Customizing the LiveCycle® Workspace ES User Interface

Adobe® LiveCycle® Workspace ES

August 2008

Version 8.0

Adobe® LiveCycle® ES 8.0 Customizing the LiveCycle® Workspace ES User Interface for Microsoft® Windows®, Linux®, and UNIX®
Edition 1.1, August 2008

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

Contents

About This Document

This document describes how to customize the Adobe® LiveCycle® Workspace ES user interface, including its colors, images, language localization, and layout.

Who should read this document?

This document is intended for programmers who are familiar with Adobe Flex™, ActionScript 3.0, MXML, Adobe XML, and Adobe Flex™ Builder™ (or the Adobe Flex SDK compiler) and who want to customize the LiveCycle Workspace ES user interface. The procedures are written using Flex SDK 2 and Flex Builder 2. You can use Flex Builder 3, some of the compile steps may be different because the SDK folder structure has changed.

To use this document effectively, it is beneficial if you have had exposure to Adobe LiveCycle ES (Enterprise Suite), are familiar with Workspace ES, and have a good understanding of the application server that you plan to use for testing. Examples in this document use a JBoss turnkey installation, which is typical for developer environments.

Before you begin

You can use either of the following to customize the Workspace ES user interface:

- Flex Builder 2.0.1 with hotfix 2 or hotfix 3
- Flex 2.0.1 SDK with hotfix 2 or hotfix 3
- Flex Builder 3

Flex SDK 2.0.1 hot fix 2 is required to compile applications and customizations for the Workspace ES user interface. This document uses a combination of both the command line options (available from the Flex SDKs) and Flex Builder.

You must have access to a LiveCycle ES server from your computer for deployment and testing; otherwise, customizations to Workspace ES cannot be tested.

Additional information

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

For information about	See
LiveCycle ES solution components	*LiveCycle ES Overview*
LiveCycle Workspace ES	*LiveCycle Workspace ES Help*
ActionScript classes and properties included with LiveCycle ES	*LiveCycle ES ActionScript Language*

For information about	See
Classes and methods included with LiveCycle ES SDK	*LiveCycle ES Java API Reference*
LiveCycle ES terminology	*LiveCycle ES Glossary*
Installing Flex Builder 2	http://www.adobe.com/go/flex2_installation
Flex 2.0.1 SDK (hotfix 2)	http://kb.adobe.com/selfservice/viewContent.do?externalId=kb402000&sliceId=2
Flex 2.0.1 SDK download (hotfix 3)	http://labs.adobe.com/technologies/flex/sdk/flex2sdk.html
Installing Flex Builder 3	http://www.adobe.com/go/flex3_installation
Patch updates and technical notes this product version	Adobe Technical Support

1 Introduction

LiveCycle Workspace ES is a Flex application that allows users to initiate and participate in form-based business processes by using a web browser. This document helps you to understand how to customize the Workspace ES user interface. It uses tasks to describe how to perform typical customizations, and uses examples to illustrate them. In addition to customizing Workspace ES, you can build Flex applications by reusing parts of the Workspace ES user interface and functionality that are exposed as ActionScript components by using the Workspace ES API.

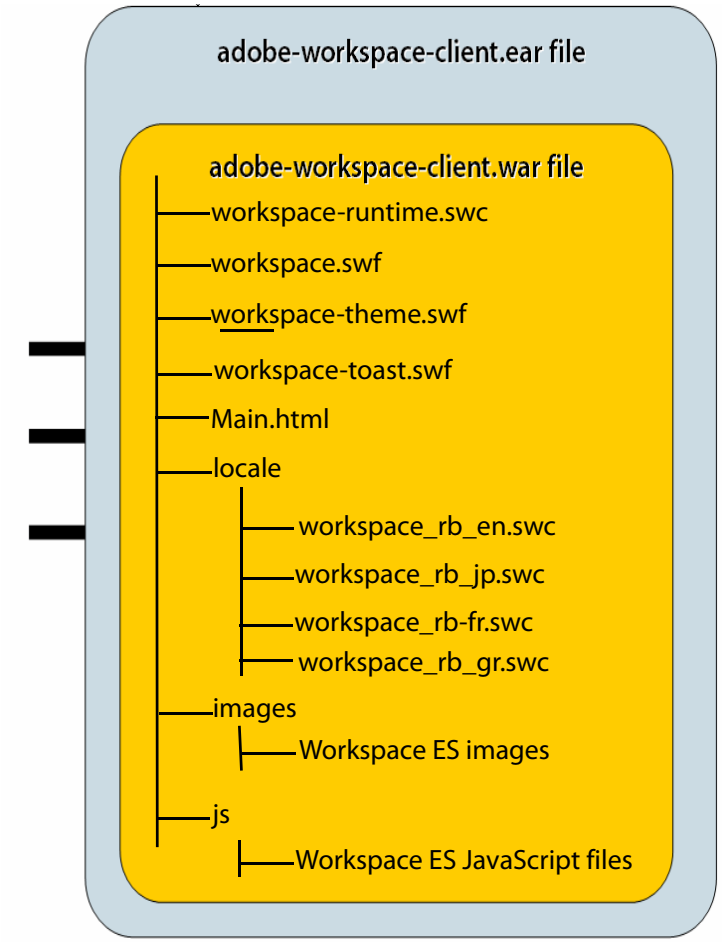
Before you begin your customizations, it is recommended that you understand some of the aspects of Workspace ES:

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

Understanding the Workspace ES EAR file structure

Workspace ES exists as an Enterprise Archive (EAR) file called *adobe-workspace-client.ear* that is deployed to an application server. The adobe-workspace-client.ear file contains the adobe-workspace-client.war file that is deployed to a web server. When you perform customizations to Workspace ES, you should not modify the adobe-workspace-client.ear file or the adobe-workspace-client.war file on your server. Instead, you can create copies of the adobe-workspace-client.ear file, modify it as required for your customizations, create a separate context root, and then deploy it to your server. By deploying your customizations in a separate EAR file, you are not affected by patches or upgrades.

This section describes the contents of the EAR file structure. 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 ES API that are 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 to load and the theme file. Modifying the workspace.swf file is not supported.

workspace-theme.swf: The theme file that contains the CSS and any images that Workspace ES uses. The SWF file is dynamically loaded by the workspace.swf file at startup. To customize the images and

styles that Workspace ES uses, create your own version of the workspace-theme.swf file and replace the existing one in the adobe-workspace-client.war file with the one you created.

workspace-toast.swf: A small application used to display informational messages called *toast messages* in the lower-right corner of Workspace ES. These messages can appear regardless of whether a PDF form, an HTML form, an Acrobat form, or a Flex application is displayed. Modifying the workspace-toast.swf file is not supported.

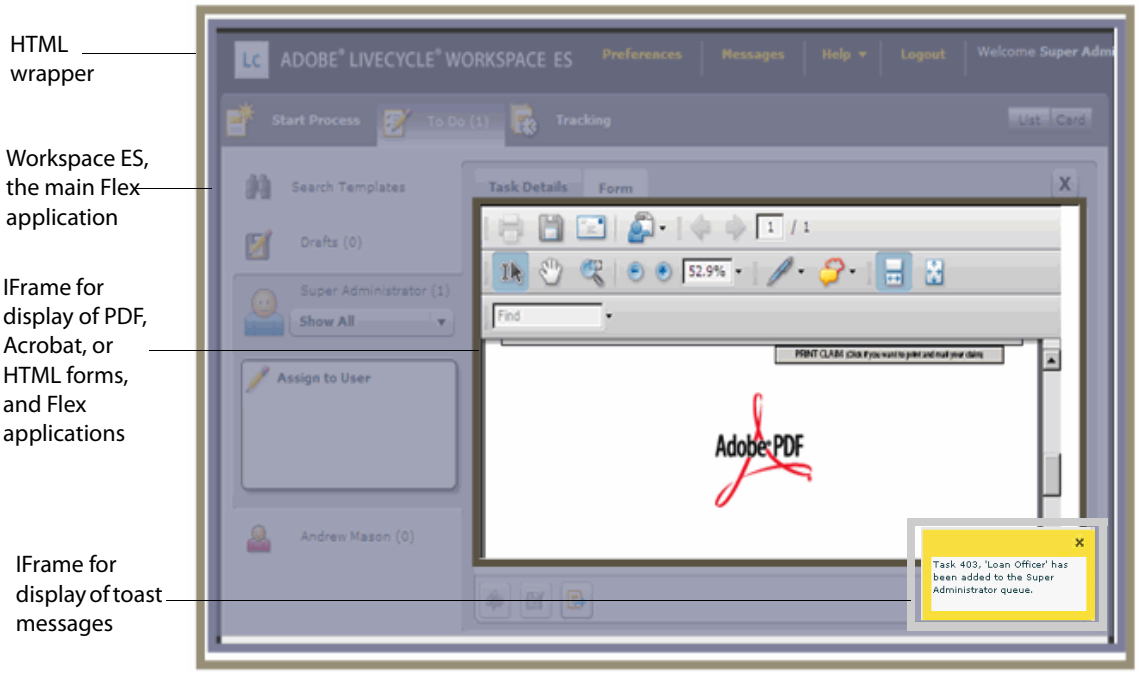
workspace_rb_*[locale]*.swf: The localization files for Workspace ES where *[locale]* specifies the locale and region-specific settings. The appropriate localization file is dynamically loaded by the workspace.swf file. Workspace ES loads the appropriate localization file for each language in the order 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, if the browser settings specify ES-BR, but no matching localization is found, Workspace ES loads ES.

You can customize the existing localization files or create your own localization files and add them to the adobe-workspace-client.war file.

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) and the toast messages. The *Main.html* wrapper provides JavaScript™ and other support for applications that are using Workspace ES 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, Flex applications, and pop-up messages (called *toast messages*)
- 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 ES 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 SWF 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 Workspace ES API architecture

Workspace ES is built by using reusable visual and non-visual components that are exposed as the *Workspace ES API*. In Flex Builder, you may see other components that are available from the Workspace ES API that are not documented in *LiveCycle ES ActionScript Reference*.

Caution: It is recommended that you do not use undocumented components from the Workspace ES 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 ES API components. The Workspace ES 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.

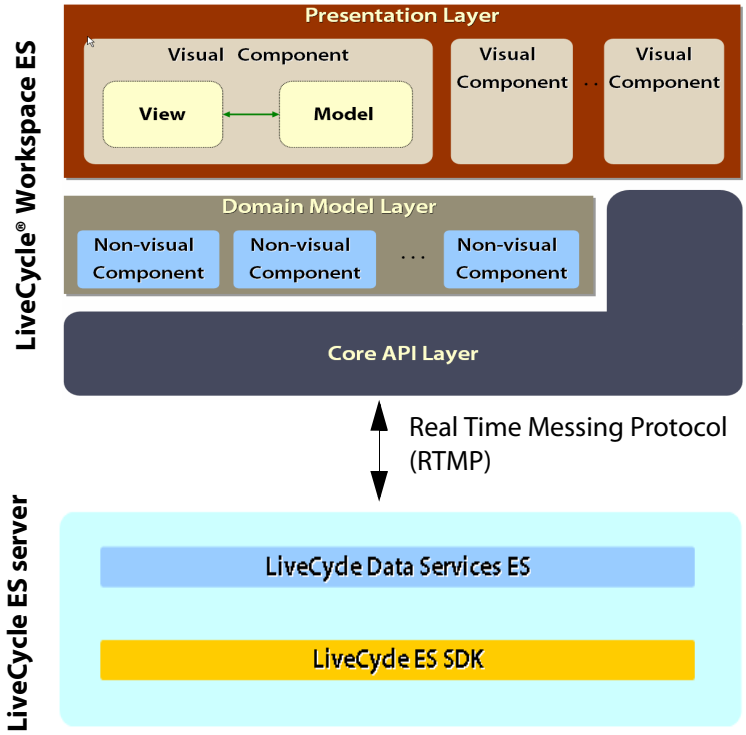
Note: Presentation model components have the same name as a view component but with the word *Model* appended as a suffix.

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 Management API from *LiveCycle ES SDK Help*. LiveCycle Data Services ES facilitates the communication between the Workspace ES APIs and LiveCycle ES SDK. The messaging channel used between the Workspace ES API and Data Services ES is Real Time Messaging Protocol (RTMP) with polling enabled, 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 ES API, and how RTMP and Data Services ES are used to communicate with the LiveCycle ES SDK to provide process management functions. *RTMP* is a TCP/IP protocol designed for high performance transmission of audio, video, and data messages.



The available Workspace ES API components are described in *LiveCycle ES 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. Although you seldom use components from the Core API layer, you may want to implement interfaces or use components to coerce other objects to retrieve information for the Flex application that you create.

Using Workspace ES API components

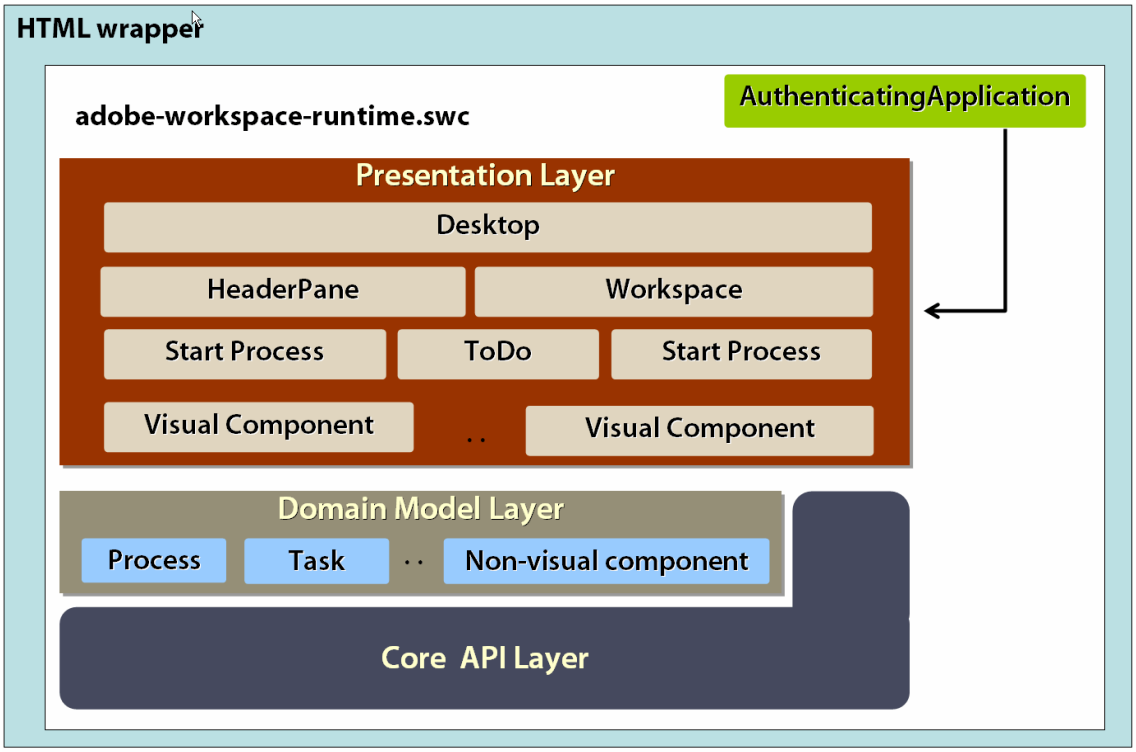
The Workspace ES API contains reusable visual and non-visual components written in MXML and ActionScript that are exposed as the *Workspace ES API*:

- *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 component, consists of several other pieces, such as the `lc:Workspace` and `lc:HeaderPane` components. The `lc:Workspace` component consists of an `lc:StartProcess`, `lc:ToDo process`, and `lc:Tracking` component. 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`.

