LIGHTROOM SDK 6

PROGRAMMERS GUIDE

Adobe

Adobe Photoshop Lightroom SDK 6 Programmers Guide

Contents

Preface

The Adobe® Photoshop® Lightroom® Software Development Kit (SDK) is a scripting interface to Lightroom that allows you to extend and customize Lightroom functionality. Use the SDK API to write plug-ins for Lightroom. This release allows you to customize the behavior of Lightroom's export and publish operations, define Lightroom-specific metadata for photos, and create customized HTML web galleries.

The Lightroom SDK

The Adobe Photoshop Lightroom SDK is available for download from:

http://www.adobe.com/devnet/photoshoplightroom/

The SDK contains these elements (paths are relative to the location that you choose during installation):

LR_SDK/Manual/Lightroom SDK Guide.pdf	This programming guide.
LR_SDK/API Reference/index.html	The home page for a complete API reference in HTML format.
LR_SDK/Sample Plugins/	Sample code that demonstrates plug-in development. For details of what these do and how to use them, see Chapter 8, "SDK Sample Plug-ins."
flickr.lrdevplugin ftp_upload.lrdevplugin	These samples demonstrate the Service Provider interface. • The first creates a service to upload selected photos to Flickr using Lightroom's Publish Manager service. • The second creates a service that uploads photos to an FTP server, using a customized Export dialog.
custommetadatasample.lrdevplugin	This sample demonstrates the creation of Lightroom-specific metadata, and the customization of the Plug-in Manager dialog and behavior. The sample creates custom metadata fields for use within Lightroom, and a dialog that displays the values of those metadata fields for selected photos. It also demonstrates how to add menu items.
metaexportfilter.lrdevplugin	Demonstrates the Export Filter Provider by defining a post-process action and a related section in the Export dialog. This action offers the user a choice of metadata values to filter on, and removes all photos that do not match that choice from the export operation. There is a predefined function, `shouldRenderPhoto()`, that helps do this.

`websample.lrwebengine`	Demonstrates the web-engine plug-in by defining a simple HTML gallery.
`helloworld.lrdevplugin`	This shows the finished code files produced by the walkthrough in Chapter 9, "Getting Started: A Tutorial Example." The walkthrough demonstrates the architecture of standard plug-ins, and the basic techniques for creating user-interface controls for Lightroom dialogs and panels.
`mymetadata.lrdevplugin` `mysample.lrwebengine`	These contain the finished code from the tutorial walkthroughs in Chapter 10 and Chapter 11.

The Lua language

The SDK defines a Lua-language scripting API. For guidance on using the Lua language, we recommend reviewing the official Lua web site, http://www.lua.org/, and the book "Programming in Lua, second edition," by Roberto Ierusalimschy. Lightroom 5 uses version 5.1.4 of the Lua language.

The Lightroom SDK provides a Lua scripting environment, which extends the Lua languages with an object-oriented infrastructure; see "The Lightroom SDK scripting environment" on page 12.

About this document

This document has the following sections:

- Chapter 1, "Using the Lightroom SDK," provides an introduction to the Lightroom SDK, with the basics of how Lua plug-ins work, and the concepts and terminology of the Lightroom SDK scripting environment.
- Chapter 2, "Writing a Lightroom Plug-in," provides an overview of standard plug-in architecture, and explains how to customize the Plug-in Manager dialog and plug-in loading behavior.
- Chapter 3, "Creating Export and Publish Services," explains how to use the SDK to create an *export plug-in*, which customizes the behavior of Lightroom's Export and Publishing Manager dialogs and export processing.
- Chapter 4, "Working with Metadata," explains how to define customized public or private metadata fields for Lightroom.
- Chapter 5, "Creating a User Interface for Your Plug-in," explains how to create and populate a dialog box or a custom section in the Plug-in Manager dialog or Export dialog with user-interface elements, using the `LrView` and `LrDialogs` namespaces.
- Chapter 6, "Writing a Web-engine Plug-in," explains how to create a Lightroom plug-in that defines a new type of web engine. This type of plug-in uses a slightly different architecture.
- Chapter 7, "Using ZStrings for Localization," explains how to localize your plug-in's user interface for different languages.
- Chapter 8, "SDK Sample Plug-ins," walks through the installation and usage of the sample plug-ins provided with the SDK.

- Chapter 9, "Getting Started: A Tutorial Example," walks through a "Hello World" tutorial to help you create your first plug-in.
- Chapter 10, "Defining Metadata: A Walkthrough," shows how to define Lightroom-specific metadata properties for your photos.
- Chapter 11, "Web Gallery Plug-ins: A Tutorial Example," shows how to define your own HTML web-engine plug-in.

Conventions used in this document

The following type styles are used for specific types of text:

Typeface Style	Used for:
`Monospace font`	Lua code and literal values, such as function names.
`Monospace bold`	Points of interest in code samples.
`Monospace italic`	Variables and placeholders.
Regular italic	Introduction of terms.

1 Using the Lightroom SDK

This chapter provides an introduction to the Lightroom SDK:

- "Writing plug-ins for Lightroom" on page 11 describes the basics of how Lua plug-ins work, including details of the information file and file-system locations.
- "The Lightroom SDK scripting environment" on page 12 explains the concepts and terminology of the Lightroom SDK scripting environment, and provides details of what tools are available to you within the SDK scripting environment.

Writing plug-ins for Lightroom

The Lightroom SDK allows you to customize and extend certain Lightroom features by creating plug-ins. For an overview of the plug-in architecture, see Chapter 2, "Writing a Lightroom Plug-in."

In general, a plug-in consists of Lua-language files (scripts) that define the plug-in functionality, and an information or manifest file that describes the contents of the plug-in. The information file must have a specific name, and be placed in a folder with the Lua source files and resource files; the folders may need to be in specific locations. The type of plug-in is determined by the folder and file placement, and by file naming conventions.

In the current release these features are extensible:

- **Menu customization:** Plug-ins can add new menu items in the Plug-in Extras menu; see Chapter 2, "Writing a Lightroom Plug-in."
- **Export and publish functionality**: You can create an *export plug-in*, which customizes the behavior of Lightroom's Export and Publishing Manager dialogs and export processing. You can add or remove items from the dialog, alter or augment the rendering process, and send images to locations other than files on the local computer. See Chapter 3, "Creating Export and Publish Services."
- **Metadata**: You can define customized public or private metadata fields for Lightroom. Public or private metadata can be associated with individual photos. You can also define new ways of arranging the display of metadata within the Library module's Metadata panel. See Chapter 4, "Working with Metadata."
- **Web engine functionality**: You can create an *HTML web-engine plug-in*, which defines a new type of HTML photo gallery. The engines you define appear in the Gallery panel at the upper right of the Web module. See Chapter 6, "Writing a Web-engine Plug-in."

The Lightroom SDK provides an API with which to define a user interface for your plug-in; see Chapter 5, "Creating a User Interface for Your Plug-in."

The Lightroom SDK scripting environment

The SDK defines a Lua-language scripting API. The Lua scripting language is a fast, light-weight, embeddable scripting language. For information about the language, see http://www.lua.org/.

The Lightroom scripting environment provides a programming structure that includes some enhancements to the basic Lua-language constructs. This section describes the API usage and terminology.

- The API defined for the Lightroom SDK is fully documented in the *Lightroom SDK API Reference*, which is part of the SDK. When you have installed the SDK, the home page is at:

  LR_SDK_install_location/API Reference/index.html

Namespaces, classes, and objects

Lightroom defines a *namespace* as a table containing a suite of functions. This is somewhat like the Lua *module*; however, Lightroom does not use or support the module system that was introduced in Lua 5.1 (see http://www.lua.org/manual/5.1/manual.html#5.3).

Lua does not have an object-oriented programming model, but it does allow Lua tables to be used in an object-like fashion, which the Lightroom SDK does. Lightroom's object and class model is derived from the one described in Chapter 16 of "Programming in Lua," available online at http://www.lua.org/pil/16.html.

In Lightroom terminology, *object* and *class* are used in the typical object-oriented fashion: that is, a class is a description of a set of behaviors that are associated with a particular data structure, and an object is a single instance of that class.

- The Lightroom SDK defines a set of namespaces and a set of classes; see "Accessing namespace functions directly" on page 12 and "Creating objects" on page 14. Plug-ins cannot define either namespaces or classes.
- The Lua language defines built-in namespaces and global functions, of which a subset are accessible in the Lightroom SDK Lua environment. See "Using built-in Lua features" on page 19.

Accessing namespace functions directly

You can access a namespace by using the built-in function `import()`; it takes a single parameter, the name of the namespace to be loaded, and returns the table of functions, which you can then access using dot notation.

For example:

```
local LrMD5 = import 'LrMD5' -- assign namespace to local variable
local digest = LrMD5.digest( 'some string' ) -- call "digest()" function in namespace
```

This example shows the convention of assigning the namespace to a variable of the same name. This practice is not enforced in any way, but helps avoid confusion.

The Lightroom SDK defines these namespaces; for complete details, see the *Lightroom SDK API Reference*.

Namespace	Description
`LrApplication`	Application-wide information; provides access to the active catalog.
`LrBinding`	Allows you to define data relationships between UI elements and other objects.
`LrColor`	Both a namespace and a class. Allows access to color values, specified using RGB or grayscale values or by name.
`LrDate`	Allows you to create and manipulate date-time values.
`LrDialogs`	Allows you to show messages in predefined modal dialogs.
`LrErrors`	Allows you to format Lua error strings to be used in error dialogs.
`LrExportSettings`	Allows you to check or set an image file format for an export operation.
`LrFileUtils`	Allows you to manipulate files and folders in the file system in a platform-independent manner.
`LrFtp`	Both a namespace and a class. The namespace functions allow you to work with the paths and settings for FTP connections created with the `LrFtp` class.
`LrFunctionContext`	Both a namespace and a class. The namespace functions allows you to make functions calls with defined methods for cleaning up resources allocated during the execution of a function or task.
`LrHttp`	Allows you to send and receive data using HTTP. Must be used within a task.
`LrLocalization`	Allows you to localize your plug-in for use in multiple languages.
`LrLogger`	Both a namespace and a class. Provides logging capability.
`LrMath`	Provides additional basic math operations not otherwise available in the Lua language.
`LrMD5`	Provides MD5 digest services.
`LrPasswords`	Provides a mechanism to store passwords in a secure fashion.
`LrPathUtils`	Allows you to manipulate file-system path strings in a platform-appropriate way. (All paths are specified in platform-specific syntax.)
`LrPhotoInfo`	Allows you to get information about individual photo files, such as their dimensions.
`LrPrefs`	Allows you define persistent preferences for your plug-in.
`LrProgressScope`	Both a namespace and a class. Allows you to provide feedback to the user about the progress of a long-running task
`LrRecursionGuard`	Both a namespace and a class. Provides a simple recursion guard for function execution.
`LrShell`	Provides access to shell functions of the platform file browser (Windows Explorer in Windows or Finder in Mac OS).

