

Best practices for team-based development with Adobe® LiveCycle® Workbench ES

Table of contents

- 1 Intended audience
- 1 The development process
- 2 Sharing assets in LiveCycle Workbench ES
- 4 Setting up folder permissions
- 4 Setting up roles for development
- 4 Developing forms and documents
- 6 Incorporating Rich Internet Applications in LiveCycle ES
- 7 Adding service components and maximizing reuse
- 8 Designing processes for reuse
- 10 Summary

When developing Adobe LiveCycle ES (Enterprise Suite) applications, proper planning can reduce the development effort, ease the application maintenance burden, and improve performance. This paper provides a best practices guide to help project teams set up and use Adobe LiveCycle Workbench ES to develop LiveCycle ES applications. You can use the best practices to help organize LiveCycle ES projects and to develop internal processes.

Intended audience

LiveCycle Workbench ES is an integrated development environment (IDE) that enables process developers, Flex developers and form authors to create automated processes and forms, and manage and organize the resources and services created during application development. This paper provides LiveCycle ES these users as well as LiveCycle administrators with the information they need to cooperatively develop automated processes, Rich Internet Applications (RIAs), and form or document applications using LiveCycle Workbench ES.

To use this documentation effectively, you should have a running server installation of LiveCycle ES or LiveCycle ES Update 1. Developers should also have the corresponding version of LiveCycle Workbench ES installed on one or more desktop computers. LiveCycle Workbench ES must be connected to a LiveCycle ES server in order to develop and store application assets.

Note: The advice and best practices in this paper are for LiveCycle ES and LiveCycle ES Update 1 (8.2). If a practice relates only to LiveCycle Update 1, it will be noted.

The development process

The development process for a LiveCycle ES application should involve at least three server environments. A development server is used to create applications and make changes to an existing application. Once the application has been thoroughly tested, it is moved to a staging environment that mirrors the production system and the application is tested again. When testing is completed, the application is then moved into a production environment. The development environment may be operated on more available and less expensive hardware, databases, and applications servers. The test environment should match as closely as possible the eventual production environment in order to provide a safe sandbox within which verification can be performed without impacting production.

Some testing practices use four or five different environments: for example, user acceptance testing, performance/load testing, and system/integration testing. For some of these purposes, virtualization technology is helpful (for example, multiple isolated environments sharing hardware); however, for other types of testing, like performance testing, it may not be applicable.

Tip

In LiveCycle Update 1, LCAs can be based on a process or on a set of resources (prior to Update 1, LCA had to be based on a process). For both types of LCAs (process and form based), internal relationships are recursively traversed in order to deterministically locate all the related content (whether it is other processes, form fragments, images, or other types of content) to retrieve everything that needed by the application.

Applications are moved from one environment to another via a LiveCycle ES archive (LCA) file. An LCA is a compressed file (similar to a Java ARchive or JAR file) that contains application assets such as the process diagram/definition, forms, image files, fragments, custom components, services, and associated resources that are required to use the process in another environment. The existing core and document services installed with LiveCycle ES (such as the User service and the Forms service) are not included in the archive because these services are expected to be present in the destination environment.

Sharing assets in LiveCycle Workbench ES

A powerful feature of LiveCycle ES is the ability for project teams to collaboratively develop the various assets that comprise a LiveCycle ES application. Assets are stored and managed on the server so they are available for use by all connected users. When new content is created or existing content needs to be modified, it can be retrieved from the server and placed in a temporary location on a user's file system.

Locking assets

LiveCycle ES assets (processes and resources) can be locked to inform users that a file is being modified by another user. The lock is a "shared lock," which does not prevent other users from modifying the asset; a lock icon indicates that the asset is in use and should not be modified.

Assets are automatically locked when they are opened for editing. When a file is locked, a lock icon appears next to the asset. You can also lock individual assets or folders. If you select Lock on a folder, its entire contents will be locked.