- *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 ES 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 ES 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 a 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 and, 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 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`, you 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 the

adobe-workspace-runtime.ear file or even build a separate Flex application that reuses components from the Workspace ES 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 the new localization file. (See Localizing LiveCycle Workspace ES.)

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 ES API to perform localization customizations.

Theme: Workspace ES has a cascading style sheet (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 their usage will help 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 Replacing Images and Modifying Colors.)

You do need to use the Workspace ES API to perform theme customization.

Layout: You can use visual components from the Workspace ES 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 Simplifying LiveCycle Workspace ES.)
- To change the layout of Workspace ES by building your own Flex application by using components that are available from the Workspace ES 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 Changing the Layout.)
- To replace certain components with your own. For example, you can replace the default login screen with a login screen that you create. (See Replacing the 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 ES API.

Understanding the source code

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

Modifying and deploying the source code on a production server is not supported. Instead of modifying the source code, it is recommended that you create your own subclasses by extending Workspace ES API components, described in *LiveCycle ES 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. It is not necessary to recompile your customized application provided that the upgraded version of Workspace ES uses the same Flex SDK version. However, it is recommended that you do recompile your customized applications to leverage any resolved product defects and additional features in Workspace ES.

2 Configuring Your Development Environment

This section describes how to configure your development environment for modifying the LiveCycle Workspace ES user interface. The steps provided in this section need to be performed on each computer that you use to customize Workspace ES.

Summary of steps

It is recommended that you complete these high-level tasks to configure your development environment to customize the Workspace ES user interface.

1. Update your version of the Flex SDK. (See Updating the Flex SDK.)
2. Update the Flex SDK with Data Services ES SWC files. (See Updating Flex SDK with Data Services ES SWCs.)
3. Copy the files from the LiveCycle ES SDK folder that are required for the different changes you make that are described in Understanding the Workspace ES EAR file structure. (See Retrieving the files for customizing Workspace ES.)
4. (Optional) Extract the Workspace ES source code to use as a reference for your customization. (See Retrieving the Workspace ES source code.)
5. (Optional) It is recommended that you deselect the option to build automatically because changes you make will need to be deployed to a web server that is connected to LiveCycle ES. In Flex Builder, select Project > Build Automatically to disable the check mark in the menu.

Updating the Flex SDK

You must use a version of the Flex SDK that is compatible with the one that the LiveCycle ES server uses. Update your development environment using the Flex SDK from the LiveCycle ES DVD only if you do not have a compatible version of the Flex SDK. The following versions are compatible with LiveCycle ES:

- Flex Builder 2.0.1 (or the plug-in to Eclipse) with Hotfix 2 or later
- Flex SDK 2.0.1 with Hotfix 2 or later
- Flex Builder 3

Note: You must compile using the Flex SDK version 2.0.1 Hotfix 2 or a later version when using Flex Builder 3. (See Using Flex Builder 3 to compile using Flex SDK 2.0.1.).

If you have projects that were created or compiled using the previous Flex SDK version, it is recommended that you close, reopen, clean, and then rebuild them after you complete the procedure below.

Caution: Do not perform the following procedure if you installed Flex Builder 2.0.1 (or the plug-in to Eclipse) with Hotfix 2 or later, Flex SDK 2.0.1 with Hotfix 2 or later, or Flex Builder 3.

➤ **To update your Flex SDK for Flex Builder 2:**

1. Copy the flex_sdk_2.zip file from the livecycle_dataservices folder on your LiveCycle ES DVD to the computer where you installed Flex Builder 2.0.1

2. Close Flex Builder if it is running.

3. Back up the Flex SDK 2 folder by renaming the folder. For a typical Flex Builder 2 installation, the Flex SDK 2 folder is \Program Files\Adobe\Flex Builder 2, which you can rename to Backup Flex SDK 2.

4. Unzip the flex_sdk_2.zip file from the \lcds\resources\flex_sdk\ folder into the Flex SDK folder located in \Program Files\Adobe\Flex Builder 2. Overwrite existing files with the new versions from the flex_sdk_2.zip file.

   Note: The Flex SDK folder must be named *Flex SDK 2*.

5. Make a backup copy of the Flex Builder lib folder that has all the JARs that are used for compilation. For example, in a typical Flex Builder 2 installation, the lib folder is in \Program Files\Adobe\Flex Builder 2\plugins\com.adobe.flexbuilder.flex_2.0.155577\. If you are using the plug-in version of Flex Builder, it is in the plug-ins folder of your Eclipse installation.

6. Copy the lib folder from the Flex SDK 2 folder and paste it into the Flex Builder plug-in folder. For example, copy the contents lib folder from \Program files\Adobe\Flex Builder 2\Flex SDK 2\ to the \Program Files\Adobe\Flex Builder 2\plugins\com.adobe.flexbuilder.flex_2.0.155577\ folder.

7. Copy the flex-compiler-oem.jar file from the lib folder (the backup copy) you made in step 5 to the lib folder you copied the contents of the new Flex SDK to in step 6.

Updating Flex SDK with Data Services ES SWCs

You must update your Flex SDK 2 version with LiveCycle Data Services ES SWC files to compile the Workspace ES source code and Flex applications that you create using the Workspace ES API. Perform one of the following procedures based on the version of Flex Builder or Flex SDK you use.

➤ **To update your Flex 2 SDK for Flex Builder 2:**

1. Close Flex Builder 2 if it is running.

2. Back up the Flex SDK 2 folder by renaming the folder. For a typical Flex Builder 2 installation, the Flex SDK 2 is in the \Program Files\Adobe\Flex Builder 2 folder, which you can rename to Backup Flex SDK 2.

3. On the application server where LiveCycle ES is deployed, go to the adobe-workspace-runtime-exp.war folder that contains the LiveCycle Data Services ES SWC file. The folder is in the location where the EAR contents are deployed to on your server.

   For example, for a turnkey installation, go to the libs folder in *[installdir]*\Adobe\LiveCycle8\jboss\server\all\tmp\deploy\tmp15107adobe-livecycle-jboss.ear-contents\adobe-workspace-runtime-exp.war\, where *[installdir]* represents the location where LiveCycle ES is installed on a server.

   From the adobe-workspace-runtime-exp.war folder, do these tasks:

   - Go to the WEB-INF\flex\locale\en_US folder and copy the fds_rb.swc file to the en_US folder in the Flex SDK 2\frameworks\locale folder. For example, copy the fds_rb.swc file to the C:\Program Files\Adobe\Flex Builder 2\Flex SDK 2\frameworks\locale\en_US folder. Overwrite the existing fds_rb.swc file if it exists.

- Go to the WEB-INF\flex\libs folder and copy the fds.swc file to the SWCS folder you created in step 2. For example, copy the fds.swc file to C:\Program Files\Adobe\Flex Builder 2\Flex SDK 2\frameworks\libs folder.

➤ **To update your Flex 2 SDK for Flex Builder 3:**

1. Close Flex Builder 3 if it is running.

2. Back up the 2.0.1 folder by renaming the folder. For a typical Flex Builder 3 installation, the 2.0.1 folder is located in the \Program Files\Adobe\Flex Builder 3\sdks folder, which you can rename to 2.0.1.backup.

3. On the application server where LiveCycle ES is deployed, navigate to the adobe-workspace-runtime-exp.war folder that contains the Data Services ES SWC file. The folder is in the location where the EAR contents are deployed to on your server.

   For example, for a turnkey installation, navigate to the libs folder in *[installdir]*\Adobe\LiveCycle8\jboss\server\all\tmp\deploy\tmp15107adobe-livecycle-jboss.ear-contents\adobe-workspace-runtime-exp.war, where *[installdir]* represents the location where LiveCycle ES is installed on a server.

   From the adobe-workspace-runtime-exp.war folder, do these tasks:

   - Navigate to WEB-INF\flex\locale\en_US folder and copy the fds_rb.swc file to the en_US folder in the *[Flex installdir]*Flex Builder 3\sdks\2.0.1\frameworks\locale folder, where *[Flex installdir]* is the location where Flex Builder 3 is installed. For example, copy the fds_rb.swc file to C:\Program Files\Adobe\Flex Builder 3\sdks\2.0.1\frameworks\locale\en_US folder. Overwrite the existing fds_rb.swc file if it exists.
   - Navigate to the WEB-INF\flex\libs folder and copy the fds.swc file to the libs folder in *[Flex installdir]* Flex Builder 3\sdks\2.0.1\frameworks. For example, copy the fds.swc file to C:\Program Files\Adobe\Flex Builder 3\sdks\2.0.1\frameworks\libs folder.

Retrieving the files for customizing Workspace ES

The files you will use to configure your project depend on the type of changes you are making to Workspace ES. You must copy all of the required files to one location on your computer in order to access them when you set up the projects to customize Workspace ES.

You must have access to the files that are installed on the LiveCycle ES server to complete these steps. You will copy a number of files and place them in a location convenient for completing other customizations.

➤ **To retrieve the files:**

1. On the computer where you installed Flex Builder or the Flex SDK for your development environment, create a folder to store all of the files that you may use for customizing the Workspace ES user interface. For example, you can create a folder called *forWSCustomization*.

2. In the for WSCustomization folder you created in step 1, create a subfolder called *SWCS*.

3. Access the LiveCycle ES SDK folder in one of the following locations and perform the following steps:

   Workbench ES installation: Go to the Workspace folder in *[installdir]*\LiveCycle ES\Workbench ES\LiveCycle_ES_SDK\misc\Process_Management\, where *[installdir]* represents where Workbench ES is installed on your computer.

   LiveCycle ES server: Go to the Workspace folder in *[installdir]*\LiveCycle_ES_SDK\misc\Process_Management\, where *[installdir]* represents where LiveCycle ES is installed on a server.

   For example, for a turnkey installation, navigate to the Workspace folder in \Adobe\LiveCycle8\LiveCycle_ES_SDK\misc\Process_Management\.

   - Copy the html-template folder to the folder you created in step 1.
   - Copy the services-config.xml and workspace-builtin.xml files to the folder you created in step 1

   Note: When you create Flex applications by using Workspace ES API components, you must use the contents from the html-template folder that is provided in the LiveCycle ES SDK to get the same functionality.

   - Copy the workspace-theme.swf and workspace-toast.swf files to the html-template folder you copied earlier. For example, copy the files to the forCustomization\html-template folder.
   - Copy the workspace-runtime.swc file to the SWCS folder you created in step 2.

4. Access the locale folder for your preferred language in one of the following locations and copy the workspace_rb.swc file to the SWCS folder you created in step 2.

   Workbench ES installation: Go to the Workspace folder in *[installdir]*\LiveCycle ES\Workbench ES\LiveCycle_ES_SDK\misc\Process_Management, where *[installdir]* represents where Workbench ES is installed on your computer.

   LiveCycle ES server: Go to the Workspace folder in *[installdir]*\LiveCycle_ES_SDK\misc\Process_Management, where *[installdir]* represents where LiveCycle ES is installed on a server.

   For example, for a turnkey installation, go to the Workspace folder in \Adobe\LiveCycle8\LiveCycle_ES_SDK\misc\Process_Management.

   For example, go to locale\en to get the English resources bundle for Workspace ES and copy the workspace_rb.swc file to the forCustomization\SWCS folder.

You should now have a folder structure with the following list of files:

forCustomization
- html-template
  - AC_OETags.js
  - alc_wks_client-html-en.properties
  - history.htm
  - history.js
  - history.swf
  - index.template.html
  - playerProductInstall.swf
  - ResourceWrapper.html
  - swfobject.js
  - workspace-theme.swf
  - workspace-toast.swf
- SWCS
  - workspace_rb.swc
  - workspace-runtime.swc

Retrieving the Workspace ES source code

The Workspace ES source code is provided as a sample to build other Flex applications. After you apply any patches to LiveCycle ES, it is recommended that you download the source code again to leverage any fixes that have been provided.

➤ **To retrieve the source code:**

1. On your computer, create a folder in which to place the files from the archived Workspace ES reference source. For example, create a folder called *wsSource*.

2. Access the LiveCycle ES SDK directory at one of the following locations:

   Workbench ES installation: Go to the Workspace folder in *[installdir]*/LiveCycle ES/Workbench ES/LiveCycle_ES_SDK/misc/Process_Management, where *[installdir]* represents where Workbench ES is installed on your computer.

   LiveCycle ES server: Go to the Workspace folder in *[installdir]*/LiveCycle_ES_SDK/misc/Process_Management, where *[installdir]* represents where LiveCycle ES is installed on a server.

   For example, for a turnkey installation, go to the Workspace folder in /Adobe/LiveCycle8/LiveCycle_ES_SDK/misc/Process_Management.

3. Copy the adobe-workspace-src.zip file to the folder that you created in step 1.

4. Using an archiving utility, extract the contents of the adobe-workspace-src.zip file to the folder you created in step 1.

Using Flex Builder 3 to compile using Flex SDK 2.0.1

You can use Flex Builder 3 to compile applications that you write using the Flex SDK 2.0.1. In Flex Builder 3, the folders where the SDK is installed has changed because Flex Builder 3 can support multiple SDK versions. Flex Builder 3 comes with both the Flex SDK 3 and Flex SDK 2.0.1 Hotfix 3

To access the proper Flex 2.0.1 Hotfix 3, you navigate to the sdks folder found in the location where you installed Flex Builder 3. This is useful to know because you may want to access command line commands described in this document. For example, you can access the command line commands from C:\Program Files\Adobe\Flex Builder 3\sdks\2.0.1\lib. When you use the command-line commands from this guide, you may need to change the path or parameters to reflect the location of the Flex SDK folder. For example, for the localization compile command:

```
compc -locale=es -output=export/es/workspace_rb.swc -source-path
./es "C:/program files/adobe/flex builder 2/flex sdk 2/frameworks/locale/
es_ES" -include-resource-bundles alc_wks_client_msg  alc_wks_client_trace
alc_wks_client_ui SharedResources collections controls core data effects
formatters logging messaging rpc skins states styles utils validators --
```

You would type the following command instead:

```
compc -locale=es -output=export/es/workspace_rb.swc -source-path
./es "C:/program files/adobe/flex builder 3/sdks/2.0.1/frameworks/locale/
es_ES" -include-resource-bundles alc_wks_client_msg  alc_wks_client_trace
alc_wks_client_ui SharedResources collections controls core data effects
formatters logging messaging rpc skins states styles utils validators --
```

➤ **To use Flex Builder 3 to compile using Flex SDK 2.01.**

1. In Flex Builder 3, in the File Navigator view, right-click your Flex project and select **Properties**.
2. In the **Properties for *[name of your project]*** where *[name of your project]* is the name of your Flex project, select **Flex Compiler**.
3. In the Flex SDK version area, select **Use a specific SDK**, and in the drop-down list beside it, select **Flex 2.0.1 Hotfix 3**.