Namespace	Description
`LrStringUtils`	Provides string manipulation utilities.
`LrSystemInfo`	Provides information about the environment in which Lightroom is running, such as whether it is 32-bit or 64-bit architecture.
`LrTasks`	Allows you to start and manage tasks that run cooperatively on Lightroom's main UI thread.
`LrView`	Both a namespace and a class. The namespace functions allow you to obtain the factory object, create bindings between UI elements and data tables, and share placement between UI elements.
`LrXml`	Both a namespace and a class. The namespace functions allows you to create an XML builder object, and to parse existing XML documents into read-only XML DOM objects.

Creating objects

When you use the `import()` function with a class, it returns a constructor function, rather than a table. Use the constructor to create objects, which you can initialize with specific property values. You can then access the functions and properties through the object using colon notation.

This example shows the standard way to create and use an object:

```
local LrLogger = import 'LrLogger'
    -- LrLogger is a constructor function, not a table with more functions
local logger = LrLogger( 'myPlugin' )
    -- Calling this function returns an instance of LrLogger, which is assigned to
    -- local variable logger. Notice the lowercase naming convention for objects.
logger:enable( 'print' )
logger:warn( 'something bad happened' )
    -- Method calls on the object that was just created.
```

There are some exceptions to this technique. You can create some objects using functions in other objects or namespaces, such as `LrApplication.activeCatalog()`. Others are created and passed to you by Lightroom.

The Lightroom SDK defines these classes; for complete details, see the *Lightroom SDK API Reference*.

Class	Description	Object creation
`LrCatalog`	Provides access to a Lightroom catalog.	Returned by `LrApplication.activeCatalog()` Most classes provide a pointer back to the catalog that contains an object, such as `LrPhoto.catalog`.

Class	Description	Object creation

`LrCollection`	Provides access to a photo collection.	Returned by:
		`LrCatalog:createCollection()`
		`LrCatalog:createSmartCollection()`
		`LrCatalog:getActiveSources()`
		`LrCatalog:getCollections()`
		`LrCatalog:getCollectionByLocalIdentifier()`
		`LrCollectionSet:getChildren()`
		`LrCollectionSet:getChildCollections`
		`LrPhoto:getContainedCollections()`

`LrCollectionSet`	Provides access to a photo collection set.	Returned by:
		`LrCatalog:createCollectionSet()`
		`LrCatalog:getCollectionSets()`
		`LrCollectionSet:getChildren()`
		`LrCollection:getParent()`
		`LrCollectionSet:getParent()`
		`LrCollectionSet:getChildCollectionSets()`

`LrColor`	Encapsulates a color.	Import constructor: `local LrColor = import 'LrColor'`
`LrDevelopPreset`	Provides access to a develop preset.	Returned by `LrDevelopPresetFolder:getDevelopPresets()`

`LrDevelopPreset Folder`	Provides access to a develop-preset folder.	Returned by:
		`LrApplication.developPresetFolders()`
		`LrDevelopPreset:getParent()`

`LrExportContext`	Encapsulates an export context.	Object is passed to your `processRenderedPhotos()` function
`LrExportRendition`	Encapsulates a single photo rendition operation, generated during an export operation.	Returned by `LrExportSession:renditions()`
`LrExportSession`	Provides access to the list of photos and renditions generated during an export operation.	Import constructor: `local LrExportSession = import 'LrExportSession'` An object is also available as the value of `exportContext.exportSession`.
`LrFilterContext`	Provides access to choices the user has made in the Export dialog, and to the list of photos to be exported.	An `LrFilterContext` object is passed to your plug-in as a parameter to your service script's `postProcessRenderedPhotos` function. You cannot import the namespace or access the properties and functions in any other way.

Class	Description	Object creation

`LrFolder`	Provides access to a file-system folder that contains photos.	Returned by:
		`LrCatalog:getFolders()`
		`LrCatalog:getActiveSources()`
		`Lrcatalog:getFolderByPath()`
		`LrFolder:getParent()`

`LrFtp`	Both a namespace and a class. The object represents an FTP connection.	Import the namespace: `local LrFtp = import 'LrFtp'` • Use the factory function, `LrFtp.create()`

`LrFunctionContext`	Both a namespace and a class. Use the object to register the cleanup handlers for the called function execution.	Import the namespace:
		`local LrFunctionContext = import 'LrFunctionContext'`
		• Object is passed to functions called using the namespace calling functions. For example:
		`LrFunctionContext.callWithContext( "showCustomDialog", function(contextObject) -- body of called function end)`

`LrKeyword`	Encapsulates a keyword.	Returned by:
		`LrCatalog:createKeyword()`
		`LrCatalog:getKeywords()`
		`LrKeyword:getChildren()`
		`LrKeyword:getParent()`

`LrLogger`	Provides a mechanism for writing debug output that can be viewed with an external log-viewer application.	Import constructor: `local LrLogger = import 'LrLogger'`
`LrObservableTable`	Implements an observable properties table.	Create an observable table by calling `LrBinding.makePropertyTable()`. Some API functions create observable tables for you.
`LrPhoto`	A single photo or virtual copy in Lightroom's active catalog.	Returned by many functions of `LrCatalog`, `LrCollection`, `LrExportSession` and so on. Many classes provide access to the photo objects that they are associated with, such as `LrKeyword:getPhotos()`.
`LrPlugin`	Provides access to the plug-in configuration, including the path and resources.	Access the object for your plug-in with the global variable `_PLUGIN`.

Class	Description	Object creation
`LrProgressScope`	Allows you to provide feedback to the user about the progress of a long-running task.	Import constructor: `local LrProgressScope = import 'LrProgressScope'`

`LrPublished Collection`	Provides access to a published-photo collection.	Access functions are parallel to those for `LrCollection`, such as:
		`LrCatalog:createPublishedCollection()`
		`LrPublishedCollectionSet:getChildren()`

`LrPublished CollectionSet`	Provides access to a published-photo collection set.	Access functions are parallel to those for `LrCollectionSet`, such as:
		`LrCatalog:getPublishedCollectionSets()`
		`LrPublishService:getChildCollectionSets()`

`LrPublishedPhoto`	Encapsulates the publishing information associated with a photo that is part of a published collection.	Returned by `LrPublishedCollection:getPublishedPhotos()`
`LrPublishService`	Provides access to a named publishing service.	Returned by `LrCatalog:getPublishServices()`
`LrRecursionGuard`	Provides a simple recursion guard for function execution.	Import constructor: `local LrRecursionGuard = import 'LrRecursionGuard'`
`LrVideoExportPreset`	Represents a single video export preset.	Returned by `LrExportSettings.videoExportPresets`.
`LrView`	Allows you to construct dialog box elements.	Import the namespace: `local LrView = import 'LrView'` • When creating a dialog to be invoked from a menu command, import namespace and obtain a factory object with the namespace function `LrView.osFactory()`. • When extending a Lightroom dialog, a factory object is passed to `sectionsForTopOfDialog()` and `sectionsForBottomOfDialog()`
`LrWebViewFactory`	Allows you to construct elements for panels in the Web module.	In a web-engine plug-in's `galleryInfo.lrweb` file, this object is passed to the `views` function. It extends the standard view factory with additional functions. This object is only available within web-engine plug-ins.

Class	Description	Object creation
`LrXml`	Both a namespace and a class. There are two types of object: • A builder object allows you to create and manipulate XML documents. • A DOM object is read-only, and allows you to examine an existing XML document.	Import the namespace: `local LrXml = import 'LrXml'` • Create a builder object with the namespace function `LrXml.createXmlBuilder()`. • Create a DOM object with the namespace function `LrXml.parseXml()`.

Accessing object functions and properties

A few classes (`LrFtp`, `LrView`, `LrXml`, and `LrFunctionContext`) act as both classes and namespaces, and allow you to call some functions directly in the imported namespace, using dot notation. By convention, the documentation uses lowercase names, as well as colon notation, to indicate that a function is called on an instance. For example:

```
LrFtp.appendFtpPaths() -- A namespace function
ftpConnection:path() -- An object function
```

Classes define both functions and properties. To access properties in objects, use the dot notation. Again, the documentation uses the lowercase naming convention to indicate an instance of a class:

```
exportRendition.photo -- An object property
```

A property can have no value; a nil property value is not the equivalent of false, zero, or the empty string. Setting a nil value for a property that has a default value causes the property to revert to the default.

Using function contexts for error handling

The `LrFunctionContext` namespace and class is a programming utility for error handling.

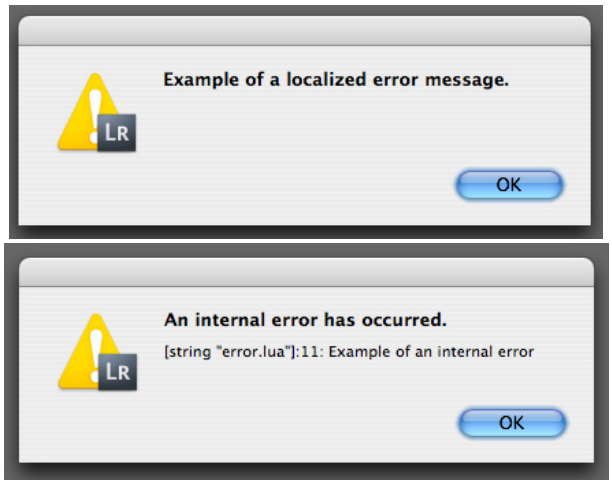
Use `LrFunctionContext.callWithContext()` to wrap a function call. This allows you to attach any number of handler functions to the call that respond to errors that may occur during the execution of the wrapped function, or clean up resources regardless of how the wrapped function terminates. If you attach multiple cleanup or error handlers to the wrapped function, the handlers are called in reverse order of attachment.

Some of the functions in `LrTasks` work in conjunction with `LrFunctionContext` to provide standardized error reporting behavior. For instance, `LrTasks.startAsyncTask()` calls `LrDialogs.attachErrorDialogToFunctionContext()`. This ensures that errors that occur during the execution of the task are reported to the user and not silently forgotten. Lightroom provides predefined error dialogs that you can customize with explanatory messages, as shown in the following example. You can use `LrTasks.startAsyncTaskWithoutErrorHandler()` if you wish to provide your own error reporting instead.