Locking an asset does not prevent other users from locking the same asset; if this happens, the file of the user that saves the asset first will become the most recent version. The lock icon notifies users that the file in use and overwriting of files is avoided. In addition, shared locks allow a team member to work on an asset if another user has forgotten to unlock it—without the intervention of an administrator.

Creating a repository structure

Using LiveCycle Workbench ES, you can access the LiveCycle ES repository with the Resource view to create folders and move assets within the folders (for example, fragments, forms, images, custom form guide SWCs, and preview XML data files). To add assets to the repository, drag the assets from your computer into the destination folder in the Resource view. To create a folder, right-click an existing folder and choose New Folder.

An easy way to organize your project is to create folders in the repository to represent categories. For example, you might have a top-level folder named "Account Opening" that contains account categories and forms. Within that folder, you could create a folder named "Account Management" which could contain forms such as Application.xdp, Contributions.xdp, and Cancellation.xdp.

The next step is to consider how fragments and various other assets relate to one another. For example, some assets will be used companywide; others may be used only within a particular department or on a specific form.

Therefore, it is best practice for each form to have its own folder to contain the assets that are specific to it (each form has two assets: the image and the form fragment that includes the image), and each category folder should have special folders to contain the assets that pertain to some or all forms within that category. Adobe also recommends that base templates (XDP files with basic layouts/functionality used to create new forms) be kept separate from these folders.

As shown in figure 1, there are two categories in the repository structure (Category A and Category B).

Category A has three folders: Form 1, Form 2, and Common:

- Common folder—Contains the assets that are common to all forms within that category
 - Frag folder—Contains fragments that are common to both forms within Category A
 - Data folder—Contains a common schema and sample XML data file that some or all forms within Category A should use

Tip

To create a top-level folder, right-click the LiveCycle ES server icon (represented by a tree) in the Resource view.

- Form 1 folder—Contains Form1.xdp (the actual form) and two folders: Frag and Res
- Frag folder—Contains all the fragments specific to Form1.xdp. (In some cases, a form object is used in multiple instances within a single form, so it is recommended that you make it a form fragment so it will only need to be updated once if the contents change.)
- Res folder—Contains all the resources (assets that aren't fragments) that are specific to Form1.xdp (an image and a form guide SWC file containing customized styles)
- Form 2 folder—Contains only one form, Form2.xdp. It doesn't contain any fragments or resources, but may use some fragments located in the category or repository common folders (such as the CompanyLogo.jpg resource)

Category B has one folder (Form 1) which contains one form (Form1.xdp). The repository has a common folder that contains fragments (Frag folder) and resources (Res folder) that are common to all forms within all categories of the repository. Note that category, common, and form folders can contain frag, res, and data folders, if necessary.