   Note: Flex 2.0.1 Hotfix 3 contains the required SDK components to compile Workspace ES and Workspace ES API components; however, you must still update the SDK with the Data Services ES SWC files. (See Updating Flex SDK with Data Services ES SWCs.)
4. Click **Apply** and then **OK**.

3 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 displayed within Workspace ES. This section describes how to replace images in Workspace ES with your own images.

Nearly all images displayed in Workspace ES can be replaced by modifying the theme file named *workspace-theme.swf*. 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 can compile and deploy your own version of the theme file to display the images you want within Workspace ES. For example, you can display the logo for your organization, as shown in this illustration, instead of the default Workspace ES image.

Summary of steps

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

1. Configure your development environment for customizing Workspace ES. (See Configuring Your Development Environment.)
2. Create a project. (See Creating a project for replacing images.)
3. Add image files to your project. (See Adding custom images to the project.)
4. Change the appropriate areas in the CSS file to use the image. (See Modifying the CSS to replace images.)
5. Compile the theme file. (See Compiling the theme file.)
6. Deploy the theme file. (See Deploying the theme file.)
7. Test the theme file changes. (See Testing the theme file.)

Creating a project for replacing images

After you configure your development environment, in Flex Builder, you create a simple project to create the folder structure and import the necessary files to compile a new theme file. In your project, you create two folders: one stores the files used to compile the new theme file, and the other stores the compiled theme file.

Note: You use the Java integrated development environment (IDE) to create a project because no ActionScript or MXML code is written.

➤ **To create a project:**

1. In Flex Builder, select **File** > **New** > **Other**.
2. In the New dialog box, select **Simple** > **Project** and then click **Next**.
3. In the **Project name** box, type a name, such as `wsImageCust`, and then click **Finish**.
4. In the Navigator view, right-click the project you created in step 3 and select **New** > **Folder**.
5. In the **Folder Name** box, type a name for the folder (for example, type `assets`) and then click **Finish**. This folder will store the Workspace ES images, custom images, and the CSS file to compile a theme file.
6. Right-click the folder you created in step 5 and select **New** > **Folder**.
7. In the New Folder dialog box, in the **Folder name** box, type a name for the folder (for example, type `custImages`), and then click **Finish**. This folder will store your custom images to add to the theme file.
8. Repeat steps 4 and 5 to create a folder for storing the compiled theme file (for example, create a folder called *export*).

9. In the folder where you extracted the Workspace ES source code (see Retrieving the Workspace ES source code), navigate to the \Workspace\src\ folder and copy the following files to the folder you created in step 5.
   - lc.css
   - images folder

Adding custom images to the project

After you create your project for compiling a new theme file, you can add custom images to the project to be 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.

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:**

- Copy your JPG, GIF, or PNG file to the folder where you store the images to compile the new theme file. For example, if you are using Flex Builder, drag the image files to the assets/custImages folder in the Flex Navigator view.

Modifying the CSS to replace images

After you copy the image to your project, you can modify the CSS file named *lc.css*. You modify the entries in the lc.css file to reference images you want displayed in Workspace ES. The lc.css file is compiled with the images into a theme file that is used by the Workspace ES run time.

The *lc.css* file contains the types of classes and style names that the visual components use. Class names refer to specific Workspace ES visual components that Workspace ES uses. For example, to replace the image that the `lc:ToDo` component uses, modify the style for `ToDo`. Some visual components are modified 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 source code; 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), you 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 give a consistent appearance in Workspace ES. (See Modifying Colors.)

Caution: Nearly all 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.

Any images that are not directly referenced in the CSS are dynamically loaded at run time from the images folder in the adobe-workspace-client.war file and can be modified without recompiling the workspace-theme.swf file.

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

1. Open the lc.css file for editing.
2. 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, you can look through the source code.
3. 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.
4. Save the lc.css file.

Example: *Changing the login screen and header pane logos with a logo called financeCorpLogo.png*

```
HeaderPane {
   logo:        Embed('images/financeCorpBanner.png');
   color:       #ffffff;
   backgroundColor:#333333;
}

...
/*

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

-------------------------------------------------------------------------------
*/
.copyright {
   color:       #999988;
}
.loginSplash {
   backgroundImage: Embed('images/financeCorpLogo.png');

}
```

Compiling the theme file

After you make changes to the lc.css file and copy all the required images to your project, you can compile a new theme file.

You use the mxmlc command to compile a theme file named workspace-theme.swf in command prompt. Because the Workspace ES run time explicitly looks for the theme file by name, it is mandatory that the SWF file is named workspace-theme.swf. You compile the theme file by performing using the mxmlc command at the command line to compile the SWC file by using the following command and options:

1. Open a command prompt by selecting **Start** > **All Programs** > **Adobe** > **Flex Builder 2**> **Adobe Flex 2 SDK Command Prompt**.

2. Go to the root level of your project, such as C:\customTheme\.

3. Type the following command:

   `mxmlc -library-path+=C:\[localWSFiles]\SWCS\workspace-runtime.swc -output .\export\workspace-theme.swf [CSS file]`

   `library-path:` The location where the SWC files you copied from the LiveCycle_ES_SDK folder onto your computer and the frameworks.swc file from the Flex SDK. *[FlexSDKDir]* is the location where the Flex SDK is installed on your computer and *[localWSFiles]* is the location where you copied the SWC files from the LiveCycle_ES_SDK folder onto your computer.

   `output:` The location and name of the theme. It is mandatory that the theme file be named workspace-theme.swf.

   `[CSS file]:` The name of the CSS file to compile. By default, you can use lc.css. However, you can copy the content of the lc.css file and rename it to keep different versions the CSS file for compiling different themes.

   For example, if you copied the Workspace ES files locally to the forWSCustomization folder, the following command-line option would be entered at the root level of your project:

   ```
   mxmlc -library-path+=C:\forWSCustomization\SWCS\workspace-runtime.swc
   -output .\export\workspace-theme.swf .\assets\lc.css
   ```

   Note: If you are not using an environment where you have the Flex SDK directory properly configured, you may need to reference the frameworks.swc file; therefore, you would use the following command:

   ```
   mxmlc -library-path+=C:\forWSCustomization\SWCS\workspace-runtime.swc
   +="C:\Program Files\Adobe\Flex Builder 2\Flex SDK\frameworks.swc"
   -output .\export\workspace-theme.swf .\assets\lc.css
   ```

After the mxmlc command finishes, you have a workspace-theme.swf file.

Deploying the theme file

After you compile the theme file (workspace-theme.swf), you can deploy it to a LiveCycle ES server for testing. You deploy your own EAR file based on the adobe-workspace-client.ear file. By separating your theme modifications into a separate EAR file, you can manage your own version of Workspace ES and be less likely affected by upgrades. Deploying a theme SWF file involves the following tasks:

Caution: Before you do these tasks, it is recommended that you create a backup copy of the adobe-workspace-client.ear file.

1. On the application server, go to the location where the adobe-workspace-client.ear is deployed. The location will depend on the application you are running.

   For example, in a JBoss Turnkey installation, you would go to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* is the location of the turnkey installation.

2. Copy the adobe-workspace-client.ear file to another location on the server or to your computer. For example, copy it to a folder named *EarBuild* on the application server.

3. Navigate to the folder you created in step 2 and create a subfolder named *WarBuild*.

4. In the folder you copied the adobe-works-client.ear file to in step 2, extract the EAR file by using an archiving utility. You can also use the `jar` command to extract the contents of the EAR file. For example, in the command prompt, navigate to the EarBuild folder and type `jar -xvf adobe-workspace-client.ear`. After you enter the `jar` command, you will see a META-INF folder and the adobe-workspace-client.war file.

5. Navigate to the META-INF folder and open the application.xml file in a text editor. Edit this XML file to give a context root for accessing your version of Workspace ES, a new web URI that specifies the name of the new WAR file, and a new display name:

   ```
   <display-name>My Workspace</display-name>
   <web-uri>my-customworkspace-client.war</web-uri>
   <context-root>customworkspace</context-root>
   ```

6. Create a new WAR file by reusing the existing adobe-workspace-client.war file and performing the following tasks:

   - Copy the adobe-workspace-client.war to the folder you created in step 3.
   - In the command prompt, navigate to the folder you created in step 3 and extract the contents of the WAR file by using an archiving utility. You can also use the `jar command` to extract the contents of the EAR file. For example, in the command prompt, navigate to the WarBuild folder, and type `jar -xvf adobe-workspace-client.war`.
   - Delete the adobe-workspace-client.war file from the folder you created in step 3.
   - Copy your modified version of the lc.css file, the folder that contains custom images, and the workspace-theme.swf file to the folder you created in step 3. You overwrite the original versions of the lc.css file and the workspace-theme.swf file.
   - Copy the customImages folder that contains your images to the folder you created in step 3.
   - (Optional) In a text editor, modify one of the alc_wks_client_html_*[locale]*.properties files, where *[locale]* represents the locale. For example, for the English locale, you can edit the alc_wks_client_html_en.properties file and modify the value for `browser.document.title` by typing `Custom Theme for Adobe Workspace ES` in place of `Adobe LiveCycle Workspace ES`.

7. Repackage the WAR and EAR files by using an archiving tool or the `jar` command and performing the following tasks:

   - Create the new WAR file with the name you specified as the web-uri from step 5. For example, navigate to the folder where you extracted the WAR file to in step 3, and type `jar -cvf my-customworkspace-client.war META-INF\MANIFEST.MF *`.
   - Move the newly created WAR file to the folder you created in step 2.
   - Create a new EAR file by using the original manifest file, the modified application.xml file, and the new WAR file. For example, navigate to the folder that contains the META-INF folder and the new

WAR file you created, and type `jar -cvfm my-customworkspace-client.ear META-INF\MANIFEST.MF META-INF my_customworkspace-client.war`.

8. Redeploy the new EAR file you created in step 7 to the LiveCycle ES server.

   For example, in a turnkey installation, you can copy the my-customworkspace-client.ear file to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* represents the installation folder.

Testing the theme file

After you redeploy your theme file, you can start Workspace ES and verify that the user interface is using the new images. It is recommended that you verify that the images are the proper size and appear correctly. For example, verify that the screen is using the image you added to the theme file.

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

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

Caution: The original Workspace ES application still exists and is accessible by the default URL. You must undeploy the adobe-workspace-client.ear file from the application server so that it is no longer available.

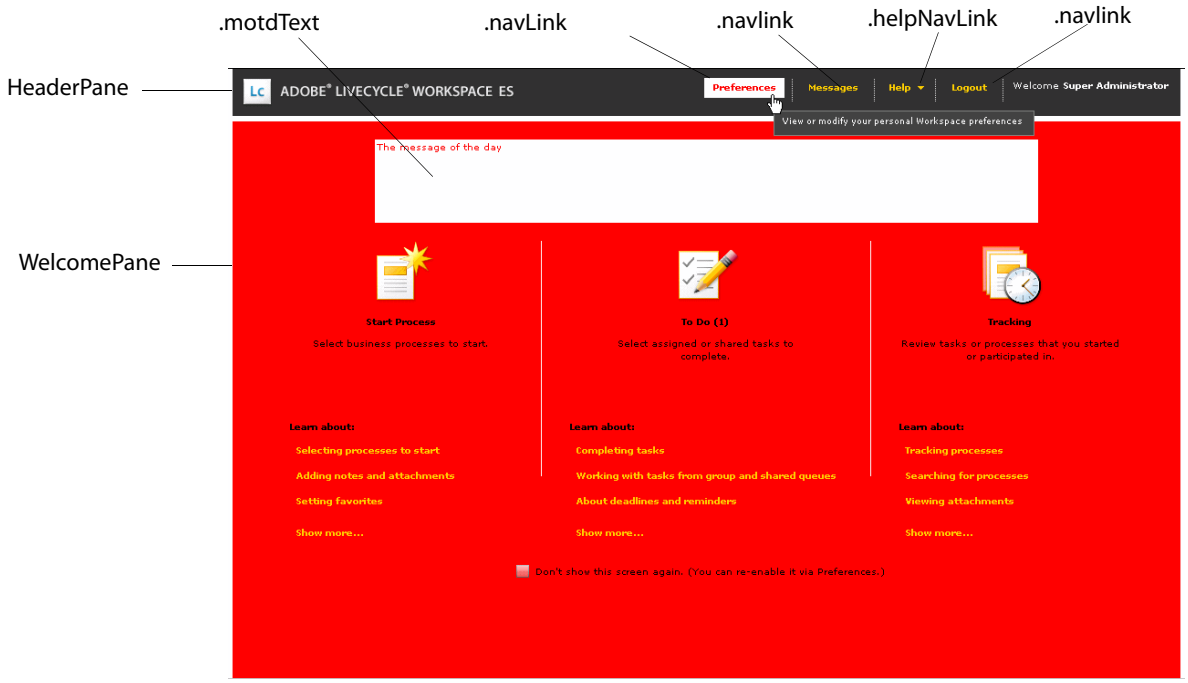
4 Modifying Colors

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

You may want to 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 a theme file named *workspace-theme.swf*. 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 can compile and deploy your own version of the theme file to display the colors you want in Workspace ES.

Workspace ES is built by using multiple visual components that are described in *LiveCycle ES ActionScript Reference*. You can determine the styles' and components' entries to modify in the CSS file by looking at the Workspace ES source code that you copied to your computer. (See Retrieving the Workspace ES source code.) For example, to determine the components that comprise the Welcome screen, you would 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 has kept the default color, the message-of-the-day area has been changed to white, the rollover and text rollover for the `.navLink` style has been changed to red text and white background, and the `lc:WelcomePane` component has 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 tasks to modify the colors in the Workspace ES. The example that accompanies these tasks describes how to change the style and class entries in the CSS file to provide a consistent color of red.

1. Configure your development environment for customizing Workspace ES. (See Configuring Your Development Environment.)
2. Create a project. (See Creating a project for modifying colors.)
3. Modify CSS file colors. (See Modifying colors in the CSS.)
4. Compile the theme file. (See Compiling the theme file.)
5. Deploy the theme file. (See Deploying the theme file.)
6. Test the theme file changes. (See Testing the theme file.)

Creating a project for modifying colors

After you configure your development environment, in Flex Builder, you create a simple project to create the folder structure and import the necessary files to compile a new theme file. In your project, you create two folders: one stores the files used to compile the new theme file, and the other stores the compiled theme file.