Example: Function context with an error dialog

```
local LrDialogs = import 'LrDialogs'
local LrErrors = import 'LrErrors'
local LrFunctionContext = import 'LrFunctionContext'
LrFunctionContext.callWithContext( 'error handling demo', function ( context )
    -- If an error is thrown during this function call context,
    -- show a standard error dialog.
    LrDialogs.attachErrorDialogToFunctionContext( context )
    -- The code needed to perform your task goes here
    -- For illustration, force an error here, throw error two different ways
    if showInternalError then -- in some case
        error "Example of an internal error" --call built-in error() function
    else -- otherwise, use the LrErrors throw function
        LrErrors.throwUserError( LOC "$$$/MyPlugin/Error/Example=Example of
                                                    a localized error message." )
    end
end )
```

This shows the predefined error dialog with customized text, according to how the error was thrown:



Using built-in Lua features

The Lua language defines built-in namespaces and global functions, of which only a subset are supported in the Lightroom SDK Lua environment, as follows:

Lua global functions

- Available in Lightroom:

  `assert()`, `dofile()`, `error()`, `getmetatable()`, `ipairs()`, `load()`, `loadfile()`, `loadstring()`, `next()`, `pairs()`, `pcall()`, `rawequal()`, `rawget()`, `rawset()`, `select()`, `setmetatable()`, `tonumber()`, `tostring()`, `type()`, `unpack()`

- Not available in Lightroom:

  `collectgarbage()`, `gcinfo()`, `getfenv()`, `module()`, `newproxy()`, `package()`, `setfenv()`

Lua standard namespaces

- Available in Lightroom: `io`, `math`, `string`

- Not available in Lightroom: `package`
- Partially available:
  - `os`: Contains only the functions `clock()`, `date()`, `time()`, and `tmpname()`. All other functions removed. Use `LrFileUtils`, `LrDate`, and `LrTasks` instead.
  - `table`: Contains all functions except `getn()`, `setn()`, and `maxn()`, which are deprecated as of Lua 5.1.
  - `coroutine`: Contains only the functions `canYield()` and `running()`.
  - `debug`: Contains only the function `getInfo()`.

Defining function contexts and tasks

Your plug-in can use a *function context* to create and manage a *task*, which is a kind of lightweight process that runs cooperatively on Lightroom's main (user interface) thread. If your service defines a lengthy export operation that would block the main Lightroom process, you should run it as a background task, using functions such as `LrFunctionContext.postAsyncTaskWithContext()`. Some API functions, such as those in the `LrHttp` namespace, are only available when called from within a background task.

The `LrFunctionContext` object helps you clean up resources following the execution of a function. You can register any number of cleanup handlers to respond to the success or failure of a function. You must create property tables within a function context, so that Lightroom can remove notifications when the table is no longer needed.

You do not create instances of `LrFunctionContext` directly. They are created by the calling functions, and exist only for the lifetime of the function call or task. Access the calling functions, such as `LrFunctionContext.postAsyncTaskWithContext()`, directly from the imported namespace. A `functionContext` object is passed as the first parameter of the call, followed by any other parameters you provide. Use the passed object to provide the cleanup handlers for the called function execution.

In general, you are responsible for creating tasks when needed. There are some exceptions, however. Many of the plug-in callback functions in the export and publish APIs are called from within tasks that Lightroom starts. These are marked as such in the API reference.

For details of the `LrFunctionContext` and `LrTasks` functions, see the *Lightroom SDK API Reference*.

Including scripts with require()

Lightroom defines a `require()` function that works in a similar, but more narrowly-defined, fashion from the version that exists in Lua. The `require()` function takes a single parameter, which is the name of another Lua file in the same plug-in. On the first call, this file is loaded and executed in the context of its plug-in; the return value is saved. If the `require()` function is called again in the same plug-in, its saved value is used (unless the entire plug-in has been garbage collected, in which case the required file is loaded and executed again).

A script to be executed this way typically has the effect of defining a table containing a suite of functions. For example:

SomeFile.lua

```
SomeFile = {}
       -- Typically a file that is required will define a global table whose name
```

```
        -- matches the file name.
        -- Note that this global is defined in a special function environment for your
        -- plug-in and does not affect Lightroom as a whole.
        -- You can give this table any name that does not conflict with built-in names
        -- and keywords from Lua and Lightroom. In general, avoid names that start with
        -- Lr to avoid conflicts with future versions of Lightroom.
function SomeFile.doSomething( arg )
    return tostring( arg )
end
```

Usage of require()

```
require 'SomeFile.lua'
    -- Causes SomeFile.lua to be executed and the value of SomeFile defined above
    -- becomes available in the scope of this file.
SomeFile.doSomething( 42 )
```

Lua syntax notes

For people unfamiliar with the Lua language, here are some syntax conventions and usage notes.

- Literal strings can be surrounded by either single or double quotes. These two statements are equivalent:

  ```
  local a = 'my string'
  local a = "my string"
  ```

- Semicolons at the ends of statements are optional. We typically omit them.

- If you call a function with a single parameter that is a string literal or a table, you can omit the parentheses around the argument list. This is frequently done when calling the built-in functions `import()` and `require()`.

  These three statements are equivalent (where `func` is a variable containing a valid function):

  ```
  func( "foo" )
  func "foo"
  func 'foo'
  ```

  These two statements are also equivalent; the simpler syntax is commonly used when building view descriptions:

  ```
  func( { a = 1, b = 2 } )
  func{ a = 1, b = 2 }
  ```

- It is useful to read the chapter on table constructors (http://www.lua.org/pil/3.6.html). There are several shorthand formats that we use widely, especially in view descriptions. For example, these forms are equivalent:

  ```
  local t = { a = 1, b = 2 }
  local t = { [ "a" ] = 1, [ "b" ] = 2 }
  local t = {}; t.a = 1; t.b = 2
  ```

- Lua defines an *array* as a table with numbered keys. Arrays in Lua are 1-based; that is, the first item in the array is at index 1, not index 0.

- The value `nil` evaluates to a Boolean value of false, but numbers (including 0) evaluate to true. Thus, in a conditional, only `nil` and `false` are considered false. If you use 0 as the condition of an `if` or `while` statement for example, the statement *is* executed, because the number 0 is a true value.

- Lightroom defines Boolean globals, `WIN_ENV` and `MAC_ENV`, which you can use to determine which platform your script is running on. The `LrSystemInfo` namespace (first available in version 3.0 of the Lightroom SDK) can provide additional information about the platform, including whether is 32-bit or 64-bit.

2 Writing a Lightroom Plug-in

The Lightroom SDK allows you to create plug-ins that customize the behavior of Lightroom in specific ways. Most types of plug-in share a common architecture, which is discussed in this chapter.

- Web Gallery plug-ins use a different architecture; see Chapter 6, "Writing a Web-engine Plug-in."

The Plug-in Manager dialog allows a user to load plug-ins from any location, enable and disable loaded plug-ins, and remove unused plug-ins. Your plug-in can customize the dialog by adding sections. See "Customizing plug-in load behavior" on page 33.

Writing standard plug-ins for Lightroom

The Lightroom SDK allows you to customize the behavior of Lightroom in specific ways using a standard plug-in (as opposed to a web-engine plug-in). The behavior implemented by your plug-in is provided by one or more Lua scripts. An information file installed in a standard plug-in folder identifies your plug-in scripts to Lightroom, and associates the plug-in with a unique name.

- You can add items to the File, Library, or Help menus to start an operation.
- Your plug-in can add a new export or publish destination with an *Export Service Provider* or *Publish Service Provider*. Such a service can also customize the Export or Publishing Manager dialog by adding and removing sections as appropriate for the destination when the user selects it. See "Creating an export or publish service" on page 37.
- Your plug-in can intercept the export process with an *Export Filter Provider*, which can apply further processing to photos that the user has chosen to export. The *post-process action* that a filter defines is applied after Lightroom's initial rendering, and before the photos are sent to the final destination. Each filter can add a section to the Export dialog, in which the user can select options and set parameters. See "Adding an export post-process action" on page 43.
- In addition to or instead of defining export customizations, your plug-in can define custom metadata fields for Lightroom. See Chapter 4, "Working with Metadata."

Declaring the contents of a plug-in

In addition to the Lua script or scripts that define your extension to Lightroom functionality, your plug-in must contain a file named `Info.lua` that describes the plug-in to Lightroom, using a Lua table of descriptive items. The table can or must include these items:

`LrSdkVersion`	Required number	The preferred version of the Lightroom SDK for this plug-in. Should be set to 6.0 for the current release; older plug-ins may have a value of 1.3, 1.4, 2.0, 3.0, 4.0 or 5.0.
`LrSdkMinimumVersion`	Optional number	The minimum version of the SDK that this plug-in can use. If your plug-in works with Lightroom 2.0, 3.0, 4.0 or 5.0 but it also provides new features specific to Lightroom 6, set this value to 2.0 and `LrSdkVersion` to 6.0. Default is the value of `LrSdkVersion`.

`LrToolkitIdentifier`	Required string	A string that uniquely identifies your plug-in. Use Java-style package names (your domain name in reversed sequence). You can use the Plug-in Manager to add multiple plug-ins with the same identifier, but only one of them can be enabled. If you enable one, any other plug-in that shares the same plug-in ID is automatically disabled. Note that `com.adobe.*`, which is used in the examples, is reserved for plug-ins written by Adobe; your own plug-ins will use your own domain name (`com.myCompany.*`).
`LrPluginName`	Required for 2.0 or newer string	A localizable string for the plug-in's display name, which appears in the Plug-in Manager dialog. Required for SDK version 2.0 or newer; ignored by earlier versions. If a plug-in defined for an earlier version is loaded in Lightroom 2.0 or newer, the Plug-in Manager displays the title of the first Export Service Provider, or if no Export Service Provider is defined, the base name of the plug-in folder.
`LrPluginInfoProvider`	Optional string	The name of the Lua file (*service definition script*) to be executed when the plug-in is loaded. This script can define functions that run when plug-in is selected or deselected in the Plug-in Manager dialog, and can add sections to the Plug-in Manager dialog that are shown when the plug-in is selected. It See "Customizing plug-in load behavior" on page 33. Ignored in any Lightroom version older than 2.0.
`LrInitPlugin`	Optional string	The name of a Lua script that is loaded and executed when your plug-in is loaded or reloaded. See "Customizing plug-in load behavior" on page 33 This item is ignored if `LrSdkVersion` is less than 2.0.
`LrForceInitPlugin`	Optional Boolean	When true, forces the initialization script to run at application startup, even for plug-ins that do not provide export or publish services or define custom metadata, as long as the plug-in contributes at least a menu item. Otherwise, the initialization script does not run until the plug-in's first use. Ignored in any Lightroom version older than 4.0.
`LrPluginInfoUrl`	Optional string	The URL of your web site, or a page that provides information about your plug-in. See "Customizing plug-in load behavior" on page 33 Ignored in any Lightroom version older than 2.0.