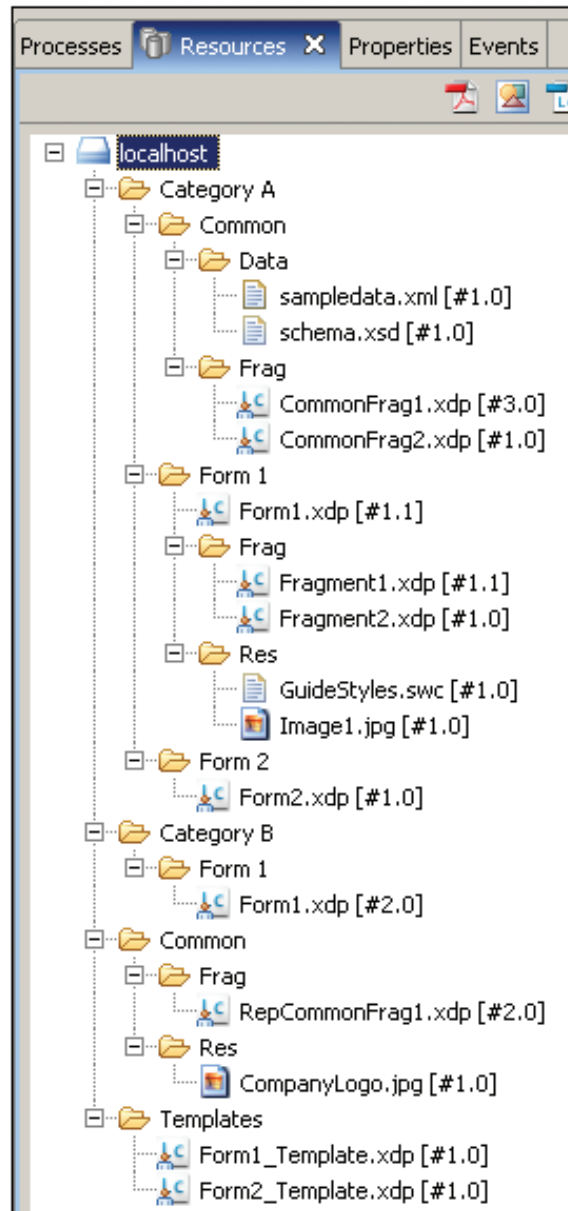


Figure 1. Repository structure

Tip

Make sure you set permissions to allow users to access a particular form *and* the fragments that it references. For instance, if you implement the recommended repository folder structure, make sure all users have at least read access to the fragments within the repository's common folder. You could assign write access for that folder only to users who have permission to change corporate content such as the company logo or common functions in a script fragment.

Setting up folder permissions

Folder permissions for resources can be set by a resource administrator, application administrator, or LiveCycle ES administrator. When rigid separation of form assets is required, it is best practice not to grant the resource administrator role to form developers or designers; instead folder-level permissions should be assigned to assets. The Resource view in LiveCycle Workbench ES supports setting these permissions (by resource, application, and LiveCycle ES administrators) which will allow you to control who has access to the contents of specific folders. Note that these are user permissions, not content permissions. For example, you cannot set up permissions to control which fragments a form can reference. Permissions are used to control the content a user may access (read and/or write) while designing fragments and forms. An administrator can also assign “delegate rights” to a team member, enabling the project team to function without a resource administrator.

If you have multiple departments that use the same forms, fragments, or resources, make sure that the permissions are set for those folders across departments to enable access for those users. You should also make sure that you set the appropriate permissions to prevent users who should not have access from opening those folders.

Setting up roles for development

To allow developers to make changes to a LiveCycle ES application, roles are assigned by a LiveCycle ES administrator. Roles can be assigned to individual users or a set of roles can be assigned to a user group. It is best practice to manage roles based on group membership, where users can be added or removed from the group, as necessary. LiveCycle ES provides several built-in roles for accomplishing common tasks.

For the majority of development environments, individual LiveCycle Workbench ES users should first be added to a specific user group and then the group should be associated with the application administrator role assigned to it.

In more formal environments, it may be desirable to distinguish forms developers from process developers. It is best practice to create a group for each using the resource administrator and process administrator roles.

In environments where rigid separation of forms collateral is required, it is best practice to manage content access using the access permissions (ACLs) associated with the repository folders. For more information, see the LiveCycle Workbench ES documentation.

The following table describes the recommended roles for application developers.

Role	Description
Resource administrator	Creates and changes permissions on resource folders. Assigns the necessary permissions to connect to a LiveCycle ES server from LiveCycle Workbench ES and to create, modify, and store forms and their associated collateral.
Process administrator	Assigns the necessary permissions to connect to a LiveCycle ES server and to create, modify, and store processes; and permissions to install, activate, start, and stop services and processes.
Application administrator	Superset of resource administrator and process administrator permissions. Application administrators can also import LCAs.