Note: You use the Java integrated development environment (IDE) to create a project because no ActionScript or MXML code is written.

➤ **To create a project:**

1. In Flex Builder, select **File** > **New** > **Other**.
2. In the New dialog box, select **Simple** > **Project** and then click **Next**.
3. In the **Project name** box, type a name (for example, type `wsColorCust`), and then click **Finish**.
4. In the **Navigator** view, right-click the project you created in step 3 and select **New** > **Folder**.
5. In the **Folder Name** box, type a name for the folder (for example, type `assets`) and then click **Finish**. This folder will store the Workspace ES images, custom images, and the CSS file to compile a theme file.
6. Repeat steps 4 and 5 to create a folder for storing the compiled theme file (for example, create a folder called *export*).
7. In the folder where you extracted the Workspace ES source code (see Retrieving the Workspace ES source code), navigate to the \Workspace\src\ folder and copy the following files to the folder you created in step 5.
   - lc.css
   - images folder

Note: Although you are not replacing any images, it is necessary to include the images when you compile a theme file.

Modifying colors in the CSS

After you create your project for compiling a new theme file, you can modify the CSS file named *lc.css* Modify the entries in the lc.css file to change the colors displayed in Workspace ES. The lc.css file is compiled with the images into a theme file that the Workspace ES run time uses.

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 source code; 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 text displayed below each task on the Welcome screen is also white and therefore will no longer be visible.

➤ **To change the colors of a screen:**

1. Open the lc.css in your project.
2. Locate and modify the appropriate attributes for the classes and styles. You will need to determine the components that need to modified.

   For example, you may want to change the color of the Welcome screen to red; therefore, to give it a consistent appearance, you need to modify the Application container, the `HeaderPane` component, and the `WelcomePane` component. Therefore, you 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`

3. Save the lc.css file.

You have changed the basic colors for the Welcome screen. The rollover colors for some of the links will still be the original colors. You can modify these 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";
}
...
...
```

Compiling the theme file

After you make changes to the lc.css file and copy all the required images to your project, you can compile a new theme file. Use the `mxmlc` command to compile a theme file named *workspace-theme.swf*. Because the Workspace ES run time explicitly looks for the theme file by name, it is mandatory that the SWF file is named workspace-theme.swf. You compile the theme file by performing the following task.

- Use the `mxmlc` command at command line to compile the SWC file by using the following command and options:

  `mxmlc -library-path+=C:\`*`[localWSFiles]`*`\SWCS\workspace-runtime.swc -output .\export\workspace-theme.swf `*`[CSS file]`*

  `library-path:` The location where the SWC files you copied from the LiveCycle_ES_SDK folder onto your computer and the frameworks.swc file from the Flex SDK. *[FlexSDKDir]* is the location where the

Flex SDK is installed on your computer and *[localWSFiles]* is the location where you copied the SWC files from the LiveCycle_ES_SDK folder onto your computer.

`output:` The location and name of the theme. It is mandatory that the theme file be named workspace-theme.swf.

`[CSS file]:` The name of the CSS file to compile. By default, you can use lc.css. However, you can copy the content of the lc.css file and rename it to keep different versions the CSS file for compiling different themes.

For example, if you copied the Workspace ES files locally to the forWSCustomization folder, the following command-line option would be entered at the root level of your project:

```
mxmlc -library-path+=C:\forWSCustomization\SWCS\workspace-runtime.swc
-output .\export\workspace-theme.swf .\assets\lc.css
```

Note: If you are not using an environment where you have the Flex SDK directory properly configured, you may need to reference the frameworks.swc file; therefore, you would use the following command:

```
mxmlc -library-path+=C:\forWSCustomization\SWCS\workspace-runtime.swc
+="C:\Program Files\Adobe\Flex Builder 2\Flex SDK\frameworks.swc"
-output .\export\workspace-theme.swf .\assets\lc.css
```

Deploying the theme file

After you compile the theme file (workspace-theme.swf), you can deploy it to a LiveCycle ES server for testing, It is recommended that you deploy your own EAR file based on the adobe-workspace-client.ear file. By separating your theme modifications into a separate EAR file, you can manage your own version of Workspace ES and be less likely affected by upgrades. Deploying a theme SWF file involves the following tasks:

Caution: Before you do these tasks, it is recommended that you create a backup copy of the adobe-workspace-client.ear file.

1. On the application server, navigate to the location where the adobe-workspace-client.ear is deployed. The location will depend on the application you are running.

   For example, in a turnkey installation, you would navigate to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* is the location of the turnkey installation.

2. Copy the adobe-workspace-client.ear file to another location on the server or to your computer. For example, copy it to a folder named *EarBuild* on the application server.

3. Navigate to the folder you created in step 2 and create a subfolder named *WarBuild*.

4. In the folder that you copied the adobe-works-client.ear file to in step 2, extract the EAR file by using an archiving utility. You can also use the `jar` command to extract the contents of the EAR file. For example, in the command prompt, navigate to the EarBuild folder and type `jar -xvf adobe-workspace-client.ear`. After you enter the `jar` command, you will see a META-INF folder and the adobe-workspace-client.war file.

5. Navigate to the META-INF folder and open the application.xml file in a text editor. Edit the application.xml file to give a context root for accessing your version of Workspace ES, a new web URI that specifies the name of the new WAR file, and a new display name:

   ```
   <display-name>My Workspace</display-name>
   ```

```
<web-uri>my-customworkspace-client.war</web-uri>
<context-root>customworkspace</context-root>
```

6. Create a new WAR file by reusing the existing adobe-workspace-client.war file by performing the following tasks:
   - Copy the adobe-workspace-client.war to the folder you created in step 3.
   - In the command prompt, navigate to the folder you created in step 3 and extract the contents of the WAR file by using an archiving utility. You can also use the `jar` command to extract the contents of the ear. For example, in the command prompt, navigate to the WarBuild folder, and type `jar -xvf adobe-workspace-client.war`.
   - Delete the adobe-workspace-client.war file from the folder you created in step 3.
   - Copy your modified version of the lc.css file, the folder containing custom images, and the workspace-theme file to the folder you created in step 3. You will overwrite the original versions of the lc.css file and the workspace-theme.file.
   - (Optional) In an text editor, modify one of the alc_wks_client_html_*[locale]*.properties files where *[locale]* represents the locale. For example, for the English locale, you can edit the alc_wks_client_html_en.properties file and modify the value for `browser.document.title` by typing `Custom Workspace` in place of `Adobe LiveCycle Workspace ES`.
7. Repackage the WAR and EAR files by using an archiving tool or the `jar` command by performing the following tasks:
   - Create the new WAR file with the name you specified as the web-uri from step 5. For example, navigate to the folder where you extracted the WAR file to in step 3 and type `jar -cvf my-customworkspace-client.war. META-INF\MANIFEST.MF *.`
   - Move the newly created WAR file to the folder you created in step 2.
   - Create a new EAR file by using the original manifest file, the modified application.xml file, and the new WAR file. For example, navigate to the folder that contains the META-INFO folder and the new WAR file you created, and type `jar -cvfm my-customworkspace-client.ear META-INF\MANIFEST.MF META-INF my_customworkspace-client.war`.
8. Redeploy the new EAR file created in step 7 to the LiveCycle ES server.

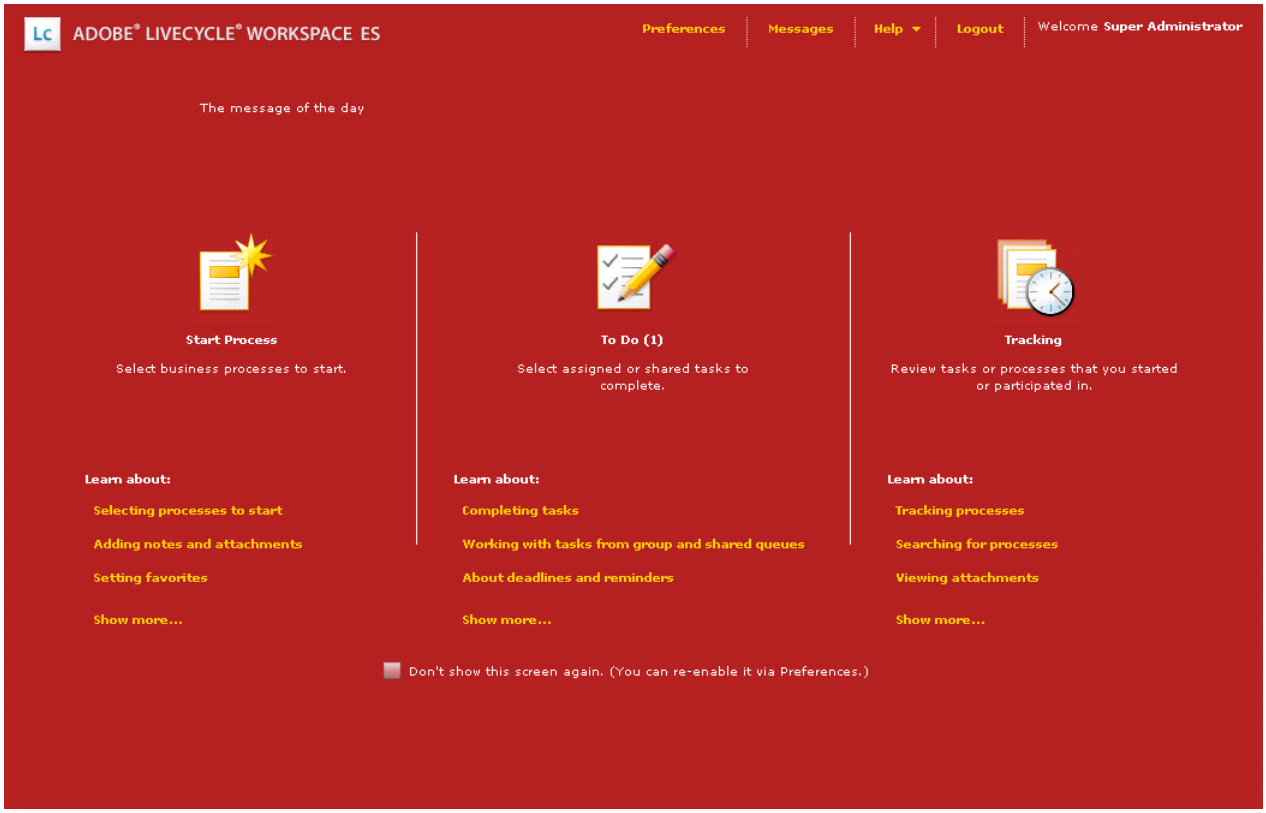
   For example, in a turnkey installation, you can copy the my-customworkspace-client.ear file to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* represents the installation folder.

Testing the theme file

After you redeploy your theme file, you can access your version of Workspace ES and verify that the user interface is using the new colors you specified. Because you deployed your own version of the EAR file with a modified context-root, you will log in to a customized version of Workspace ES. For example, in a turnkey installation, if you deployed an EAR file with a context root named *customworkspace*, you can access your version of Workspace ES by typing the URL `http://[servername]:8080/customworkspace/` in a web browser, where *[servername]* represents the name of the LiveCycle ES server.

Depending on the colors you use, you may notice that some text may not be properly displayed or not visible. For example, if you chose white as the background color, all the default white text is no longer visible.

It is recommended that you also verify that the screens look consistent. For example, if you changed the background color to red, no 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.

5 Localizing LiveCycle Workspace ES

LiveCycle Workspace ES is localized for the English, French, German, and Japanese languages. This section describes how to localize Workspace ES to another language such as Spanish or Chinese. 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.

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

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

The localization text is loaded from individual localization SWF files that are part of the adobe-workspace-runtime.ear file. Workspace ES dynamically loads the appropriate localization SWF file based on the locale specified in the user's web browser settings. If multiple locales are specified, Workspace ES searches, based on the order specified in the web browser settings, for a matching SWF file within the adobe-workspace-client.ear file.

You can provide support for additional languages by creating a localization SWF file and deploying it to Workspace ES. However, due to Adobe Flash® Player limitations, you can only customize Workspace ES languages that are displayed from left to right and from top to bottom.

When you want to create a new locale, you modify properties files, which contain the strings displayed in Workspace ES. You compile the properties files into a SWF file that you deploy on the LiveCycle ES server.

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

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

Note: This section describes how to localize only the Workspace ES client. To complete localization, you should localize the text in the queuesharing.lca file in Workbench ES and localize the LiveCycle Workspace ES Administration found in the LiveCycle Administration Console.

Summary of steps

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

1. Set up your development environment. (See Configuring Your Development Environment.)
2. Create a project. (See Creating a project for localization customizations.)
3. Modify the strings in the properties files as required. (See Modifying the properties files.)
4. Create and rename a localization SWF file. (See Compiling the localization SWF file.)
5. Deploy the localization SWF file to Workspace ES. (See Deploying the localization SWF file.)
6. Test the localization SWF file in Workspace ES. (See Testing the localization file.)

Creating a project for localization customizations

After setting up your development environment, you must create a project and copy the properties files to the project. A localization SWF file is compiled by using properties files. The properties files for the English, French, German, and Japanese languages are available in the Workspace ES source code that is provided as a reference when you deploy the LiveCycle ES Samples. You must copy these properties files to a new project to modify them for a custom locale. You can extract the source by following the steps described at Retrieving the Workspace ES source code.

➤ **To create a project:**

1. In Flex Builder, select **File** > **New** > **Other**.
2. In the New dialog box, select **Simple** > **Project** and then click **Next**.
3. In the **Project name** box, type a name (for example, type `wsLocaleCust`), and then click **Finish**.
4. In the Navigator view, right-click the project you created in step 3 and select **New** > **Folder**.
5. In the **Folder Name** box, type a name for the folder, such as `es`, and then click **Finish**. This folder will store properties files for creating a new localization file.

6. Repeat steps 4 and 5 to create a folder for storing the compiled localization file. For example, create a folder called *export*. After you complete this step, you should have a folder structure that looks like the following illustration:

7. From the folder where you extracted the Workspace ES source code as a reference (see Retrieving the Workspace ES source code), copy the following files from the Workspace\src folder to the folder you created in step 5 to store the properties files:
   - alc_wks_client_msg.properties
   - alc_wks_client_trace.properties
   - alc_wks_client_ui.properties

Modifying the properties files

After creating a project, you must modify the strings within the properties files that Workspace ES uses to to display error messages, tracing messages, and text in the user interface. These properties files can be modified to create a new localization SWF file.

In each properties file, a list of keys are paired with each text string. The keys are used internally by Workspace ES to determine the 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 ES API use. You must not modify this value.

`[string]:` A string that consists of the text that 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_trace.properties: Workspace ES uses this file for tracing and debugging, and is usually not modified. For example, the following key and string (consisting of both text and tokens) are used to trace the time it takes for a transaction to occur:

```
ALC-WKS-008-000 = {0} completed in {1} seconds.
```

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 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 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. In place of the text `Invalid user name or password`, type 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]
```

Compiling the localization SWF file

The localization SWF file contains the resources that Workspace ES requires to display text for the user interface, error messages, and tracing messages. Workspace ES dynamically determines the localization SWF file to use based on the web browser settings.