`LrShutdownPlugin`	Optional string	The name of a Lua script that is loaded and executed when the user unloads this plug-in. This script is executed only if the user loaded or reloaded the plug-in through the Plug-in Manager dialog, or approved updating of the catalog structure, during the current Lightroom session. Ignored in any Lightroom version older than 3.0.
`LrShutdownApp`	Optional string	The name of a Lua script that is loaded and executed when Lightroom is shutting down. See "Application termination script" on page 29. This item is ignored if `LrSdkVersion` is less than 4.0.
`LrEnablePlugin`	Optional string	The name of a Lua script that is called when the user enables this plug-in from the Plug-in Manager dialog. Ignored in any Lightroom version older than 3.0.
`LrDisablePlugin`	Optional string	The name of a Lua script that is called when the user disables this plug-in from the Plug-in Manager dialog. Ignored in any Lightroom version older than 3.0.
`VERSION`	Optional table	Allows you to provide a version number for your plug-in that is displayed in the Plug-in Manager dialog. The version number is for your own use, and need not relate to Lightroom's version number. The table has these members: `major` (number) `minor` (number) `revision` (number) `build` (number) `display` (string) The numeric values are assembled in order to create a version designation in the default format: *`major.minor.revision.build`* The `build` value is omitted if zero; both `revision` and `build` are omitted if both are zero. If you prefer to display a version number in a different format, include the display string in the `display` field. You can use `LOC` to localize the string; see "The LOC function" on page 150. Ignored in any Lightroom version older than 2.0.

`LrExportMenuItems`
`LrLibraryMenuItems`
`LrHelpMenuItems`
	Optional
	
	table of tables
		These allow you to add new script-defined menu items to the Plug-in Extras submenus that appear in Lightroom's File, Library, and Help menus.
		
		For details of the member tables, see "Defining menu items for a plug-in" on page 35:

`LrExportFilterProvider`	Optional table of tables	Adds one or more new export filters, which can process photos before they are rendered for the export destination. Each item is a table with these entries: `title` (string): The display name of the filter. `file` (string): The name of the Lua file (*filter definition script*) to be executed when the filter is chosen. See "Adding an export post-process action" on page 43. `id` (string): A unique identifying string for this filter. `requiresFilter` (string, optional): The identifier for another filter that must be used with this one. Can be combined with other services (export services, custom metadata) or can be the only service provided by the plug-in. Ignored in any Lightroom version older than 2.0.
`URLHandler`	Optional string	Optional. The name of a file that provides custom URL handling. The file must return a table that contains a `'URLHandler'` entry. This must be a function that takes a URL as a parameter. Lightroom is configured to be the registered receiver of URLs starting with `'lightroom://'`. When it receives such a URL, it checks registered handlers to find an appropriate one. For example, the plug-in with the ID `'my.plugin.id'` can register a handler for URLs starting with `'lightroom://my.plugin.id'`. Ignored in any Lightroom version older than 4.0.

`LrExportServiceProvider`	Optional table of tables	Adds one or more new export destinations or publish service providers. Each item is a table with these entries: `title` (string): The display name of this export destination. `file` (string): The name of the Lua file (*service definition script*) to be executed when the export destination or publish service is chosen. See "Defining an export service" on page 37. `builtInPresetsDir` (string, optional): The name of a subdirectory in the plug-in directory that contains predefined export settings values. This must be a simple folder name; it cannot contain any path-significant characters, such as slashes. Lightroom preset files are identified by the `.lrtemplate` extension. (Publish services are not associated with presets.) `id` (string, optional): A unique identifier for this export service. If none is provided, a simple consecutive number value (1, 2, 3...) is assigned. Can be combined with other services (export filters, custom metadata) or can be the only service provided by the plug-in.
`LrMetadataProvider`	Optional table of tables	Adds custom metadata fields, available only in Lightroom. There can be only one such item in a plug-in. It contains a string, the name of the Lua file (*metadata definition script*) that defines the fields. See "Adding custom metadata" on page 71. Can be combined with other services (export services, export filters) or can be the only service provided by the plug-in. Ignored in any Lightroom version older than 2.0.
`LrMetadataTagsetFactory`	Optional table of tables	Adds a tagset of predefined metadata fields. The user can select defined metadata tagsets from a menu in the Metadata panel. Some tagsets are supplied by Lightroom; this allows your plug-in to supply additional tagsets. See "Adding custom metadata tagsets" on page 76. Ignored in any Lightroom version older than 2.0.

`LrAlsoUseBuiltInTranslations`	Optional Boolean	Controls behavior of built-in LOC function. When true, strings that are not found in the plug-in's translation file are checked against Lightroom's translation file for the current language. Note that string keys are not guaranteed to remain the same across version releases. Ignored in any Lightroom version older than 3.0.
`LrLimitNumberOfTempRenditions`	Optional Boolean	Controls whether Lightroom will throttle the number of temporary image files on disk waiting for the plug-in's processing during export. The intent of this option is to allow protection against high disk space consumption during large exports. If true, the plug-in is expected to remove the temporary rendition files when it is done with them. Ignored in any Lightroom version older than 5.0.

The loader environment

The `Info.lua` file is a special Lua environment that is much more restrictive than the general SDK Lua environment in which other scripts run.

- The standard Lua namespace `string` is available, and you can use the `LOC` function for localization of display strings in this file (see Chapter 7, "Using ZStrings for Localization").
- You can use `WIN_ENV` and `MAC_ENV` environment variables, and the `_VERSION` variable that contains most of the version information otherwise available through `LrApplication.versionTable()`.

However, you cannot use any of the other Lua or Lightroom globals defined in the SDK scripting environment, (see "The Lightroom SDK scripting environment" on page 12). For example, you cannot use `import` or `require` in this context.

Here is an example of an `Info.lua` file for a plug-in that adds items to the Lightroom menus:

```
return {
    LrSdkVersion = 5.0,
    LrSdkMinimumVersion = 1.3, -- minimum SDK version required by this plug-in

    LrToolkitIdentifier = 'com.adobe.lightroom.sdk.helloworld',

    LrPluginName = LOC "$$$/HelloWorld/PluginName=Hello World Sample",

    -- Add the menu item to the File menu.

    LrExportMenuItems = {
        title = "Hello World Dialog",
        file = "ExportMenuItem.lua",
    },

    -- Add the menu item to the Library menu.
    LrLibraryMenuItems = {
        {
            title = LOC "$$$/HelloWorld/CustomDialog=Hello World Custom Dialog",
```

```
            file = "ShowCustomDialog.lua",
        },
        {
            title = LOC "$$$/HelloWorld/MultiBind=Hello world Custom Dialog with
MultipleBind",
            file = "CustomDialogWithMultipleBind.lua",
        },
        {
            title = LOC "$$$/HelloWorld/RadioButtons=Hello world RadioButtons",
            file = "RadioButtons.lua",
        },
        {
            title = LOC "$$$/HelloWorld/DialogObserver=Hello world Custom Dialog with
Observer",
            file = "CustomDialogWithObserver.lua",
        },
    },
    VERSION = { major=3, minor=0, revision=0, build=200000, },
}
```

You can find more examples in the sample plug-ins provided with the SDK.

Application termination script

`LrShutdownApp` specifies a script that is called at application shutdown time. This script must return a table that contains an `LrShutdownFunction` member; this function takes two arguments, `doneFunction` and `progressFunction`, in that order.

- The `doneFunction` takes no arguments; it is called when the plug-in's shutdown tasks have been completed (which allows the plug-in's shutdown operation to use asynchronous tasks).
- The `progressFunction` is invoked to report progress during the execution of the plug-in's shutdown tasks. This function takes two arguments, percent complete (number between 0 and 1) and a descriptive display string for the progress dialog. It must return a Boolean value, true if the user clicks Cancel in the progress dialog.

If 10 seconds pass without progress being reported by a call to the `progressFunction`, the shutdown task is assumed hung or terminated, and application shutdown proceeds. The plug-in shutdown must monitor this status and respond appropriately to it.

The `LrDialogs` namespace is not available in the environment of the shutdown task.

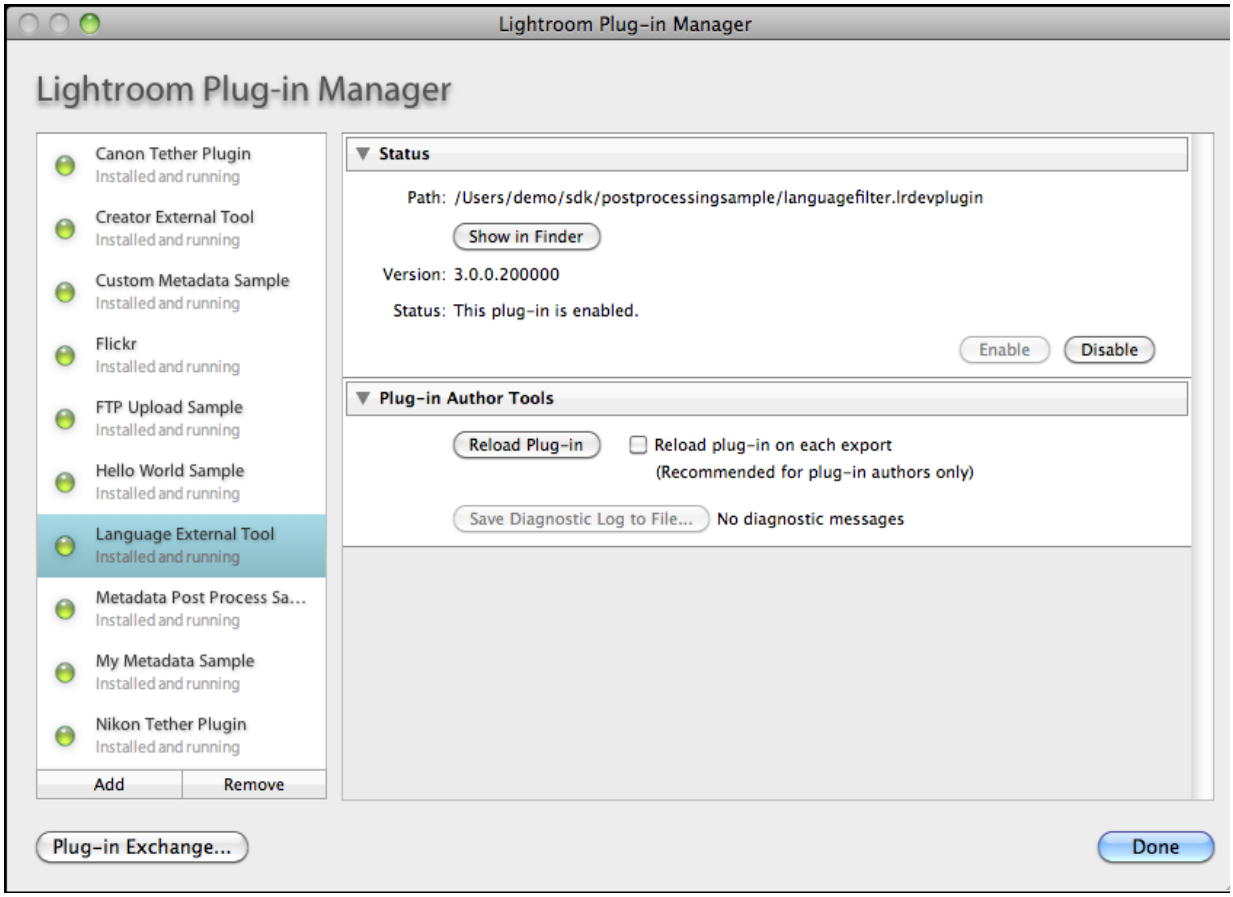
Delivering a standard plug-in

Package your Lua files (the information file, `Info.lua`, and all Lua scripts) in a single folder with a suffix of `.lrplugin`; for example, `MyPluginName.lrplugin`.