Developing forms and documents

About fragments

Fragments are reusable parts of forms or document templates. They can be shared between form authors to speed development and make maintenance easier. There are various ways that you can take existing content and make them into fragments. Adobe LiveCycle Designer ES software supports two kinds of fragments (from the form perspective in LiveCycle Workbench ES): form fragments and script object fragments. The difference is form fragments are reusable pieces of forms or documents contained in a subform, whereas script object fragments are reusable scripts (more specifically, reusable libraries of JavaScript functions).

We will use the term fragment to speak generally of both form and script object fragments.

Creating fragments

The first step in converting content into fragments is to determine whether the content should be a fragment or not. The best candidates are content pieces that you will need to use in multiple forms but must be identical in all the forms that use it. A good example is a company logo—it needs to be on every form your company produces, but you do not want to update it in each file if there are changes.

Another candidate for converting into a fragment is content that you create as you develop your forms. For example, you may have JavaScript functions that perform various actions on form fields. Perhaps there are functions that validate certain types of values like a serial number with a structure unique to your company. It is recommended that you put the serial number validation function in a script object fragment that is used by all of your company's forms should the serial number's structure ever change.

As you plan how to convert existing content into form fragments, think about how you will bring them together in various forms. You will probably have some common JavaScript functions which would imply at least one script object fragment that would be used by all forms. It is recommended that you start by creating this script object fragment so that you can easily include it in all the forms you create. In fact, you could modify your company's LiveCycle Designer ES templates so that every template references this company script object fragment—making it easy for users to create a new form that is properly setup.

When to use nested fragments

In LiveCycle Designer ES, fragments are essentially containers of form objects or scripts. Since a fragment is also a form object, you can have a fragment that references another fragment. This is called “nesting.”

Nested fragments can be useful, especially in cases where you have a set of fragments that should always be used together. Perhaps you have many fragments that represent legal clauses, so you could have a legal clauses fragment that nests all of the existing legal clause fragments. Then, when you need to include the legal clauses in a form, you will only need to use the one fragment and all the nested fragments will update automatically. LiveCycle Designer ES can create subform sets which are typically used with nested fragments. Furthermore, if a new legal clause is created, you can easily add it as a fragment to the legal clauses fragment so that all existing forms would automatically update with the new legal clause—without manually adding the new clause to every form.

Fragment dependency checking

Before you start creating your fragments, it is recommended that you plan the folder structure for storing them in the LiveCycle ES repository. For more information, see the “Creating a repository structure” section in this document. Keep in mind that a fragment is essentially a reference from one form to an object in another form, whether it's a form object (a subform) or a script object. References are created via a relative path from the form that references the fragment to the fragment itself. When you move fragments and forms within the repository, it does not automatically update the fragment reference paths.

To avoid breaking links to forms that reference a fragment, you can use a new LiveCycle Workbench ES Update 1 feature that identifies which forms/fragments depend on a particular asset. (Note that it does not identify which assets a form or a fragment depends on, only the opposite.) To use this feature, select a fragment in the repository and select Context > Relationships. An example is shown in figure 2.

Fragments inside and outside of the repository

LiveCycle ES lets you work with fragments in various locations: your local file system or on a shared network drive (for example, via WebDAV) using the LiveCycle Designer ES fragment library palette or the LiveCycle Workbench ES repository.

Tip

You should be wary of utilizing circular references using nested form fragments—that is when a nested form fragment references another form fragment earlier in the chain. A circular reference acts like a recursive function—calling itself forever because it's missing a stop condition.

Tip

The relationships feature in the repository only shows one level deep. If a fragment is used in a form, you will see the form as being related. However, if this fragment is nested within another fragment that is used in a form, you will only see the nesting fragment, not the form. You can use the relationships feature on the nesting fragment to visualize the next level.

LiveCycle Designer ES is installed as part of LiveCycle Workbench ES, but can also run as a standalone application. You may want to run the standalone LiveCycle Designer if you prefer to operate in a language other than English. LiveCycle Designer is available in English, French, German and Japanese. When it is run as a standalone application, LiveCycle Designer ES stores and accesses fragments on the file system. When it is run as part of LiveCycle Workbench ES, fragments are created and stored in the LiveCycle ES repository.