After editing the properties files for the new locale, you must compile the localization SWF file by using the `compc` command-line command. For information about compiling resources, see *Flex 2 Developer's Guide*.

The SWC file that you create contains two files: library.swf and catalog.xml. You must extract the library.swf file from the SWC file and rename the file. The value that you use for *[locale]* is the same value that is used to identify the language in the web browser.

When Workspace ES loads, it attempts to find a localization SWF file with a specific locale suffix to match the locale specified in the web browser settings. You compile a localization file by performing the following task.

Note: If the Flex SDK does not have a locale for the language you are localizing to, you can copy an existing locale folder and translate each properties file to the language you are localizing to. For example, if you are localizing to Spanish, you create a new locale folder, such as es_ES, under *[Flex installation]*/Flex SDK 2/frameworks/locale/, where *[Flex installation]* represents the location where the Flex SDK is installed. You copy the contents of the en_US locale folder to the es_ES folder, and then, in each .properties file, translate each value to the right of the equal sign to Spanish.

1. Use the `compc` command line to compile the SWC file by using the following parameters:

   `locale:` The language code of the locale that you are localizing to and that maps to the value the user sets in the browser settings. For example, specify `es` for Spanish.

   `output:` The file name in which the new localization SWC file is saved to and folder, which the file SWC file is saved to.

   `source-path:` The source properties to be compiled. You must include the location of these two folders:

   - The modified properties files from Workspace ES.
   - The locale folder for the language you are localizing to in the Flex SDK 2, which is a collection of properties files.

   `include-resource-bundles:` The resource bundles to include in your localization SWF file. You must include these bundles:

   - alc_wks_client_msg
   - alc_wks_client_trace
   - alc_wks_client_ui
   - SharedResources
   - collections
   - controls
   - core
   - data
   - effects
   - formatters
   - logging
   - messaging
   - rpc
   - skins
   - states
   - styles
   - utils
   - validators

   For example, you can specify the following command line to compile a SWC file for Spanish. The following command assumes that you installed Flex Builder in the default location, created a new locale folder named es_ES, and copied the files from the LiveCycle ES SDK folder to the \WSCustomization\SWCS folder on your computer:

   ```
   compc -locale=es -output=export/es/workspace_rb.swc -source-path
   ./es "C:/program files/adobe/flex builder 2/flex sdk 2/frameworks/locale/
   es_ES" -include-resource-bundles alc_wks_client_msg  alc_wks_client_trace
   alc_wks_client_ui SharedResources collections controls core data effects
   formatters logging messaging rpc skins states styles utils validators --
   ```

   Note: If the command fails, you may not have an es_ES folder created under your locale folder.

2. Using an archiving utility, open the SWC file you created. For example, open the export/es/workspace_rb.swc. file

   Note: You may need to change the file name extension to use an archiving utility. For example, change the .swc extension to .zip.

3. Extract the library.swf file and rename the file to *workspace_rb_[locale].swf*, where *[locale]* is the language code.

   For example, to create a localization SWF file for the Spanish locale, rename the library.swf file to workspace_rb_es.swf. However, if you want support for Spanish in Columbia, rename the file to workspace_rb_es-co.swf.

Deploying the localization SWF file

After you compile the localization file, you can deploy it to a LiveCycle ES server for testing. It is recommended that you deploy your own EAR file based on the adobe-workspace-client.ear file. By separating your localization modifications into a separate EAR file, you can manage your own version of Workspace ES and less likely be affected by upgrades. Deploying a localization SWF file involves the following tasks:

Caution: Before you do these tasks, it is recommended that you create a backup copy of the adobe-workspace-client.ear file.

1. On the application server, navigate to the location where the adobe-workspace-client.ear is deployed. The location will depend on the application you are running.

   For example, in a turnkey installation, you would navigate to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* is the location of the turnkey installation.

2. Copy the adobe-workspace-client.ear file to another location on the server or to your computer. For example, copy it to a folder named *EarBuild* on the application server.

3. Navigate to the folder you created in step 2 and create a subfolder named *WarBuild*.

4. In the folder that you copied the adobe-works-client.ear file to in step 2, extract the EAR file by using an archiving utility. You can also use the `jar` command to extract the contents of the EAR file. For example, in the command prompt, navigate to the EarBuild folder and type `jar -xvf adobe-workspace-client.ear`. After you enter the `jar` command, you will see a META-INF folder and the adobe-workspace-client.war file.

5. Navigate to the META-INF folder and open the application.xml file in a text editor. Edit the application.xml file to give a context root for accessing your version of Workspace ES, a new web URI that specifies the name of the new WAR file, and a new display name:

   ```
   <display-name>Localized Workspace</display-name>
   <web-uri>my-localizedworkspace-client.war</web-uri>
   <context-root>localizedworkspace</context-root>
   ```

6. Add new content to the WAR file by performing the following tasks:

   - Copy the adobe-workspace-client.war to the folder you created in step 3.
   - In the command prompt, navigate to the folder you created in step 3 and extract the contents of the WAR file by using an archiving utility. You can also use the `jar` command to extract the contents of the EAR file. For example, in the command prompt, navigate to the WarBuild folder and type `jar -xvf adobe-workspace-client.war`.

- Delete the adobe-workspace-client.war file from the folder you created in step 3.
- In Windows Explorer, navigate to the locale folder. You will see other workspace_rb_[*locale*].swf files.
- Copy the newly created workspace_rb_[*locale*].swf file to the locale folder.
- In a text editor, create a copy of one of the alc_wks_client_html_*[locale]*.properties files, where *[locale]* represents the locale, and modify the strings to match the localization customization you are performing. Rename the file with the appropriate locale setting.

  For example, for the Spanish locale, you can copy the alc_wks_client_html_en.properties file, rename it to alc_wks_client_html_es.properties, type Spanish translations for the values of `browser.document.title` and `no.flash.player`, and then save the file .

7. Repackage the WAR and EAR files by using an archiving tool or the `jar` command by performing the following tasks:
   - Create the new WAR file with the name you specified as the web-uri from step 5. For example, go to the folder where you extracted the WAR to in step 3 and type the following text:

     ```
     jar -cvfm my-localizedworkspace-client.war META-INF\MANIFEST.MF *
     ```

   - Move the newly created WAR file to the folder you created in step 2.
   - Create a new EAR file by using the original manifest file, the modified application.xml file, and the new WAR file. For example, go to the folder that contains the META-INFO folder and the new WAR file you created, and type the following text:

     ```
     jar -cvfm my-localizedworkspace-client.ear META-INF\*
     my_localizedworkspace-client.war
     ```

8. Redeploy the new EAR file created in step 7 to the LiveCycle ES server.

   For example, in a turnkey installation, you can copy the my-localizedworkspace-client.ear file to the *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* represents the installation folder.

Testing the localization file

After deploying the localization file, you must test to ensure that your changes appear correctly. 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 in the Workspace ES user interface.

Each time Workspace ES loads within a web browser, it searches for a localization SWF file that matches the locale(s) specified in the web browser settings. The languages, which are selected, are based on individual web browser settings and the available SWF files in the adobe-workspace-client.war. For example, your web browser settings may specify that Spanish is the first preferred language; however, if a matching localization SWF file is not available, the next language is selected.

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

The localization SWF file contains the resources that Workspace ES requires to display text for the user interface, error messages, and tracing messages. Workspace ES dynamically determines the localization SWF file to use based on the web browser settings.

Caution: To see your changes, you may need to delete the 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 S**elect a language to add** list, select the language that corresponds to the new localized SWF file and click **Add**. For example, if you created a localization file for Spanish, you would 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 changes.

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

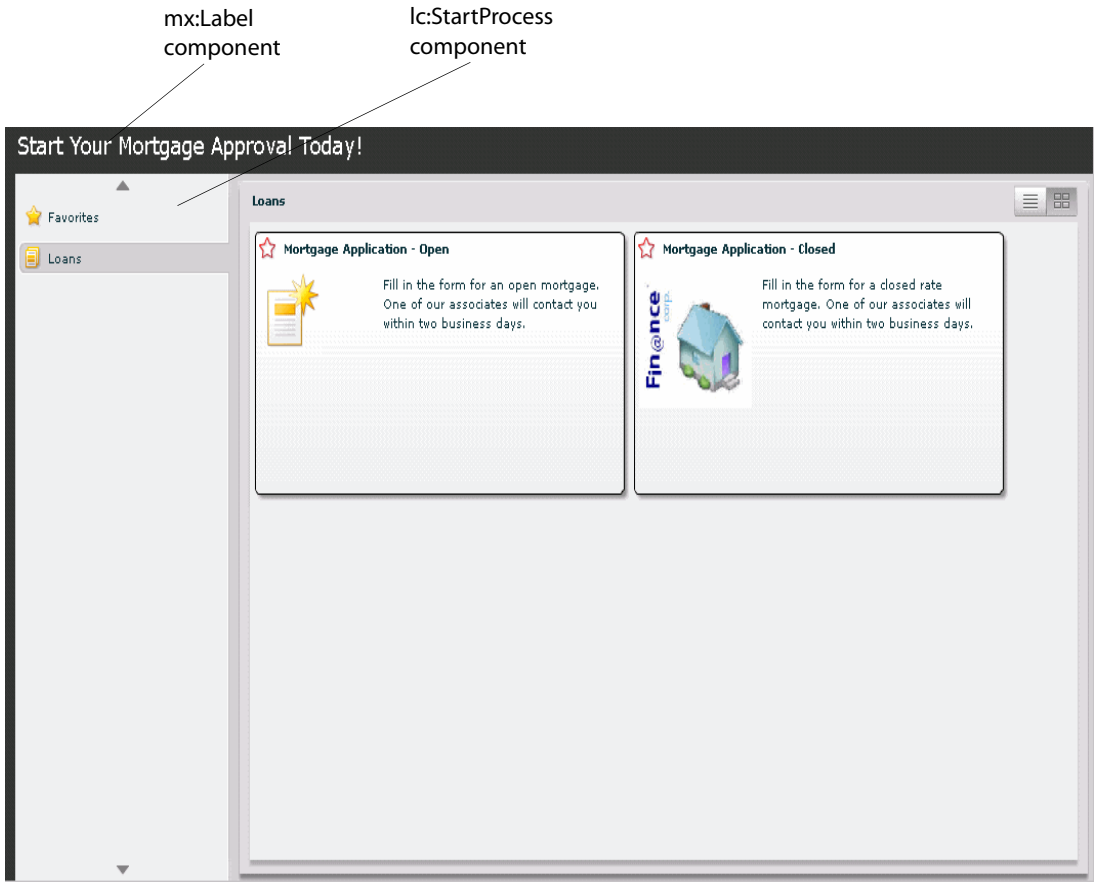
1. Start Internet Explorer and select **Tools** > **Internet Options**.
2. On the **General** tab, click **Languages**.
3. In the Languages Preference dialog box, click **Add**.
4. In the Add Language dialog box, select the language that corresponds to the new localized SWF file and click **OK**.
5. 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 Spanish, you would 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 your changes.

6 Simplifying LiveCycle Workspace ES

LiveCycle Workspace ES is built using self-contained visual components and non-visual components collectively called the *LiveCycle Workspace ES API*, which provides process management functions as described in *LiveCycle ES ActionScript Reference*. Various Workspace ES components expose different parts of the Workspace ES user interface and functionality to your users. Some components provide all the functions required for a specific task. 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, or you may use Workspace ES between multiple organizations and need to transcend traditional organizational boundaries. You may not want your users to see the To Do tab because it allows them to participate in business processes. These users can be customer service representatives in your company or even external users, such as applicants for loans.

You can create a separate Flex application that provides users access to a limited portion of the Workspace ES user interface to only start the processes. In this case, you need to create a Flex application that uses the `lc:StartProcess` component, which contains all the necessary 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 to start an approval process for a mortgage.



Summary of steps

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

1. Configure your development environment for customizing Workspace ES. (See Configuring Your Development Environment.)
2. Create a project. (See Creating a Flex project for building a simplified Workspace ES.)
3. Create the application logic and compile the Flex application. (See Creating the application logic for a simplified Workspace ES.)
4. Deploy the simplified Workspace ES to the LiveCycle ES server. (See Deploying a simplified Workspace ES.)
5. Test the simplified Workspace ES. (See Testing a simplified Workspace ES.)

Creating a Flex project for building a simplified Workspace ES

After you configure your environment, you can use the visual components from the Workspace ES API to create a Flex application for simplifying the Workspace ES user interface.

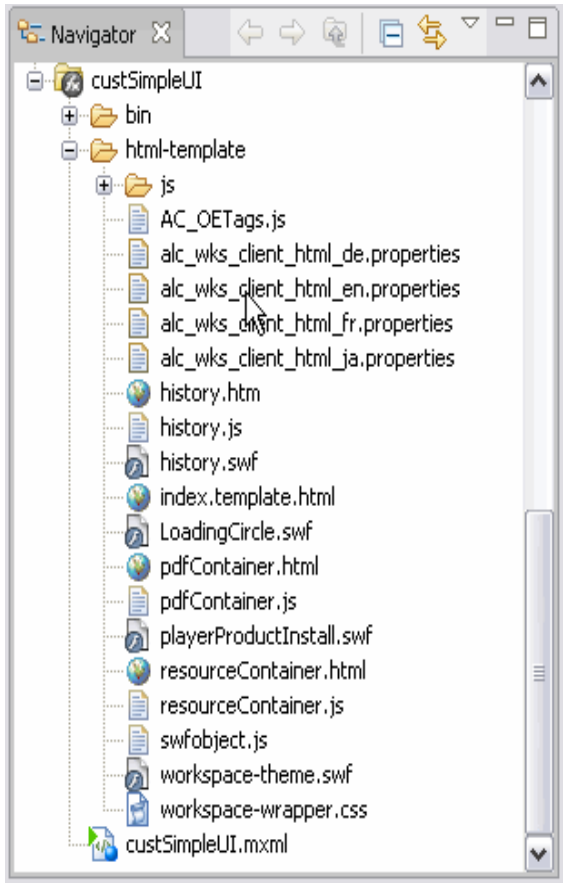
You must configure your Flex application to reference the workspace-runtime.swc, workspace_rb.swc, and fds.swc files before you start coding your application logic by using Workspace ES components. You must also create a namespace in your default application file to access classes, components, and interfaces from the Workspace ES API.