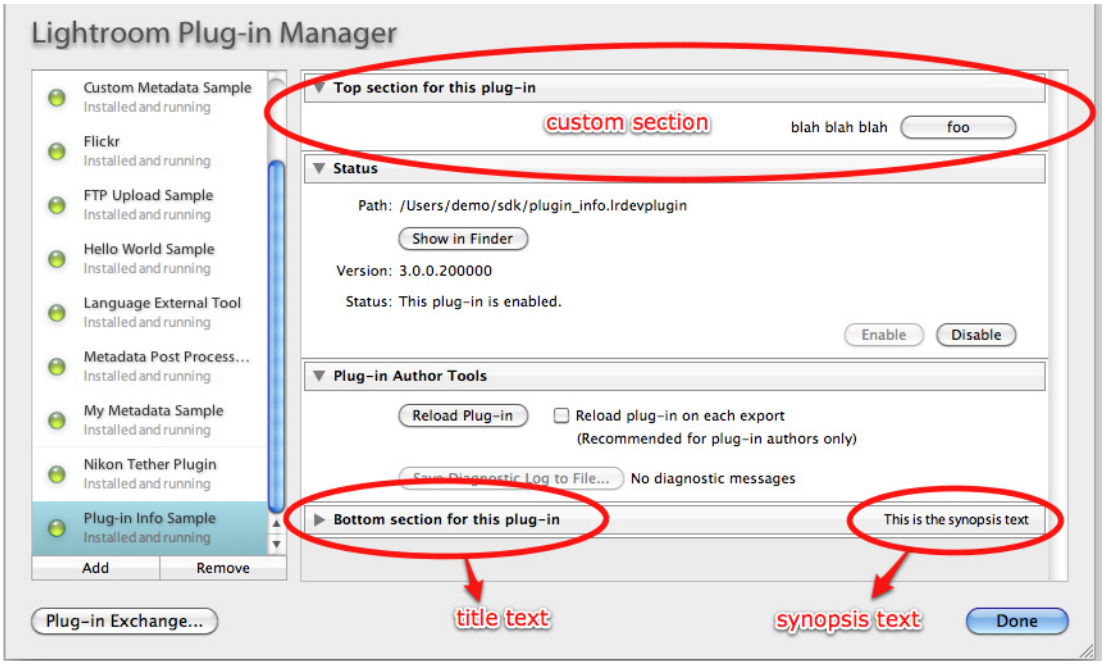
NOTE: In Mac OS, the suffix `.lrplugin` creates a package, which looks like a single file. For convenience, you can use the suffix `.lrdevplugin` during development, and change the extension to `.lrplugin` for delivery. The `.lrdevplugin` suffix is recognized by Lightroom but does not trigger the package behavior in the Mac OS Finder.

A plug-in folder can reside in any location on the hard drive. Users can load the plug-in using the Add button in the Plug-in Manager. Once it is added, users can enable or disable it through the dialog, reload it using the Plug-in Author Tools in the dialog, or unload it using the Remove button.

The standard Plug-in Manager dialog in Lightroom looks like this:



Your plug-in can customize the Plug-in Manager, adding sections that appear when a user selects your plug-in in the dialog. For example, in the following figure, the plug-in called "Plug-in Info Sample" defines two custom sections, one above and one below the Lightroom-defined sections. They appear only when the plug-in is selected in the list. The sections are collapsible, and you can define a descriptive string (a *synopsis*) to appear on the right side when the section is closed.



For details of how to define these sections, see See "Adding custom sections to the Plug-in Manager" on page 34.

Automatic plug-in loading

Lightroom automatically checks for plug-ins in the standard Modules folder where other Lightroom settings are stored:

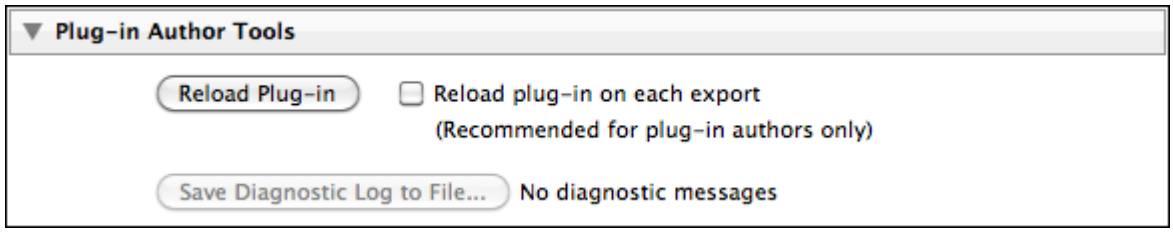
In Mac OS (current user)	`~/Library/Application Support/Adobe/Lightroom/Modules`
In Mac OS (all users)	`/Library/Application Support/Adobe/Lightroom/Modules`
In Windows XP	`C:\Documents and Users\`*`username`*`\Application Data\Adobe\Lightroom\Modules`
In Windows 7/Vista	`C:\Users\`*`username`*`\AppData\Roaming\Adobe\Lightroom\Modules`

You may want to use this location if, for example, you are writing an installer that installs a Lightroom plug-in and also installs a helper application.

Plug-ins that are installed in this location are automatically listed in the Plug-in Manager dialog. You can use the dialog to enable or disable such a plug-in, but not to remove it. The Remove button is dimmed when such a plug-in is selected.

Debugging standard plug-ins

The Plug-in Manager also provides access to tools for plug-in authors.



This section is generally not needed by end users, and is closed by default. If you open the "Plug-in Author Tools" section, you can:

- Reload a plug-in after you make code changes.
- Choose to have Lightroom reload the plug-in automatically on each export or publish operation.

  NOTE: Reloading a plug-in interactively or automatically after export does not reload any localization dictionaries supplied with that plug-in. The translation dictionaries are read only when the plug-in is first loaded or Lightroom is restarted. See Chapter 7, "Using ZStrings for Localization."
- Choose a file to which to save diagnostic messages if a plug-in fails to load, or encounters an error at any stage of its operation.

The Lightroom SDK does not supply a development environment in which to debug your plug-ins, but it does supply the `LrLogger` namespace, which allows you to configure a log file and viewer for trace information of various kinds, and add tracing statements to your scripts. For an example, see "Debugging your plug-in" on page 186.

Finding deprecated calls

You can add a debugging flag to your configuration file (`config.lua`) to help you find occurrences of calls to deprecated functionality in your plug-in.

```
sdkDeprecation.action=throw|log
```

When you specify the action `throw`, Lightroom throws an exception each time a deprecated call is executed.

To write deprecated call occurrences to a file, you must specify the action `log`, and also define this logger in your `config.lua` file:

```
loggers.AgSdkDeprecation = {
    logLevel = "info",
    action = "logfile",
}
```

Lightroom writes log message to this file:

- (Windows) *<user_home>*`\My Documents\AgSdkDeprecation.log`
- (Mac OS) `~/Documents/AgSdkDeprecation.log`.

Customizing plug-in load behavior

You can customize your plug-in's behavior when it is loaded or selected in the Plug-in Manager dialog. To do this, the `Info.lua` file for your plug-in can include these entries:

- **`LrPluginInfoProvider`** points to a script that can return any or all of the following function definitions which customize appearance or behavior of the Plug-in Manager dialog when the plug-in is selected:

Item	Description
`startDialog` `endDialog`	Initialization and termination functions that run when your plug-in is selected or deselected in the Plug-in Manager dialog. The same items are defined slightly differently in an Export Service Provider to run when the service is selected or deselected in the Export dialog; see "Initialization and termination functions for the Plug-in Manager" on page 34.
`sectionsForTopOfDialog` `sectionsForBottomOfDialog`	Definitions for one or more new sections to display in the Plug-in Manager dialog when your plug-in is selected in the dialog. The same items can be defined in an Export Service Provider or Export Filter Provider to customize the Export dialog when the service or filter is selected or deselected there; see "Adding custom sections to the Plug-in Manager" on page 34 for details.

- **`LrInitPlugin`** points to a script that runs when the plug-in is loaded or reloaded. You can use this to initialize values in your plug-in's global function environment, which are protected from garbage collection. When the plug-in is reloaded, a new environment is created. All previous values are discarded and this function is executed again.
- **`LrShutdownPlugin`** and **`LrShutdownApp`** point to scripts that allow you to control the termination procedure for your plug-in, specified by `LrShutdownPlugin` and `LrShutdownApp`.. You can use these to clean up private data. See "Application termination script" on page 29.
- **`LrPluginInfoUrl`** gives the URL of your web site, or a page that provides information about your plug-in. This URL is displayed in the Status section of the Plug-in Manager dialog when your plug-in is loaded and selected. The URL is also displayed as part of the error message if your plug-in fails to load properly or cannot be found.
- **`LrEnablePlugin`** and **`LrDisablePlugin`** provide functions to be called when your plug-in is enabled or disabled in the Plug-in Manager dialog.