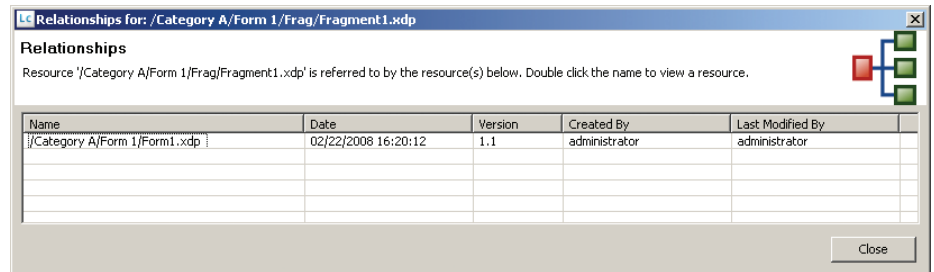


Figure 2. Fragment relationships

You can publish your form designs to a shared or web folder so other users or applications can access them. You publish form designs using commands in LiveCycle Designer ES File menu. If a form design contains links to external files, the links are modified to reflect the new location of the file. You will need write access to the folders you want to publish. Your network administrator can set up the necessary permissions.

Tip

You can easily add forms and fragments created outside of LiveCycle Workbench ES and the repository, if all the forms refer to fragments using relative paths (the default when you use LiveCycle Designer ES to add fragments to your forms). Just drag single forms or entire folders (even those that contain subfolders) from your file system directly into the repository.

If you are using LiveCycle Designer ES outside of LiveCycle Workbench ES, and want to use resources such as images and fragments that reside in the repository, you can set up a web folder on your file system and map it to the location in the repository. When you publish the form, links to the resources are maintained. You use the Publish to Repository command in LiveCycle Designer to copy the files into the repository.

Alternatively, you can create the form design on your file system. When the form design is complete, you can drag the folder and any subfolders into the appropriate folder in the LiveCycle Workbench ES Resources view. The folder structure in the Resources view must reflect the structure on the file system.

You will need access to LiveCycle ES to read and write files in the repository. Your network administrator can set up the necessary permissions.

Incorporating Rich Internet Applications in LiveCycle ES

LiveCycle ES applications can incorporate RIAs using Flex to enhance the user experience and speed the data capture process. These RIAs can be developed from scratch using Adobe Flex™ Builder™ software or using LiveCycle ES components that can be customized and enhanced. Form Guides are an Adobe Flex-based wizard-style data capture interface created from an XDP form template. Form guides are built with the guide builder tool in LiveCycle Designer ES. Form guide layouts, panels, and components can be customized using Flex Builder. Adobe LiveCycle Workspace ES software is a Flex-based user interface for interacting with LiveCycle ES processes. Flex RIAs can be used for data capture inside LiveCycle Workspace ES.

Tip

You only need to install Flex Builder on developer desktops that will be creating RIAs or customizing form guides. Other LiveCycle ES application developers can use the compiled components without having Flex Builder installed on their desktops.

Using Flex Builder with LiveCycle Workbench ES

The Flex Builder development environment can be run with LiveCycle Workbench ES, as they both share the Eclipse Java Development IDE as a foundation. If you will be running Flex Builder and LiveCycle Workbench ES in the same environment, it is recommended that you install Eclipse version 3.2.2 first. Then install both LiveCycle Workbench ES and Flex Builder plug-ins into Eclipse. For more information on installing Eclipse plug-ins, see the Flex Builder and LiveCycle Workbench ES documentation.

When the Flex Builder and LiveCycle Workbench ES plug-ins are installed within Eclipse, the Flex Builder Navigator view and the LiveCycle Workbench ES Resources view appear in the same Eclipse window. Flex Builder projects and their assets continue to be stored on the local file system; LiveCycle Workbench ES assets continue to be stored in the LiveCycle ES repository.