➤ **To create a project:**

1. In Flex Builder,select **File** > **New** > **Flex Project**.
2. In the New Flex Project dialog box, select **Basic (for example, XML or web service from PHP/JSP/ASP.NET)** and click **Next**.
3. In the New Flex Project dialog box, in the **Project Name** box, type a name for the Flex project. For example, type `custSimpleUI`.
4. In the Project Contents area, perform one of following tasks, and then click **Next**.
   - To save your project to a default location, verify that **Use Default Location** is selected.
   - Deselect **Use Default Location**, click **Browse**, select the folder to which to save your project, and then click **OK**.
5. In the New Flex Project dialog box, click the **Library Path** tab and click **Add SWC**.
6. In the Add SWC dialog box, click **Browse** and select the workspace-runtime.swc file that you copied to your computer, and then click **OK**.
7. Repeat steps 5 and 6 to add the workspace_rb.swc and fds.swc file to your Flex project.
8. When you return to the New Flex Project dialog box, click **Finish**.
9. In the Navigator view, locate your new Flex project, and right-click **html-template** and select **Delete**.

10. Drag the html-template folder that you copied earlier to your computer from the LiveCycle_ES_SDK folder to your new Flex project.

    The contents of your Flex project and the html-template folder should look similar to the following illustration.



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 ES API. By default, a Flex project is created with an `mx:Application` container. The Workspace ES 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 required to communicate with the LiveCycle ES server.

➤ **To create the application logic:**

1. In Flex Builder, in the default application MXML file (for example, custSimpleUI.mxml) and in the default `<mx:Application>` tag, add the LiveCycle ES namespace by typing `xmlns:lc="http://www.adobe.com/2006/livecycle"` as an attribute.

2. Replace the `<mx:Application>` tag with `lc:AuthenticatingApplication`.

3. To add instructions to the screen, perform the following tasks:
   - Add an `mx:VBox` container.
   - Under the opening `<mx:VBox>` tag, add an `mx:Label` component.
4. 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.
5. Select **File** > **Save** to save your default application file.
6. Select **Project** > **Build Project** to compile your project.

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

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"
   layout="absolute">

   <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>
```

Tip: To include any changes to the theme file, such as colors or images, you can copy the workspace-theme.swf file to the html-template folder in your project before you build your project.

Deploying a simplified Workspace ES

Although you can build your Flex application like any other Flex application, you will notice that it cannot run locally on your computer. After you compile your Flex application, you must deploy it to a web server that is connected to a LiveCycle ES server. The specific tasks you do to deploy your application depend on the web server that you use and how you have that web server configured with LiveCycle ES. For example, if you are using a turnkey installation of the LiveCycle ES server, you can deploy to the available Apache Tomcat web server.

It is important to include the following files when you deploy your Flex application:

- All the files in your output folder from your Flex application. In a typical Flex project configuration, this folder is named *bin*.
- (Optional) For a modified theme file, you must deploy your version of the workspace-theme.swf file.

- (Optional) To display the Flex application in different languages or to use a localization file you created, you must copy the locale directory to the folder where you deployed your Flex application. The locale folder is available from the adobe-workspace-client.war file from within the adobe-workspace-client.ear file. For example, in a turnkey installation of LiveCycle ES server, you would navigate to *[installdir]*\jboss\server\all\deploy folder where *[installdir]* is the location of the turnkey installation.

Tip: You can also copy the locale folder from where Workspace ES is deployed on your web server. For example, in a turnkey installation of the LiveCycle ES server, you can copy *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\WS.war\locale folder to the folder where you are deploying your Flex application, where *[installdir]* is the location of the turnkey installation.

Deploying a Flex application involves the following tasks:

1. On the web server connected to your installation of LiveCycle ES, navigate to the location to deploy your files. For example, in a 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 a turnkey installation of the LiveCycle ES server.

2. Create a new folder for your Flex application. For example, you can create a folder named *custSimpleUI* in the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

3. From the Flex project where you built your Flex application, copy files that were created after a successful build to the folder you created in step 2. For example, copy the contents from the bin folder to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custSimpleUI\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

4. (Optional) Copy a modified theme file to the folder you created in step 2. For example, copy the workspace-theme.swf file from your computer to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custSimpleUI\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

5. (Optional) Copy the locale folder to the folder you created in step 2. You can copy any localization files that you created to the new locale folder. For example, if you created a localization file, you can copy it to *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custSimpleUI\locale, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

Testing a simplified Workspace ES

You can test your Flex application after you successfully deploy it to a web server that is connected to LiveCycle ES. You access your Flex application by using a URL that includes the name of the server that you deployed your Flex application to and the name of the default application from your Flex project.

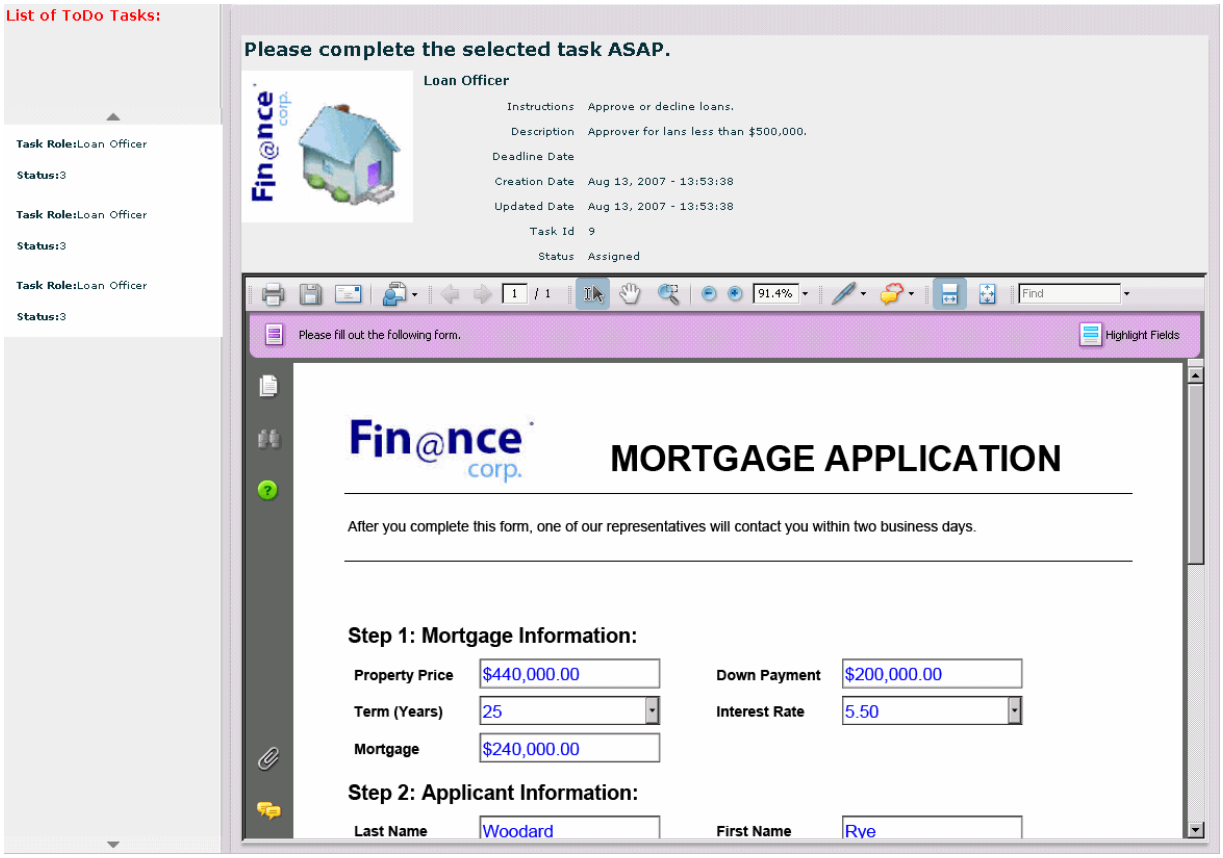
For example, for a turnkey installation of LiveCycle ES, you may have deployed your Flex application to the custSimpleUI folder in *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\. The application file is named *custSimpleUI.mxml*; therefore, in a web browser, you would type `http://[servername]:8080/custSimpleUI/custSimpleUI.html`, where `[servername]` is the name of the server where LiveCycle ES is installed.

7 Changing the Layout

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

To leverage users existing knowledge and reduce training costs, many organizations create custom applications that have a similar appearance to applications that their users are familiar with.

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 ES API to create a three-pane view that shows a user's assigned tasks, task image and details, and the form 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 ES API components in order 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 the 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 Your Development Environment.)
2. Create a project. (Creating a Flex project to create a custom layout.)
3. Create the application logic. You may need to create custom view and presentation model components. (See Creating the application logic for a custom layout.)
4. Deploy the Flex application to the LiveCycle ES server. (See Deploying the custom layout to a web server.)
5. Test the Flex application. (See Testing the custom layout.)

Creating a Flex project to create a custom layout

After you have configured your development environment, you can use the Workspace ES API to create a Flex application with a custom layout. You must configure your Flex application to reference the workspace-runtime.swc, workspace_rb.swc, and fds.swc files before you start creating your application logic using Workspace ES components and create folders to store custom components you create.

➤ **To create a project:**

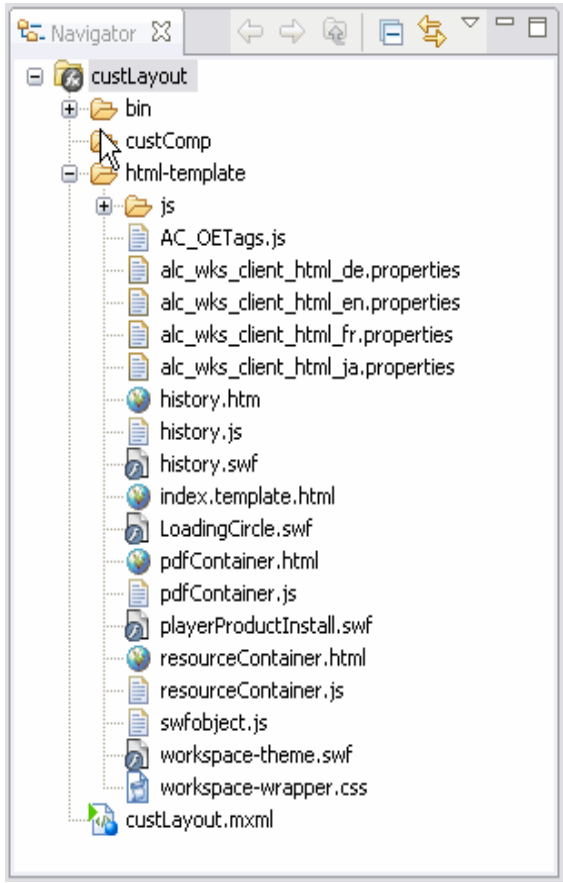
1. In Flex Builder, select **File** > **New** > **Flex Project**.
2. In the New Flex Project dialog box, select **Basic (for example, XML or web service from PHP/JSP/ASP.NET)** and click **Next**.
3. In the New Flex Project dialog box, in the **Project Name** box, type a name for the Flex project. For example, type `custLayout`.
4. Perform one of following tasks in the Project Contents area, and then click **Next**:
   - To save your project to a default location, select **Use Default Location**.
   - Click **Browse** and select the folder to save your project to, and then click **OK**.
5. In the New Flex Project dialog box, click the **Library Path** tab and then click **Add SWC**.
6. In the Add SWC dialog box, click **Browse** and select the workspace-runtime.swc file that you copied to your computer, and then click **OK**.
7. Repeat steps 5 and 6 to add the workspace_rb.swc and fds.swc files.
8. When you return to the New Flex Project dialog box, click **Finish**.
9. In the Navigator view, locate your new Flex project, right-click **html-template**, and then select **Delete**.

10. Drag the html-template folder that you copied earlier to your computer from the LiveCycle_ES_SDK folder to your new Flex project. The html-template and its contents should be copied to your Flex project.

11. In the Navigator view, right-click your Flex project and select **New** > **Folder**.

12. In the New Folder dialog box, in the **Folder name** box, type the name to store components that you create and then click **Finish**. For example, type `custComp`.

    The contents of your Flex project and the html-template folder should look similar to the following illustration.



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 ES 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.

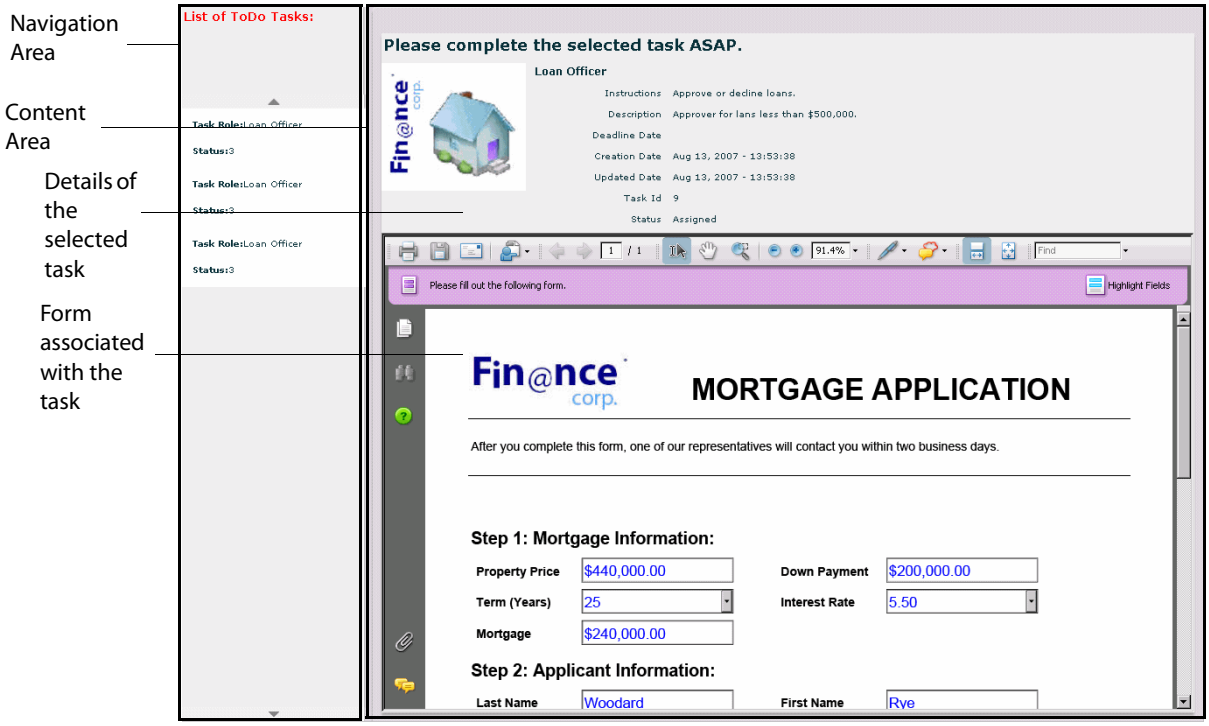
To create a new layout, you will need to use components from the Presentation layer, Domain Model layer, and Core API layer. For example, to work with tasks, you use the `lc:Task` component (Domain Model layer component) and to work with the navigation area, you use the `lc:EmbossedNavigator` component (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, you create a new layout by using multiple custom visual components that consist of view and presentation model components that your default application file uses like any other component in the Presentation layer. By organizing your code into separate view and presentation model components, it promotes future code reuse and ease in testing the code. The new layout is a three-pane interface that contains the following areas:

Navigation area: 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 is displayed in the content area.

Content area: Displays the selected item from the navigation area. For example, in the following illustration, the details of the task are in the topmost area and the form 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 will display 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 to which the task belongs, 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 presentation model component for the content area.
2. Creating a view component for the content area.
3. Creating the presentation model component for the navigation area

4. Creating a view component for the navigation area.

5. Creating the default application for a custom layout.

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 ES API to retrieve information about details of the selected task and also the form associated with the task.

When you create a presentation model, you need to make the properties and methods that store and access data respectively, bindable. 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, you need to 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 the name for the package. For example, type `custComp`.

3. In the **Name** box, type in a name for the class. For example, type `CustTaskFormDisplayModel`.

4. Beside the **Superclass** box, click **Browse** and select **PresentationModel - lc.presentationmodel**, 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`