For example:

```
return {
    LrSdkVersion = 5.0,
    LrSdkMinimumVersion = 2.0, -- minimum SDK version required by this plug-in
    LrToolkitIdentifier = 'com.adobe.lightroom.sample.plug-in-info',
    LrPluginName = LOC "$$$/PluginInfo/Name=Plug-in Info Sample",
    LrPluginInfoProvider = 'PluginInfoProvider.lua',
    LrInitPlugin = 'PluginInit.lua',
    LrPluginInfoUrl = 'http://www.mycompany.com/lrplugin_info.html',
}
```

Initialization and termination functions for the Plug-in Manager

In addition to the **LrInitPlugin** and **LrShutdownPlugin** scripts that are run on load and unload, you can provide functions to be called when your plug-in is selected or deselected. To do so, the service definition script for your Plug-in Info Provider should return these table entries, which contain the function definitions:

```
startDialog = function( propertyTable ) ... end,
endDialog = function( propertyTable ) ... end,
```

When the functions are defined in a Plug-in Info Provider:

- The `startDialog` function is called whenever your plug-in is selected in the Plug-in Manager.
- The `endDialog` function is called when the user deselects the plug-in in the Plug-in Manager.

NOTE: These same entries can be supplied by an Export Service Provider, although the definitions are slightly different. Functions defined in an Export Service Provider are executed only when the plug-in is selected in the Export dialog, never from the Plug-in Manager dialog. See "Initialization and termination functions for services" on page 40.

The *propertyTable* parameter for both functions is an empty, observable table which you can use to keep private data for your plug-in. (See "Binding UI values to data values" on page 97.) This table is discarded when your plug-in is deselected in the Plug-in Manager or when the Plug-in Manager dialog is closed. It is not preserved across sessions. You can use `LrPreferences` if you want to save information across invocations.

These are blocking calls. If you need to start a long-running task (such as network access), create a *task* using the `LrTasks` namespace. See "Defining function contexts and tasks" on page 20.

Adding custom sections to the Plug-in Manager

Your plug-in can define one or more sections to be displayed in the Plug-in Manager dialog (when defined in an `LrPluginInfoProvider` entry) or in the Export or Publishing Manager dialog (when defined in an `LrExportServiceProvider` entry). The custom sections can be shown above or below the Lightroom standard sections for the dialog.

To customize the dialog, define a function that returns a table of sections, defined using `LrView` objects. The function is the value of one of these service entries:

```
sectionsForTopOfDialog = function( viewFactory, propertyTable ) ... end,
sectionsForBottomOfDialog = function( viewFactory, propertyTable ) ... end,
```

> **NOTE:** Similar functions can be defined in an Export Service Provider, to customize the Export dialog when the export destination is selected or the Publishing Manager dialog for a publish service, and also in an Export Filter Provider, to add a section to the dialog when a post-process action is selected. See "Customizing the Export and Publishing Manager dialogs" on page 55 and "Adding an export post-process action" on page 43.

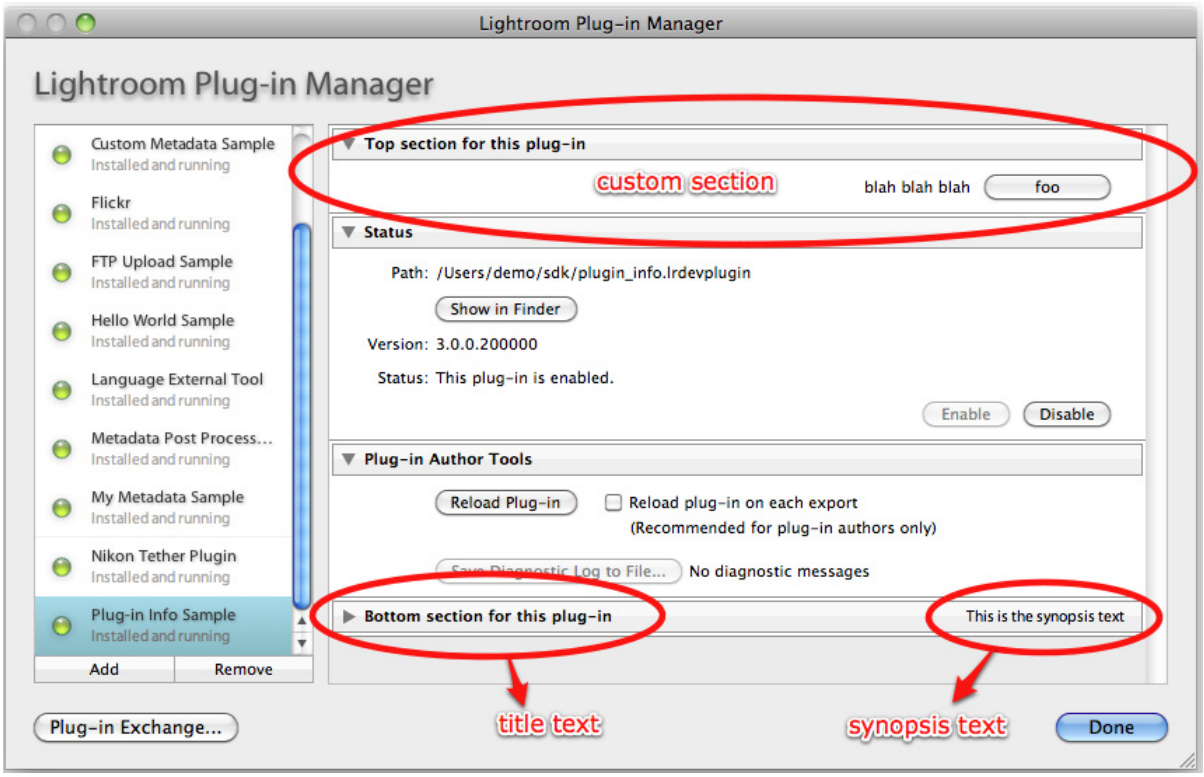
Lightroom passes your function a factory object which allows you to create the `LrView` objects that define the elements of your sections; that is, UI controls, such as buttons and text, and containers that group the controls and determine the layout. For additional details of the dialog elements you can create with `LrView`, see "Adding custom dialog views" on page 85."

The function that you define here returns a table of tables, where each table defines one dialog section:

```
sectionsForTopOfDialog = function( viewFactory, propertyTable )
    return {
              { ...section entry ... },
              { ...section entry ... },
           ...
           }
    }
end
```

A section entry table defines the contents of an implicit container, which Lightroom creates to hold your view hierarchy.

- Each section entry sets a `title` and `synopsis` for the section; the section is identified by the `title` text on the left, and is collapsible. When in the collapsed state, the `synopsis` text is shown on the right.
- The rest of the table entry creates the UI elements that are shown when the section is expanded. To create the UI elements, use the `LrView` factory passed to your top-level `sectionsFor...` function. This process is explained in more detail in "Adding custom dialog views" on page 85.



When adding sections to the Plug-in Manager, the *propertyTable* parameter for both functions is an empty, observable table which you can use to keep private data for your plug-in for a dynamic user interface. See "Binding UI values to data values" on page 97.

Defining menu items for a plug-in

To define a menu item that starts your plug-in's UI, or performs any other script-defined operation, the table returned by your `Info.lua` file must include one of these items:

```
LrExportMenuItems
LrLibraryMenuItems
LrHelpMenuItems
```

- Items that you add in `LrExportMenuItems` appear in the Plug-in Extras submenu of the File menu (immediately below the Export section).
- Items that you add to the Library and Help menus appear in the Plug-in Extras submenu of those menus.

Each item is a table of tables, and each member table defines one menu item. The member table for a menu item should contain these entries:

`title`	(String) The display name of the menu item.
`file`	(String) The name of the Lua file (*service definition script*) to be executed when the menu item is chosen. See "Defining an export service" on page 37.
`enabledWhen`	(String, optional) A condition under which to enable this menu item. One of: • `photosAvailable`: Item enabled when any photos or videos are available in the grid. • `photosSelected`: Item enabled when any photos are selected. Ignores videos. • `videosSelected`: Item enabled when any videos are selected. Ignores still photos. • `anythingSelected`: Item enabled when any photos or videos are selected. If the selection is very large (>5000 photos), menu items are enabled regardless of this value.

In Windows, the title string can include an ampersand (`&`) character in the name to make the following character a keyboard shortcut or accelerator that invokes that item. (This feature is not available in Mac OS; on that platform, the `&` character is automatically removed if present.)

For example, suppose the table returned by your `Info.lua` file contains this item:

```
LrExportMenuItems = { title = 'My &Plugin', file = 'somefile.lua' }
```

This would create a command in the File > Plug-in Extras menu with the label "My Plugin". The command would execute the script found in `somefile.lua`.

The user can invoke this command using the keyboard accelerator sequence ALT F + U + ENTER + P.

- ALT F brings up the File menu
- U is the keyboard accelerator for the Plug-in Extras submenu
- P is the keyboard accelerator for the new command

3 Creating Export and Publish Services

Plug-ins can customize Lightroom's export and publish behavior. You can:

- Create an Export Filter that modifies a photo after Lightroom performs the initial rendering, but before it is passed to it final export destination.
- Alter the rendering process, or define post-processing actions for rendered photos.
- Send rendered images to locations other than files on the local computer.
- Customize the Export and Publishing Manager dialogs by adding or removing sections.

Creating an export or publish service

Lightroom offers two customizable mechanisms for transferring photos to an external location: *export* and *publish*.

- **Export** is a one-time operation: the user selects some photos, the photos are rendered once and transferred to their destination. Lightroom maintains no further record of the export (although some plug-ins do retain such a record via custom metadata). By default, Lightroom provides export services for a user-selected location on the local hard drive and CD/DVD drive. Plug-ins can define export services to extend this support to other destinations (typically web services or devices). See "Defining an export service" on page 37.
- **Publish** is similar to export, but represents an ongoing relationship between Lightroom and the destination. You publish a collection (or many collections) of photos to a location when you want them to stay up to date as you change the photos. To publish, the user establishes a connection to a destination (typically, but not necessarily, a web service) in the Publishing Manager dialog. By default, Lightroom provides publish service support for user-selected locations on the local hard drive and Flickr. As with export services, plug-ins can define publish services to extend this support to other destinations. See "Defining a publish service" on page 41.

Though they are presented differently, there is a lot in common between these two mechanisms and they share much of the same API. Export services are the simpler of the two, so we'll start by describing this API, then describe additional functionality that is specific to publish services.

Defining an export service

To define the functionality of your export service, write a Lua script that returns a table; each predefined entry in the table describes a specific type of customization. You then declare the name of your service and associate it with the defining script, in the same way you declare any plug-in.

To declare an Export Service Provider, add the following block to your `Info.lua` file:

```
LrExportServiceProvider = {
    title = "Service Name",  -- this string appears as the Export destination
    file = "MyExportServiceProvider.lua",  -- the service definition script
    builtInPresetsDir = "myPresets", -- an optional subfolder for presets
},
```

The `title` and `file` entries are required. You can use the built-in function `LOC` and a ZString if you wish to localize the service's title; see details in Chapter 7, "Using ZStrings for Localization."