Customizing form guides

Flex projects can be compiled in an SWC. An SWC file is an archive file for Flex components and other assets. SWC files contain a SWF file and a catalog.xml file. Form guides can be customized with new or modified layouts, panels, and components (objects). For more information on creating custom components for guides, visit http://help.adobe.com/en_US/livecycle/es/fgcustomize.pdf. When a Flex project has been compiled to an SWC, it can be moved into the LiveCycle ES repository for others to use. Using the Guide Builder tool (in LiveCycle Workbench ES), a user can browse for an SWC file from the LiveCycle ES repository like any other LiveCycle ES resource.

Creating Flex applications for LiveCycle Workspace ES

You can create a Flex RIA to provide data capture capabilities and enable it to work inside of LiveCycle Workspace ES. You can also enable existing Flex RIAs to work with LiveCycle Workspace ES by incorporating the appropriate event listeners. To view these approaches, visit http://help.adobe.com/en_US/livecycle/es/createflexapps.pdf.

After you develop your Flex RIA, save your Flex project and compile the Flex RIA as a SWF file. Drag the SWF file into the LiveCycle Workbench ES Resource view. You can then reference the Flex RIA from your LiveCycle ES process and set up a LiveCycle Workspace ES endpoint. You will reference the RIA using a form variable, which is created using the form data type. The form variable represents the data that is passed from the form. You can access the data when you design your process using XPath Builder.

Adding service components and maximizing reuse

The LiveCycle ES Foundation contains a service component framework that provides an easy way to implement core services, such as e-mail, FTP, and file utilities; and document services such as forms, Adobe LiveCycle Reader® Extensions ES and Adobe LiveCycle Output ES software. However, use LiveCycle Workbench ES to easily install custom service components that can extend the functionality of processes you will create in the Process Designer.

A custom service component can use any Java™ class to access files, databases, e-mail systems, and messaging queues. It can use external libraries to integrate with third-party applications and systems. These custom service components are simple Java objects consisting of an interface, implementation, and an XML file describing the service component. Then they are compiled and placed in a JAR file that is installed into LiveCycle ES via the LiveCycle Workbench ES.

Once a custom service component is installed, it is available from the Services view (under the category you selected when you installed and activated it) and can be dragged to any process. For a step-by-step guide to creating custom service components, visit www.adobe.com/devnet/livecycle/articles/dsc_development.html.

There is a Script component available that enables a process author to write custom Java code directly into the process. If your task requires more than a few lines of Java code, it is recommended that you create a custom service component.

The custom component is more efficient (the Java code in the Script component must be interpreted at runtime), easier to debug (the debugging environment can be attached to the LiveCycle ES application server) and maximizes reuse. Otherwise, the script would have to be copied from process to process; the custom component can be dragged onto any process.

A service component has several properties or metadata that are useful to process developers. They are configured in the component.xml file during development and are visible in the Properties view in LiveCycle Workbench ES.

Tip

To automatically create a form variable, drag the SWF file from the Resource view to the Variables view.

Tip

Make sure you use a good package naming convention (for example, com.mycompany.mycategory.csc.ServiceName).

- Service name—Make sure you provide a name for the component that is human readable; it will appear in the LiveCycle Workbench ES list of Services for process developers to select from (for example, ConvertCSVToXML).
- Description—This should succinctly describe the task of the component.
- Image/icon—You should create a unique icon for your component that identifies its function. This makes it easier to see on the process map.
- Category—Make sure that the category you provide for the component is consistent with its use. (For example, if it is a common component to manipulate paths and filenames, place it in the existing common category). Alternatively, you may want to place all your own custom components in categories of your own. Although categories cannot be nested, you can prefix the category with your company or department name (for example, MyCompany_Common).

A service component can have a number of configuration parameters. They are not the same as service method parameters. You can set configuration parameters globally from the Components view in LiveCycle Workbench ES (for example, the JDBC service component will have a global DataSource value). Configuration parameters have names, types, default values, and hints that are descriptions for the end user.