```
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 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 currently associated with the selected task. For example, create a variable named `currTaskForm`.
   - `String`: Objects that store the name of the step and any associated errors. For example, create variables named `currStepName` and `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 currStepName:String;
   public var currTaskInfoModel:TaskInfoModel;
   public var currTaskImageModel:TaskImageModel;
```

4. Create a private function that takes a parameter of type `WorkspaceFaultEvent` and returns `void`. The purpose of the function is to handle 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`. The other purpose of this setter method is to handle the currently selected task from the navigation area and to retrieve the information for the selected task and the associated form.The following tasks are required for the setter method function:
   - 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 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 would look 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; however, for convenience, you can the use the `bindProperty` method, which is inherited from `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.
   - When the `lc:Task` object or string for the step name is updated, the `currStepName` is updated.

   For example, the function would look similar to the following code if you named the information in the 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");
   bindProperty(this, "currStepName",
                this, ["currTask", "currStepName"]);
}
```

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 display name of the currently selected task.
      public var currStepName:String;
      //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");
          bindProperty(this, "currStepName",
                       this, ["currTask", "currStepName"]);
        }

      }
    }
```

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 presentation model 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, you need to 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 `CustTaskFormDisplayModel`.
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 steps 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 to use the Workspace ES API components and custom components 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 to access the authenticated session information required to use the Workspace ES 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 associated with the process for the task and the details of the selected task. The bottom pane displays the form 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 a `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 will display the form associated with the selected task.

   Your code may appear as follows:

```
    <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 it to a view component:

   - `session`: Bind the attribute to the `lc:SessionMap` component you created in step 2.

- `TaskImageModel`: Bind the attribute to the `model` attribute of the `lc:TaskImage` object you created in step 3.
- `TaskInfoModel`: Bind the attribute to the `lc:TaskImage` object you created in step 3.
- `TaskForm`: Bind the attribute to the `lc:TaskForm` component you created in step 3.
- `Object`: Bind the attribute to the setter method that updates all the information in the presentation model component. For example, the setter method for the presentation model you created by following the steps in Creating a presentation model component for the content area was `currData`.

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

- `lc:SessionMap`: session
- `lc:TaskImageModel`: currTaskImageModel
- `lc:TaskInfoModel`: currTaskInfoModel
- `lc:TaskForm`: currTaskForm

```
<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*

```
<mx:Canvasxmlns: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 are displayed in the navigation area. The list of tasks can be all tasks that a user has been assigned (completed, active, and draft) or it can list only the active tasks assigned to the user.

When you create a presentation model component, you need to make the properties and methods that store and access data respectively, bindable. Bindable properties and methods permit 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, you need to 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 the name for the package. For example, type `custComp`.
3. In the **Name** box, type in a name for the class. For example, type `CustNavigatorModel`.
4. Beside the **Superclass** box, click **Browse** and select **PresentationModel - lc.presentationmodel**, 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`

   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 presentation model component are bindable to update in the user interface.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 using the `lc:QueuesManager` object by 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"));
   tasks = 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 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"));
            tasks = 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 the 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 task 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, you 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 to use the Workspace ES API components and custom components 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 required to use the Workspace ES 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 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. 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 typically would 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 displaying 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.
10. Select **File** > **Save** to save the view component for the navigation area.

Example: *The view component for the navigation area*

```
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 ES API -->
   <lc:EmbossedNavigator dataProvider="{myModel.tasks}"
                         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 default application MXML file (for example custLayout.mxml), in the default `<mx:Application>` tag, add the following attributes to include the Workspace ES 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 a Flex project to create a custom layout.)

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

3. After the opening `<lc:AuthenticatingApplication>` tag, add the component that you created for displaying 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.

4. Select **File** > **Save** to save the default application file.

5. Select **Project** > **Build** to compile your project.

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

Example: *Creating a custom layout to view tasks assigned to 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>
```

Tip: To include color or image changes, copy your modified workspace-theme.swf file to the html-template folder in your project before you compile it.

Deploying the custom layout to a web server

Although you can build your Flex application like any other Flex application, you will notice that it cannot run locally in Flex. After you compile your Flex application, you must deploy it to a web server that is connected to a LiveCycle ES server. The specific tasks you do to deploy your application depend on the web server you use and how you have that web server configured with LiveCycle ES. For example, if you are using a turnkey installation of the LiveCycle ES server, you can deploy to the available Tomcat web server.

When you deploy your Flex application, it is important that the following files are included:

- All the files in your output directory from your Flex application. In a typical Flex project configuration, this is the **bin** folder.
- (Optional) For a modified theme file, you must deploy your version of the workspace-theme.swf file.
- (Optional) To display the Flex application in different languages or use a localization file you created, you must copy the locale directory to the folder where you deployed your Flex application. The locale folder is available from the adobe-workspace-client.war file from within the adobe-workspace-client.ear file. For example, in a turnkey installation of the LiveCycle ES server, you would navigate to *[installdir]*\jboss\server\all\deploy folder, where *[installdir]* is the location of the turnkey installation.

Deploying a Flex application involves the following tasks:

1. On the web server connected to your installation of LiveCycle ES, navigate to the location to deploy your files. For example, in a 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 a turnkey installation of the LiveCycle ES server.

2. Create a folder for your Flex application. For example, you can create a new folder called *custLayout* in the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\ folder, where *[installdir]* is the location for a turnkey installation of the LiveCycle ES server.

3. From the Flex project where you built your Flex application, copy files that were created after a successful build to the folder you created in step 2. For example, copy the contents from the bin folder to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLayout\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

4. (Optional) Copy a modified theme file to the folder you created in step 2. For example, copy the workspace-theme.swf file from your computer to the *[installdir]*\jboss\server\all\deploy\ jbossweb-tomcat55.sar\ROOT.war\custLayout\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

5. (Optional) Copy the locale folder to the folder you created in step 2. You can copy any localization files that you created to the new locale folder. For example, if you created a localization file, copy it to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLayout\locale, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

Testing the custom layout

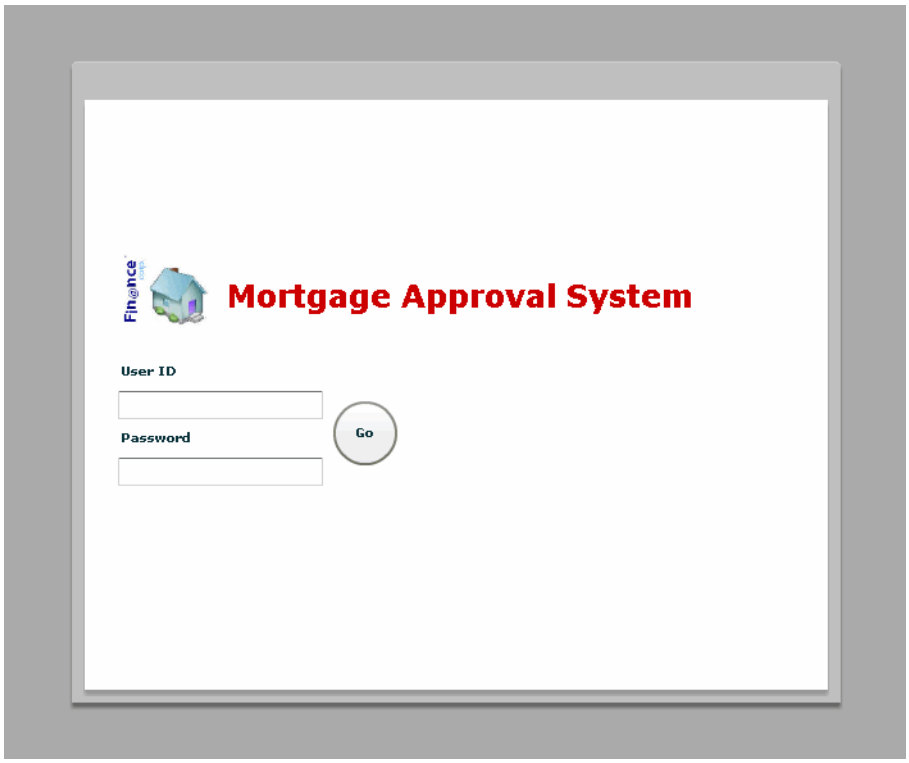
You can test your Flex application after you successfully deploy it to a web server that is connected to LiveCycle ES. You access your Flex application by using a URL that includes the name of the server you deployed your Flex application to and the name of the default application from your Flex project.

For example, for a turnkey installation of LiveCycle ES, you may have deployed your Flex application to a the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLayout\ folder, where the default application file is named *custLayout.mxml*. In a web browser, you would type `http://[servername]:8080.custLayout/custLayout.html`, where `[servername]` is the name of the server where LiveCycle ES is installed.

8 Replacing the Login Screen

LiveCycle Workspace ES is built using self-contained visual components and non-visual components collectively called the *LiveCycle Workspace ES API*, which provides process management functions as described in *LiveCycle ES ActionScript Reference*. You can use the `lc:AuthenticatingApplication` component, which handles displaying the login screen to authenticate the user, in order to use a custom login screen that you create. If you need to only modify the login screen, you can use the Workspace ES application as a component. 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 appears completely different from the default screen provided by Workspace ES.



Summary of steps

You must complete these high-level tasks to create a Flex application in order to reuse Workspace ES components to replace the login screen and then use the Workspace ES application as a component. The example that accompanies these tasks describes how to implement the necessary interface for a login screen, create the 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 Your Development Environment.)
2. Create a project. (Creating a Flex project to create a custom login screen.)
3. Create the application logic for a custom login screen and compile the Flex application. (See Creating the application logic for a custom login screen.)
4. Deploy your SWF file to the LiveCycle ES server. (See Deploying the custom screen.)
5. Test your new Flex application. (See Testing the custom login screen.)

Creating a Flex project to create a custom login screen

After you configure your environment to customize Workspace ES, you can use the Workspace ES API to create a Flex application in order to create a custom login screen.

You must configure your Flex application to reference the workspace-runtime.swc, workspace_rb.swc, and fds.swc files before you start creating your application logic by using Workspace ES components. You must also create a namespace in your default application file to access classes, components, and interfaces from the Workspace ES API.

➤ **To create a project:**

1. In Flex Builder, select **File** > **New** > **Flex Project**.
2. In the New Flex Project dialog box, select **Basic (for example, XML or web service from PHP/JSP/ASP.NET)** and click **Next**.
3. In the New Flex Project dialog box, in the **Project Name** box, type a name for the Flex project.
4. In the Project Contents area, perform one of these tasks, and then click **Next**:
   - To save your project to a default location, select **Use Default Location**.
   - Click **Browse** and select the folder to save your project to, and then click **OK**.
5. In the New Flex Project dialog box, click the **Library Path** tab and then click **Add SWC**.
6. In the Add SWC dialog box, click **Browse** and select the workspace-runtime.swc file that you copied to your computer, and then click **OK**.
7. Repeat steps 5 and 6 to add the workspace_rb.swc and fds.swc file to your Flex project.
8. When you return to the New Flex Project dialog box, click **Finish**.
9. In the Navigator view, locate your new Flex project, right-click **html-template** and select **Delete**.

10. Drag the html-template folder that you copied to your computer earlier from the LiveCycle_ES_SDK folder to your new Flex project. The html-template folder and its contents should be copied to your Flex project.

11. In the Navigator view, right-click your Flex project and select **New** > **Folder**.

12. In the New Folder dialog box, in the **Folder name** box, type a name and then click **Finish**. For example, type `custComp`. This folder will store all the components you will create.

13. Right-click the **custComp** folder and select **New** > **Folder**.

14. In the New Folder dialog box, in the **Folder name** box, type a name, and then click **Finish**. For example, type `custImages`. This folder will store custom images that will be used in your project.

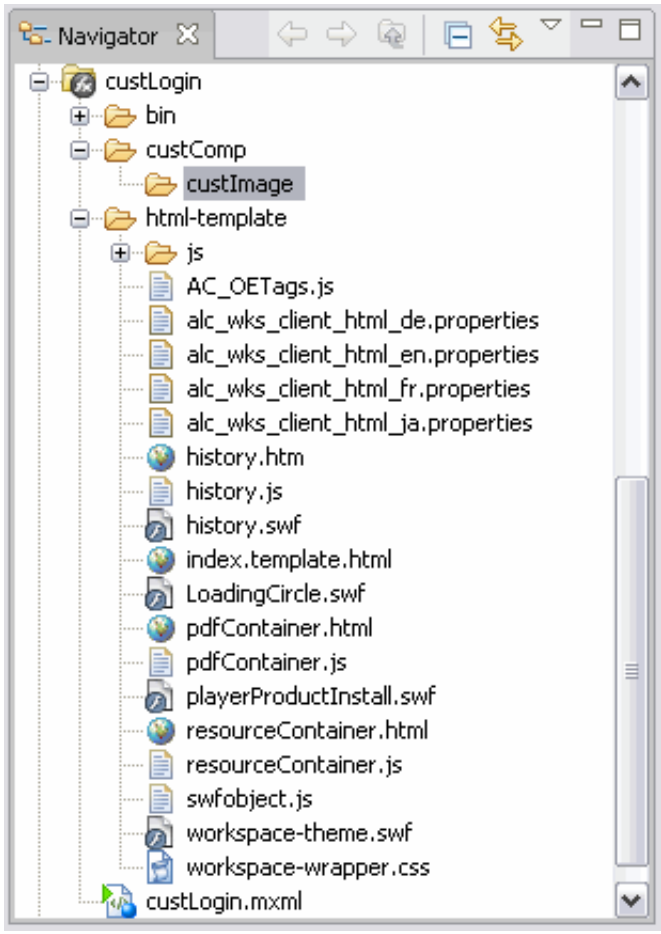
15. Copy a custom image into the assets folder. For example, you can copy the houseImage.jpg image file from the LiveCycle_ES_SDK folder from one of these locations:

    Workbench ES installation: Navigate to the Workspace folder in *[installdir]*\LiveCycle ES\ Workbench ES\ LiveCycle_ES_SDK\misc\Process_Management\, where *[installdir]* represents 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\, where *[installdir]* represents where LiveCycle ES is installed on a server.