The service definition script should return a table that contains:

- A pair of functions that initialize and terminate your export service.
- Settings that you define for your export service.
- One or more items that define the desired customizations for the Export dialog. These types of customizations are defined:
  - Restrict the built-in services offered by the Export dialog.
  - Customize the Export dialog by defining new sections.
- A function that defines the export operation to be performed on rendered photos (required).

Here is an example of a table returned by a service definition script:

```
return {
    startDialog = function( propertyTable ) ... end,
    endDialog = function( propertyTable, why ) ... end,
    exportPresetFields = { { key = 'myPluginSetting', default = 'Initial value' } },
    showSections = { 'fileNaming', 'imageSettings' },
    sectionsForBottomOfDialog = function( viewFactory, propertyTable ) ... end,
    processRenderedPhotos = function( functionContext, exportContext ) ... end
}
```

These are the specific items that can be in the table returned by the service definition script for an Export or Publish Service Provider:

Item	Description
`startDialog` `endDialog`	Initialization and termination functions for your plug-in; see "Initialization and termination functions for services" on page 40.
`exportPresetFields`	A set of properties that you define for your plug-in that you want to be persistent between sessions. See "Remembering user choices" on page 59. These are added to the built-in settings defined by Lightroom; see "Lightroom built-in property keys" on page 61.
`processRenderedPhotos`	A callback function that manages the rendering and subsequent handling of exported photos; see "Customizing the Export and Publishing Manager dialogs" on page 55.

Item	Description
`canExportToTemporaryLocation`	A Boolean value that indicates whether the service provider can place files in a temporary export destination in the local file system. When true, an additional item, "Temporary folder (will be discarded upon completion)" is added to the Export To pop-up menu at the top of the Export dialog. Default is false. (This is tied to `LR_export_destinationType`; see "Lightroom built-in property keys" on page 61.) If the user selects this option, the file naming options in the dialog disappear and the files are written to a hidden temporary folder on the local hard drive. When the Export Service Provider has completed its work, this folder and its contents are deleted. If your plug-in hides the Export Location section of the dialog, you do not need to use this option. The temporary folder behavior happens automatically in that case. Optional in Lightroom SDK 2.0 and later. Ignored in earlier versions.
`showSections` `hideSections`	A callback function that returns a table of built-in sections to include or exclude from those displayed in the Export dialog. See "Restricting existing service functionality" on page 58 for details.
`allowFileFormats` `disallowFileFormats`	A table of file formats to include or exclude from those offered in the Export dialog. See "Restricting existing service functionality" on page 58 for details.
`allowColorSpaces` `disallowColorSpaces`	A table of color spaces to include or exclude from those offered in the Export dialog. See "Restricting existing service functionality" on page 58 for details.
`sectionsForTopOfDialog` `sectionsForBottomOfDialog`	Definitions for one or more new sections to display in the Export dialog; see "Adding custom sections to the Export or Publishing Manager dialog" on page 56 for details.
`hidePrintResolution`	When true, the options for sizing in the Image Sizing section are shown only in pixel units; all mention of print units such as inches, centimeters, and pixels-per-inch are hidden.
additional publish service options	If this is a publish service, there are a number of other options that you can specify; see "Publish service options" on page 42.

Initialization and termination functions for services

You can provide functions to be called when a post-process action or export destination defined by your plug-in is selected or deselected in the Export or Publishing Manager dialog. To do so, the service definition script for your Export Filter Provider or Export Service Provider should return these table entries, which contain the function definitions:

```
startDialog = function( propertyTable ) ... end,
endDialog = function( propertyTable, why ) ... end,
```

- The `startDialog` function is called when the user chooses a post-process action or export destination provided by this plug-in in the Export dialog, or when the destination is already selected when the dialog is invoked, remembered from the previous export operation.
- The `endDialog` function is called when the user deselects the action or export destination in the Export dialog, or dismisses the Export dialog.

NOTE: Similar entries can be supplied by a Plug-in Info Provider, although the definitions are slightly different. Functions defined in a Plug-in Info Provider are executed only when the plug-in is selected in the Plug-in Manager dialog, never from the Export dialog. See "Initialization and termination functions for the Plug-in Manager" on page 34.

The *propertyTable* parameter for both functions is a table which contains the most recent settings for your export plug-in, including both settings that you have defined and Lightroom-defined export settings (see "Remembering user choices" on page 59). When your plug-in is being used as a publish service provider, the property table contains additional values that tell you about the publishing status; see "Publish Service properties" on page 70.

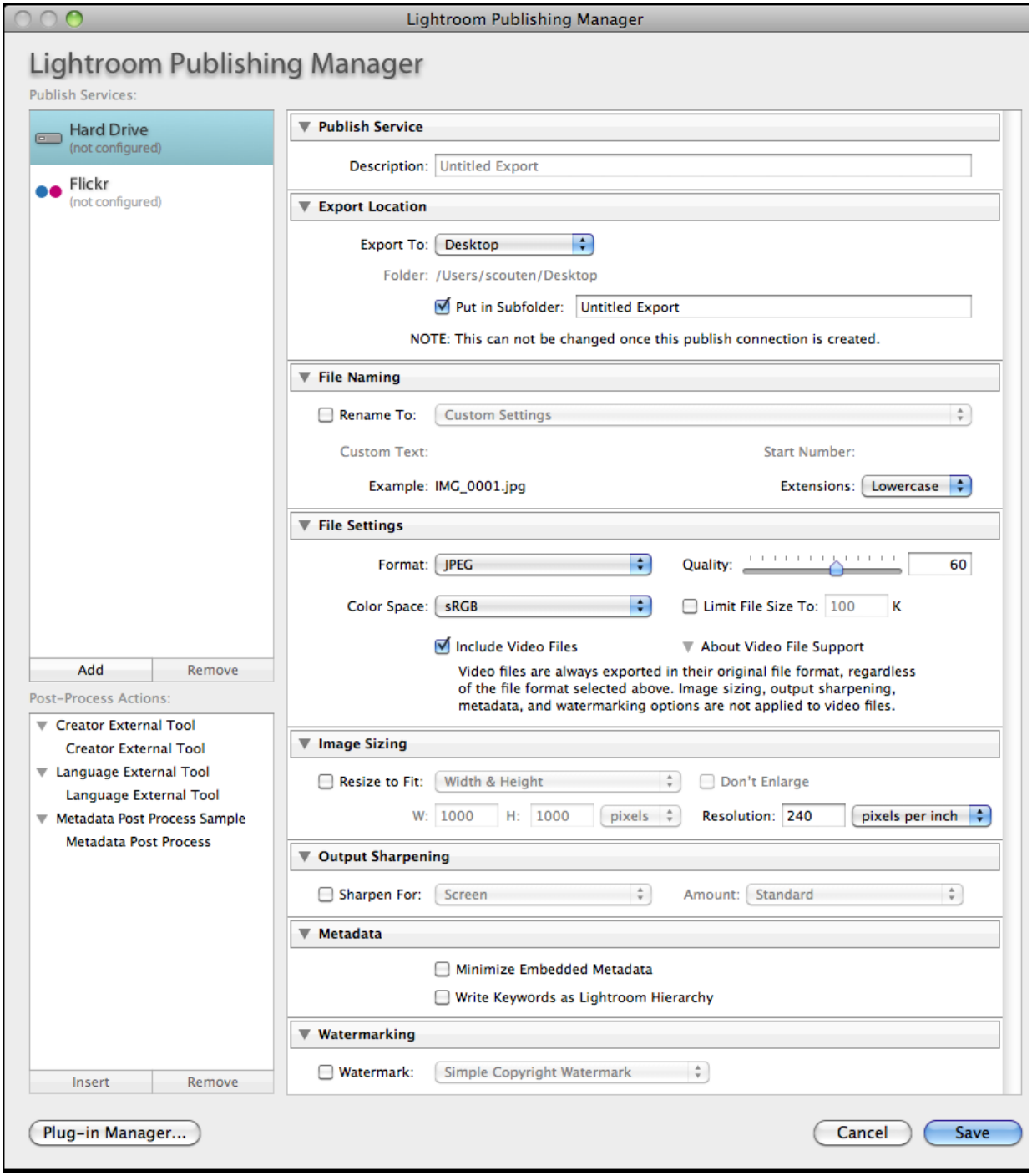
When your plug-in is deactivated, Lightroom calls your `endDialog` function with *why* set to one of the following string values:

`changedServiceProvider`	A different Export Service Provider was chosen as the export or publish destination. Your plug-in is no longer active.
`ok`	The user clicked the "Export" or "Save" button. The Export or Publishing Manager dialog has closed. • For an export service, Lightroom begins exporting images through your plug-in. Do not attempt to start uploading photos at this point; use the `processRenderedPhotos` callback function to do that. See "Customizing the Export and Publishing Manager dialogs" on page 55. • For a publish service, Lightroom creates a new publish service in the Publish Services panel that can be used later to publish images through your plug-in.
`cancel`	The user clicked the "Cancel" button and the Export or Publishing Manager dialog has closed without initiating the operation.

These are blocking calls. If you need to start a long-running task (such as network access), create a *task* using the `LrTasks` namespace. See "Defining function contexts and tasks" on page 20.

Defining a publish service

Starting with version 3.0, Lightroom allows you to send photos to a local or network destination for publication. The publication process is similar to the export process, and the Publishing Manager dialog is largely similar to the Export dialog.



A publish service differs from an export service in these ways:

- The publish service keeps information about what has been published previously, which allows you to export only new or changed images to the same destination.

- A publish service keeps track of the locations to which items have been published, and can access those locations.
- A publish service can manage collections or folders on the remote export destination from within the Lightroom catalog.
- A publish service can retrieve comment and rating information that has been added to an image after publication.

Within each publish service, the user can create one or more collections, just like those in the Library module's Collections panel. These are referred to as *published collections*. The user adds photos to published collections (or has them added automatically if it is a published smart collection). When the user chooses to publish (by choosing Publish Now from the published collection's context menu, for example), Lightroom brings the collection up to date, synchronizing the local photos with the remote copies. Updating the collection may involve one or more of the following operations:

- Export new photos to the collection.
- Re-export existing photos to the collection (updating the rendering or metadata if they have changed since the previous publish).
- Delete photos from the collection.
- Update the sorting sequence for the collection (if applicable).
- Download comments and ratings (if applicable) from the remote service.

Publish service options

To define the functionality of your publish service, write a Lua script that returns a table; each predefined entry in the table describes a specific type of customization. You then declare the name of your service and associate it with the defining script, in the same way you declare any plug-in.