Here are some guidelines for creating configuration parameters. Note that types are simple and complex Java data types.

- Name—Should be meaningful and describe the parameter. They can be human readable and include spaces (for example, Transport Protocol).
- Hints—A concise description of the parameter. It appears as a tool-tip when the cursor hovers over the parameter in LiveCycle Workbench ES.

A component will also have one or more service operations/methods. These operations have names, hints, and input and output parameters. You drag the service operation onto the process map; the process developer will provide parameter values for the service operation.

- Operation name—Naming conventions should follow Java methods (for example, readValueFromFile).
- Operation hint—A human readable value that appears in the Properties view of the component. It should be a brief description of the function of this service.
- Parameter name—Naming conventions should follow Java method parameters (for example, fileName).
- Parameter title—Appears in the Property view as the parameter label. It should describe what the end user is required to capture against this parameter.
- Parameter hint—A human readable value that appears in the Properties view of the component. It should be a brief description of what the parameter is used for.

Designing processes for reuse

Before you start to use the process design perspective in LiveCycle Workbench ES to lay out your process, take the time to properly analyze the requirements. LiveCycle ES and LiveCycle Workbench ES provide a variety of features help ensure that functionality is implemented correctly (such as invoking subprocesses, short/long lived, gateways, events, and exception handling). Process reuse in Service Oriented Architectures is also know as “Process Serving”.

When designing processes for reuse, your planning should include:

- Defining the process granularity
- Designing the application program interface (API)
- Ensuring the process is discoverable

- Making the operation of the process clear
- Documenting the requirements
- Considering endpoints and security
- Deciding whether there is specific, custom functionality that requires a custom service component

It is important to decide on the granularity of the process you want to reuse. A process that is too small may not have sufficient functionality to make it worthwhile to reuse. Conversely, a process containing many activities may be too complicated or imply too many requirements to reuse. As a rule-of-thumb, processes that are intended for reuse should contain between five and ten activities. (Although there are reasonable processes containing a single activity where the purpose of the reuse is to abstract the parameter mappings). In general, a proliferation of small processes will clutter your design environment for no actual benefit and may result in performance degradation in high-volume production environments. A large process that accepts a myriad of input parameters to discriminate behavior for specific use cases will result in a cumbersome process interface that is difficult to maintain with evolving requirements. Ideally, you want the process to be in between the two extremes—not too small and not too big.

There are no absolutes in selecting the process granularity. As you become more experienced, you will become more adept at evaluating the trade-offs for your target users. You should notice commonly repeated patterns that span multiple processes—abstracting the pattern into a reusable subprocess will reduce complexity where the pattern is used and isolate the patterned behavior where it will be more easily maintained. The practice of abstracting the common pattern to a subprocess only after it has been observed to occur in more than one location will help you decide when it is appropriate to create a subprocess.

Note that although processes should be kept small so they will not be too difficult to maintain, it is possible to reuse a process by adding input variables that drive decision making. An example is a process used to render forms for both online and offline use exposed as a web service. By using an input variable to indicate offline or online use, the client calls only one web service regardless of whether it renders HTML or PDF forms.

Once the decision to isolate a portion of processing into a subprocess has been made, you will need to design the interface to the subprocess. The process interface is comprised of its name, its invocation policy (short lived or long lived) and the set of input and output variables that it exposes (its public variables). Together this information is known as the process signature. Because process consumers will embed direct references to these values, any change to the process signature will require visiting all consumers to update their references. This can be onerous and may not even be possible in all cases (such as if you are unsure where all the references are). In addition to the public signature, the internal workings of the process may necessitate the existence of additional temporary variables. These variables can be considered as private variables whose characteristics may be freely changed in the future without impacting the public signature. Because of the additional overhead associated with a signature change, it is worth taking extra care to get the process signature “as close to correct” as possible early in development.