    For example, for a turnkey installation, navigate to the Workspace folder in \Adobe\LiveCycle8\ LiveCycle_ES_SDK\misc\Process_Management\.

The contents of your Flex project and the html-template folder should look similar to the following illustration.



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. To use the Workspace ES API, you replace the `<mx:Application>` tag with the `lc:AuthenticatingApplication` container. The `lc:AuthenticatingApplication` component functions similar to the `mx:Application` container, but it displays the Workspace ES login screen as necessary and stores the session information required to use Workspace ES 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, you complete the following high-level tasks:

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

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 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 **Name** box, type in a name for the class. For example, type `CustLoginPageAdapter`.

3. 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**.

4. In the Interfaces area, click **Add**, select **ILoginPage - lc.core**, and then click **OK**.

5. Confirm that only **Generate functions inherited from interfaces** is selected and then click **Finish**.

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

After you create your ActionScript class in Flex Builder, complete these tasks to write the application logic code implementing 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`). 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 need to 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.

2. 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);
   }
   ```

3. For the `serverUrl` property, perform the following tasks:

   - 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 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 would look like the following code if you created a property named `myErrorMessage`.

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

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

4. For the `userid` property, perform the following tasks:
   - 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 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 would look like the following code if you created a property named `myUserid`:

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

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

5. For the `errorMessage` property, perform the following tasks:
   - 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 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 would look like the following code if you created 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*), you can 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 a separate logging file.

6. For the `password` property , perform the following tasks:
   - 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 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 would look like the following code if you created a property named `myPassword`:

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

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

7. For the `relogin` property, perform the following tasks:
   - 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 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 would look like the following code if you created a property named `myRelogin`:

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

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

8. Delete all other functions that have been added into your class; otherwise, you will get an error when you compile your project.

9. 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.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 an 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. By creating a separate class that implements the details of implementing the `ILogin` class, it 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 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 **LoginPageAdapter**, which is the ActionScript class you created earlier 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 tasks to write the application logic code for the new login screen:

1. In the CustLoginPage.mxml file, for the `cc:CustLoginPageAdapter` component, you must add `http://www.adobe.com/2006/livecycle` as the `lc` namespace to access the Workspace ES API components and any 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 `width` and `height` attributes to a value of `100%` for the `cc:LoginPageAdapter` 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"
                            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 appears 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 Replacing the Login Screen, you would perform the following tasks:

   - 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 set 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:CutLoginPageAdapter
   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">

   <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:LoginPageAdapter>
```

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 tasks:

1. In Flex Builder, in the default application MXML file (for example, custLogin.mxml), in the default `<mx:Application>` tag, add the following attributes to include the Workspace ES 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.

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

3. 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" >
```

4. 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.

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

6. 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***

```
<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" >

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

</lc:AuthenticatingApplication>
```

Tip: To include theme customizations to any of the components, copy your customized workspace-theme.swf file to the html-template folder in your project before you build your project.

Deploying the custom screen

Although you can build your Flex application like any other Flex application, you will notice that it does not run locally. After you compile your Flex application, you must deploy it to a web server that is connected to a LiveCycle ES server. The specific tasks you do to deploy your application depend on the web server that you use and how you have that web server configured with LiveCycle ES. For example, if

you are using a turnkey installation of the LiveCycle ES server, you can deploy to the available Tomcat web server.

When you deploy your Flex application, it is important that the following files are included:

- All the files in your output directory from your Flex application. In a typical Flex project configuration, this is the **bin** folder.
- (Optional) For a customized theme file, you must deploy your version of the workspace-theme.swf file.
- (Optional) To display the Flex application in different languages or to use a localization file you created, you must copy the locale directory to the folder where you deployed your Flex application. The locale folder is available from the adobe-workspace-client.war file from within the adobe-workspace-client.ear file. For example, in a turnkey installation of LiveCycle ES server, you would navigate to *[installdir]*\jboss\server\all\deploy folder where *[installdir]* is the location of the turnkey installation.

Deploying a Flex application involves the following tasks:

1. On the web server connected to your installation of LiveCycle ES, navigate to the location to deploy your files. For example, in a 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 a turnkey installation of the LiveCycle ES server.

2. Create a new folder for your Flex application. For example, you can create a new folder called *custLogin* in the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

3. From the Flex project where you built your Flex application, copy files that were created after a successful build to the folder that you created in step 2. For example, copy the contents from the bin folder to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLogin\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

4. (Optional) Copy a customized theme file to the folder you created in step 2.For example, copy the workspace-theme.swf file from your computer to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLogin\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

5. (Optional) Copy the locale folder to the folder you created in step 2. You can copy any localization files that you created to the new locale folder. For example, if you created a localization file, copy it to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLogin\locale, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

Testing the custom login screen

You can test your Flex application after you successfully deploy it to a web server that is connected to LiveCycle ES. You would access your Flex application by using a URL that includes the name of the server you deployed your Flex application to and the name of the default application from your Flex project.

For example, for a turnkey installation of LiveCycle ES, you may have deployed your Flex application to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\custLogin\ folder, where the default application file is named *custLogin.mxml*. In a web browser, you would type `http://[servername]:8080/custLogin/custLogin.html`, where `[servername]` is the name of the server where LiveCycle ES is installed.

9 Compiling LiveCycle Workspace ES

LiveCycle Workspace ES is a Flex application. The source code for Workspace ES is available if you have the LiveCycle ES Samples installed. You can copy the source code to your computer and compile Workspace ES for the purpose of understanding how it works or to test changes to color, images, or localizations under a different Workspace ES user interface instance. It is possible to have multiple instances of Workspace ES on the same LiveCycle ES server.

It is not recommended that you perform any customizations to the Workspace ES source code and deploy it to production. When you compile Workspace ES, it is recommended that you always use the latest version of the source code. The source code is updated in the LiveCycle ES SDK folder each time a patch is applied.

Note: Customizing Workspace ES source code and deploying it to production is not supported.

Summary of steps

You must complete these high-level tasks to compile the Workspace ES source code that is provided as part the LiveCycle ES Samples.

1. Configure your development environment for customizing Workspace ES. (See Configuring Your Development Environment.)
2. Import the Workspace ES source code and compile the Workspace ES application. (See Importing and compiling Workspace ES.)
3. Deploy the compiled application. (See Deploying the compiled Workspace ES application for testing.)
4. Test the compiled application. (See Testing the compiled Workspace ES application.)

Importing and compiling Workspace ES

After you configure your environment to customize Workspace ES, you can import the Workspace ES source code to compile and run Workspace ES to better understand the source code.

You must configure your Flex application to use the fds.swc and workspace_rb.swc files before you can compile Workspace ES. In addition, you must remove an invalid reference to the existing fds.swc file if you do not have LiveCycle Data Services ES installed.

Perform one of the following procedures based on the version of Flex Builder you are using to compile the Workspace ES source code.

➤ **To import and compile the Workspace ES source code in Flex Builder 2:**

1. In Flex Builder, select **File** > **Import Project**.
2. In the Import dialog box, select **Existing Projects into Workspace** and click **Next**.
3. Verify that **Select root directory** is selected and click **Browse**.

4. In the Browse For Folder dialog box, navigate to and select the folder from which you extracted the source code, and then click **OK**. For example, select the **wsSource** folder.

5. Click **Finish**.

6. In the Navigator view, the Workspace project appears. Copy the services-config.xml file to the Workspace project. The services-config.xml file is one of the files you copied to your computer. (See Retrieving the files for customizing Workspace ES.)

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

8. In the Properties for Workspace dialog box, select **Flex Build Path** and click the **Library Path** tab.

9. In the Build patch libraries area, select **$(FRAMEWORKS)\libs\fds.swc** and click **Remove**.

10. Click **Add SWC** and, in the Add SWC dialog box, click **Browse**.

11. In the Choose a SWC File dialog box, go to the folder where you copied the SWC files for customizing Workspace ES, select **fds.swc**, and then click **Open**. For example, select fds.swc in the \forWSCustomization\SWCS folder.

12. When you return to the Add SWC dialog box, click **OK**.

13. Click **Add SWC** and click **Browse**.

14. In the Choose a SWC File dialog box, go to the folder where you copied the SWC files for customizing Workspace ES, select **workspace_rb.swc**, and then click **Open**. For example, select workspace_rb.swc in the \forWSCustomization\SWCS folder.

15. When you return to the Add SWC dialog box, click **OK**.

16. When you return to the Properties for Workspace dialog box, in the left pane, select **Flex Compiler**.

17. In the Additional compiler options, type `-services ../services-config.xml` to the beginning of the existing compiler options. Your compiler options should look like the following text:

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

18. (Optional) If you want to compile to another locale other than US English, you can type the locale in place of the default en_US value for the `-locale` parameter in the compiler options and then click **Apply**.

    Note: You must also add the localization file you create to the html-template\locale\ folder in your Flex project (see Localizing LiveCycle Workspace ES) and use the 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.

    For example, to compile Workspace ES to international Spanish, you type the value of `es_ES` for the `-locale` parameter as shown in this example:

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

19. When you return to the Properties for Workspace dialog box, click **OK**.

20. Select **Project** > **Build Project** to compile Workspace ES.

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

➤ **To import and compile the Workspace ES source code using Flex Builder 3:**

1. In Flex Builder, select **File** > **Import** > **Flex Project**.
2. In the Import Flex Project dialog box, click the **Browse** button beside the **Archive file** box.
3. In the Open dialog box, navigate to and select the adobe-workspace-src.zip file that you copied to you your computer (see Retrieving the Workspace ES source code), click **Open**, and then click **Finish**.
4. In the Choose Flex SDK Version dialog box, select **Use a specific SDK**, select **Flex 2.0.1 Hotfix 3** in the list beside it, and click **OK**. In the Flex Navigator view, the Workspace project appears.
5. Copy the services-config.xml file to the Workspace project. The services-config.xml file is one of the files you copied to your computer. (See Retrieving the files for customizing Workspace ES.)
6. In the Flex Navigator view, right-click **Workspace** and select **Properties**.
7. In the Properties for Workspace dialog box, select **Flex Build Path** and click the **Library Path** tab.
8. Click **Add SWC**.
9. In the Add SWC dialog box, click **Browse**.
10. In the Choose a SWC file dialog box, navigate to the folder where you copied the SWC files for customizing Workspace ES, select **workspace_rb.swc**, and then click **Open**. For example, select workspace_rb.swc in the \forWSCustomization\SWCS folder.
11. When you return to the Add SWC dialog box, click **OK**.
12. When you return to the Properties for Workspace dialog box, select **Flex Compiler** in the left pane.
13. In the Additional compiler options, type `-services ../services-config.xml` to the beginning of the existing compiler options and then click **Apply**. Your compiler options should look like the following text:

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

14. (Optional) To compile to another locale other than US English, type the locale in place of the default en_US value for the `-locale` parameter in the compiler options, and then click **Apply**.

    Note: You must also add the localization file you create to the html-template\locale\ folder in your Flex project (see Localizing LiveCycle Workspace ES) and use the 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.

    For example, to compile Workspace ES to international Spanish, type the value of `es_ES` for the `-locale` parameter as shown in this example:

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

15. When you return to the Properties for Workspace dialog box, click **OK**.
16. Select **Project** > **Build Project** to compile Workspace ES.

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

Configuring Flex Builder for debugging

You can modify the Flex project properties to step through the code in the Flex Debugging perspective available with Flex Builder. The Flex Debugging perspective is useful for placing breakpoints to stop and step through the source code.

➤ **To modify the Flex project properties:**

1. In the **Output folder** box, click **Browse**, navigate to the folder you created to store the compiled Workspace ES application that was created in Deploying the compiled Workspace ES application for testing, and then click **OK**. For example, if you have a turnkey installation on your computer, type `[installdir]\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\testworkspace\`, where `[installdir]` is the location for the turnkey installation of LiveCycle ES server.

2. In the **Output folder URL** box, type the URL (but not the Main.html file name extension), specifying where you will access the compiled Workspace ES application as described in Testing the compiled Workspace ES application. For example, type `http://[servername]:8080/testworkspace`, where `[servername]` is the name of the server where LiveCycle ES is installed.

Deploying the compiled Workspace ES application for testing

After you compile Workspace ES, you must deploy it for testing purposes. It is important that you include all the files in your output directory from your compiled Workspace ES when you deploy it. In a typical Flex project configuration, the output directory is the **bin** folder.

Deploying a compiled Workspace ES application involves the following tasks:

1. On the web server that is connected to your installation of LiveCycle ES, navigate to the location to deploy the compiled Workspace ES. For example, in a 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 a turnkey installation of the LiveCycle ES server.

2. Create a new folder for your test version of Workspace ES. Do not use the name workspace because the installed version of Workspace ES is already using it.

   For example, you can create a new folder called *testworkspace* in the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

3. From the Flex project where you built the Workspace ES application, copy files that were created after a successful build to the folder you created in step 2. For example, you can copy the contents from the bin folder to the *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\testworkspace\ folder, where *[installdir]* is the location for a turnkey installation of LiveCycle ES server.

Tip: If you want to step through the code while Flex Builder is in debug mode, configure the Output folder box to reference the folder you created in step 2.

Testing the compiled Workspace ES application

You can test your compiled version of Workspace ES by logging in using a web browser.

For example, for a turnkey installation of LiveCycle ES, you may have deployed your compiled version of Workspace ES to *[installdir]*\jboss\server\all\deploy\jbossweb-tomcat55.sar\ROOT.war\testWorkspace\ folder. To access the compiled Workspace ES application, type `http://[servername]:8080/testworkspace/Main.html`, where *`[servername]`* is the name of the server where LiveCycle ES is installed.

10 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
The image you added to a customized theme file does not appear within Workspace ES.	• Copy the image to the image folder when you compile your theme file. • 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 portion of the name you used for the localization file is the same as the locale code you specified in the `-locale` option when you compiled it.
The localization file customization does not display the correct language.	• Verify that the localization SWF file is copied and packaged in the adobe-workspace-runtime.ear file. • Verify that your web browser settings are set to display the language that you are displaying first. • Verify that the locale extension for your localization file matches the locale you are specifying in the web browser. For example, if your locale is *es*, your localization file should be named workspace_rb_es.swf. • 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. Verify that you have included the Workspace ES default images if you customized the theme file. • 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 source code after you copy it locally to your computer. (See Retrieving the Workspace ES source code.)
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 components do not update with information	• You may not have passed the `lc:SessionMap` object. Each Workspace ES API object requires setting the `session` attribute to a valid `lc:SessionMap` object.

Issue	Resolutions
Display of forms and functions do not seem to work properly in the web browser.	• You may not have copied the contents of the html-template directory provided with the LiveCycle ES SDK when you compiled your project.
After upgrading, the images that were replaced and the colors and localization changes are missing.	• During an upgrade, the adobe-workspace-client.ear file has been replaced. It is recommended that you deploy your customizations into a separate EAR file. You will need to redeploy your theme file and any custom localization files.