dir"
 location="C:/Adobe/LiveCycle8.2/LiveCycle_ES_SDK" />
<property name="application.title" value="Simple Customization WS"/>
<property name="application.name" value="CustLoginPage"/>
<property name="app.name" value="CustLoginPageWS"/>
```

3. Save your changes.

After you compile and deploy the EAR file, the custom login page should appear when you try to login to your custom application that you created. For example, to access the custom Workspace ES application with a custom login screen in a JBoss turnkey installation, in a web browser, type `http://[servername]:8080/CustLoginPage`, where `[servername]` is the name of your computer. The context root is `CustLoginPage`.

12 Troubleshooting

This section summarizes the common issues that may occur when you customize the LiveCycle Workspace ES user interface and the recommended resolutions.

Issue	Resolutions
Your project does not build.	• Confirm that you built an Ant builder and that you have the new builder enabled in Flex builder. • If you are using only the Flex SDK to compile, you must use ANT instead of the Flex compile commands to create a custom EAR file. • Configuring the Flex compiler may cause the contents of the html-template folder to be overwritten. Recopy the html-template contents from the provided Workspace ES source to the html-template folder in your Workspace project. • Verify that the name of the `app.name` property in the build.xml file is correct.
The image you added to a customized theme file does not appear within Workspace ES.	• Copy the image to the theme\images folder in the Workspace project. • Clear your web browser cache by deleting all online and offline files that are cached.
For a localization file customization, nothing is displayed in the web browser.	• Verify that the locale code (both the language code and the country code) are set properly in your web browser.
The localization file customization does not display the correct language.	• Verify that the localization file is packaged with the EAR file you created. The localization file is in the WAR package and in the locale folder. • Verify that locale code is set to display first. • Verify that the locale extension for your localization file matches the locale you configured in the workspace-config.js file in the html-template folder. • Clear your web browser cache by deleting all online and offline files that are cached.
Images that are used for Workspace ES components in your Flex application do not display properly.	• Almost all images are part of the theme file. If you customized the theme file, verify that you included the Workspace ES default images. • Some images are not compiled as part of the theme file. You can copy the images folder into your project and deploy it as part of the application. You can copy the image directory from the Workspace ES stand-alone after you copy it locally to your computer.

Issue	**Resolutions**
Text disappeared after the colors were changed.	● You may have changed the background color to the same color as the text color. Look for a relevant location to modify the color property to ensure that it is different from the background color.
Workspace ES components do not update with information	● You may not have passed the `lc:SessionMap` object. Each Workspace API object requires setting the `session` attribute to a valid `lc:SessionMap` object.
Display of forms and functions do not seem to work properly in the web browser.	● When you compiled your project, you may not have copied the contents of the html-template directory provided with the LiveCycle ES SDK.
After upgrading, the images that were replaced and the colors and localization changes are missing.	● During an upgrade, the adobe-workspace-client.ear file was replaced. It is recommended that you deploy your customizations into a separate EAR file. You must redeploy your theme file and custom localization files.