Tip

Categories cannot be nested—but you can associate categories together by using specific naming conventions (for example, HR_Common, HR_Pension, HR_Leave, and so on).

You should make sure that your processes are discoverable by potential users. The name of the process should follow a meaningful naming pattern that employs a consistent capitalization policy. Using “verb first, camel caps” is a good practice; for example, naming a process “RenderFormsForPortal” is much better than “Myforms.” The process should also include a short description (which will be displayed to consumers in the Property view when they use it and is also used to provide filtering support during search). A reusable process should also specify an image so that it is readily distinguishable from other subprocesses at its point of consumption. Related processes should be placed into the same category or common processes into a distinct category.

You should make the operational details of the process clear. Ideally, a potential user will be able to intuit the behavior of your process based solely on its signature with some additional information, as provided by the process description. However, in most cases consumers will verify their understanding by examining the process definition. There are a variety of best practices that you can combine to make the behavior of the process clear.

- **Annotations**—You can use the text annotations (in LiveCycle ES Update 1) to embed descriptive text directly into the process definition. These annotations will not affect the runtime behavior but can greatly improve the expressiveness of your diagram.
- **Route names**—By default, routes are given internal names to distinguish one from another. LiveCycle Workbench ES suppresses the display of these default route names (such as Route0) to avoid unnecessary diagram clutter. You can rename these routes to describe the conditions under which a particular route will be followed.
- **Swimlanes**—Traditionally a swimlane is used to distinguish the tasks performed by one actor involved in sequence of processing from another. If the process contains multiple users with processing responsibilities passing back and forth between them, the traditional use of swimlanes is ideal. However, recall that swimlanes have no runtime semantic—they are strictly intended to be used as a communication mechanism. As such, a swimlane can be effectively used anywhere that it adds clarity to process behavior. As an example, you can use swimlanes to distinguish different phases of processing within a short-lived process involving no human interactions.
- **Color**—You can use the background color for swimlanes to call attention to portions of the process diagram. For example, using a red background where error processing occurs or a white background to specify process prerequisites will result in a more easily approachable environment.
- **Process requirements**—A process that is intended for reuse will invariably be invoked in more than one context. If the process makes assumptions about the calling context or the environment in which it expects to execute these prerequisites, it should be clearly expressed in the process diagram. Some examples of prerequisites include: a SQL activity that expects to operate on a known database table, a user activity that make assignments to a specific user group, a web service invocation to a specific URL.
- **Endpoints and security**—Although it is technically possible to directly invoke a process that is intended for reuse, a reusable process is (by definition) designed to be used within the context of a calling process. Service Oriented Architecture (SOA) is one way to maximize reuse, so typically processes are exposed and consumed as web services. However, as there are a number of invocation methods available (WatchFolder, EMail, Remoting, TaskManager) it cannot be assumed that the process will always be invoked using either web services or the built-in Enterprise JavaBeans (EJB) endpoint that is automatically provided for all activated processes. Following a similar rationale, the process cannot make assumptions about the user context associated with the current request. If you find yourself contemplating the security or invocation policy of the process, it is likely that the process may not be suitable for reuse. Use of categories is one way to distinguish between common, reusable processes and those that are designed for a specific purpose and/or endpoint.

Technical guide feedback

We welcome your comments. Please send any feedback on this technical guide to LCES-Feedback@adobe.com.

Summary

With proper planning, project teams can quickly and collaboratively develop LiveCycle ES applications. Thoughtful use of user roles, permissions, repository structure, and fragments will reduce the development effort, ease the application maintenance burden, and improve performance. This paper provides many common best practices, but you will undoubtedly refine and develop your own as you work with LiveCycle ES.



Adobe

Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
USA
www.adobe.com

Adobe, the Adobe logo, Flex, Flex Builder, LiveCycle, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

© 2008 Adobe Systems Incorporated. All rights reserved. Printed in the USA.
95010462 03/08