To declare a Publish service, add the following block to your `Info.lua` file:

```
LrExportServiceProvider = {
    title = "Service Name",  -- this string appears in the Publish Services panel
    file = "MyPublishServiceProvider.lua",  -- the service definition script
},
```

Notice that this differs from an Export service in that you do not provide a presets folder location. A plug-in cannot control the settings in the Publishing Manager dialog using presets, although it can provide default values for its own settings, and modify its own settings on dialog startup using the `startDialog` callback.

With the exception of export presets, a publish service is a superset of an export service. The publish service definition script can create any of the elements that an export service does, with only minor differences. In addition, your publish service should define additional elements that control the publication data and processes.

The API defines a number of additional items that allow you to customize the publishing experience for the end user, and to reflect changes in the collection structure made by your plug-in. These are the additional items that can be in the table returned by the service definition script for a Publish Service

Provider; for details of how to define these items and what they do, see the API Reference documentation and the Flickr plug-in.

Adding an export post-process action

An *Export Filter Provider* is a script that allows you to modify a photo after Lightroom performs the initial rendering, but before it is passed to it final export destination, by defining a *post-process action*. A post-process action can modify the rendered images, or can suppress the export of certain photos, based on any criteria that you define. The script is identified by the `LrExportFilterProvider` entry in the plug-in's `Info.lua` file.

A single SDK export plug-in can define one or more Export Filter Providers, one or more Export Service Providers, or both. In any given export session, there must be exactly one Export Service Provider, but there can be any number of post-process actions (or none). A single Export Filter Provider can define multiple actions. Post-process actions are executed in a specific sequence, partly determined by user choices. If you set up a dependency using the `requiresFilter` option, the sequence of execution honors that dependency .

While Export Service Providers can add multiple sections at either the top or the bottom of the Export dialog, each post-process action can provide only one section for the dialog, which is always inserted after Lightroom's built-in sections, and before any "bottom" sections defined by the Export Service Provider.

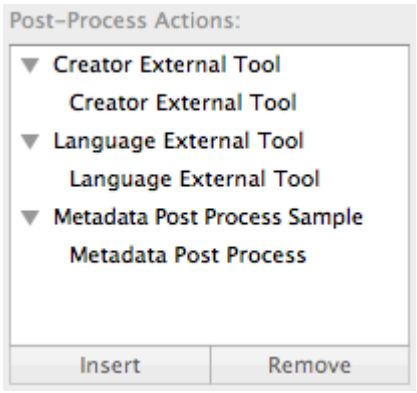
Export Filter Providers cannot define presets of their own, but can be included in an Export Service preset; see "Creating an export or publish service" on page 37.

A post-process action is inserted between Lightroom's initial rendering of photos and the writing of the rendered image files to their destination (either the default destination, or one provided by a plug-in's export service). A post-process action (or set of actions) can be applied to photos that are being exported to any destination; that is, an Export Filter Provider does not need to be part of the same plug-in that provides the export service.

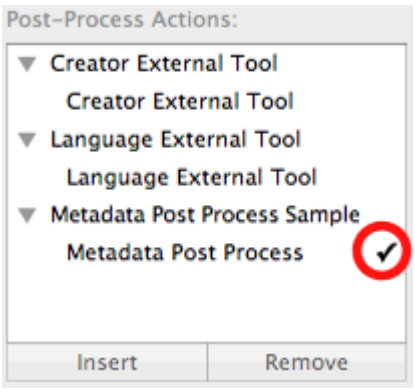
For details, see "How post-process actions are executed" on page 50

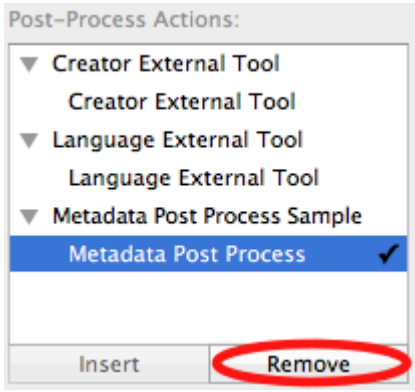
Inserting and removing actions

When any plug-in defining a post-process action is loaded, the action appears in the Post-Process Actions section of the Export dialog, on the left below the Presets section. When you open the plug-in, the individual actions defined by the plug-in appear as choices below it. When you select an action, the Insert button is enabled, allowing you to insert the action into the processing queue. (You can also insert an action by double-clicking it.)

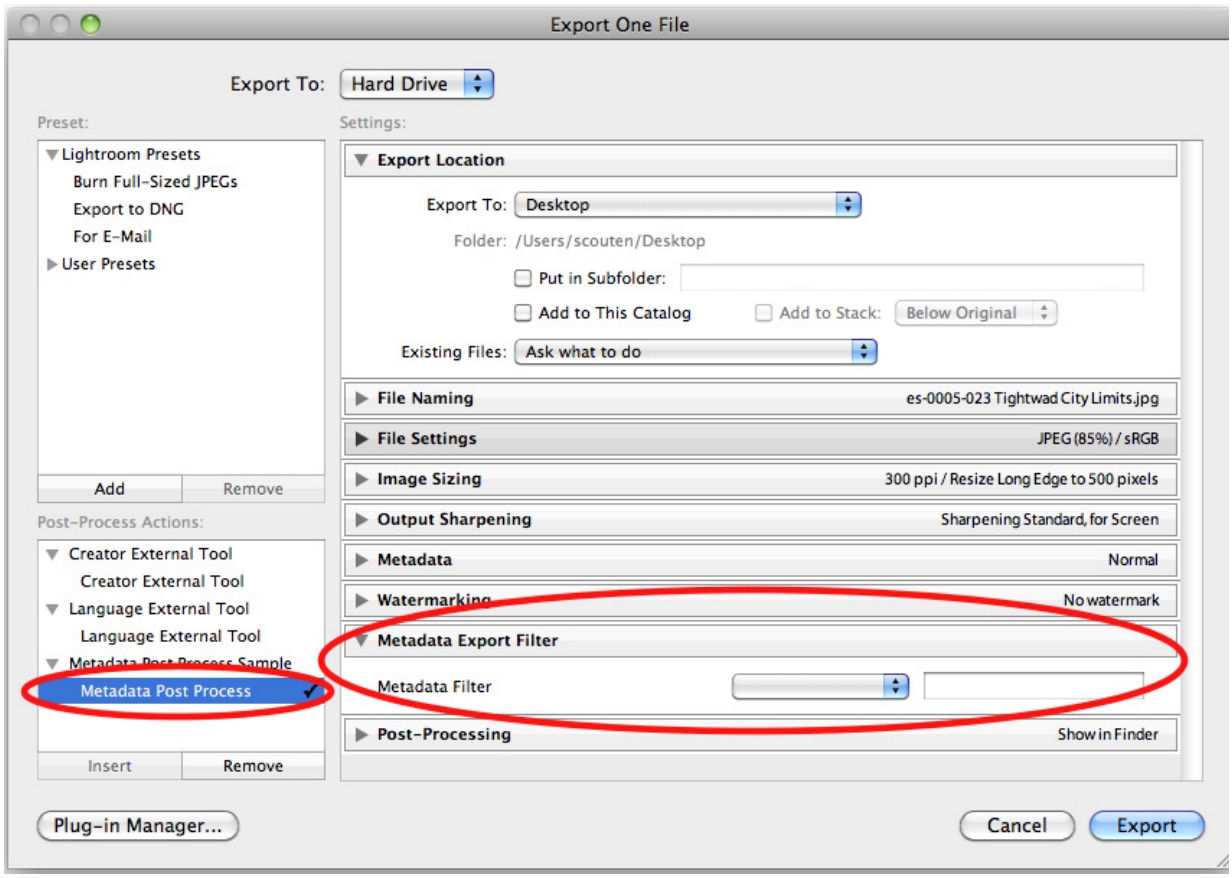


An action that has been inserted is flagged with a check mark; when it is selected, the Remove button is enabled, allowing you to remove it from the processing queue.





When an action is inserted, the related section is shown in the Export dialog. You can also remove the action from the queue by clicking the X icon in the related section.



Action dependencies

You can set up a dependency among a set of actions, such that one action actually performs the photo processing, and other actions in the set are used to determine the parameters for that operation. The one that performs the rendition is typically the only one that defines a `filterRenderedPhotos()` function. This main action is *required* by the others in the set. To declare the dependency, make the ID of the main action the value of the `requiresFilter` option for the dependent actions.

Each post-process action can define a single section for the Export dialog. When the user selects an action, that action's dialog section is shown, along with that of the required filter, if there is one.

For example, suppose your plug-in has defined:

1. MyAction, the main action that performs the filtering operation
2. Color, which allows the user to choose a color to be used by MyAction
3. Lines, which allows a user to choose line widths to be used by MyAction
4. An Export Service Provider that performs an FTP upload

When the user chooses the FTP-upload export destination and clicks **Export**, the service provider requests an export rendition for each photo that is active at the time. If the user does not choose any actions, the request is satisfied directly by Lightroom using `LrExportRendition`.

If the user inserts Color, the dialog shows both the section for defining a border, and the section for MyAction, which is required by Color. After making all the necessary choices for the chosen actions, the

user clicks **Export**. In this case, the request is intercepted and redirected to Color. Color receives a list of renditions that it is expected to satisfy; the action then makes its own rendition request. This request is similarly intercepted and sent to MyAction. MyAction performs the actual processing and makes its own request for renditions. When there are no more actions, the requests are satisfied directly by Lightroom using `LrExportRendition`.

Each action runs in its own task in Lightroom (see "Defining function contexts and tasks" on page 20), which means that the operations performed by each action can be performed in parallel. An action task first requests its renditions, then iterates through them making its transformations as appropriate. When the action is done rendering each photo, it signals the downstream task which can then process the rendered photo. For a more detailed description of the processing path, see "How post-process actions are executed" on page 50.

Declaring export post-process actions

In the `Info.lua` file for a plug-in that defines an export post-process action, you must set `LrSdkVersion = 2.0` (or higher) in order for your filters to be recognized. If you set `LrSdkMinimumVersion = 1.3`, the plug-in can be loaded in Lightroom 1.3, but the post-process actions are ignored.

To declare one or more post-process actions, add the following block to your `Info.lua` file:

```
LrExportFilterProvider = {
    title = "Filter Name",  -- this display string appears in the dialog
    file = "MyExportFilterProvider.lua",  -- the action definition script
    id = "myFilter",  -- a unique identifier for the action
    requiresFilter  = "mainFilter",  -- optional
    supportsVideo = "true" -- optional
},
```

There can be one or many action definitions. Each definition is a table with up to four items:

- `title` (string): The localizable display name of the action, which appears in the Post-Process Actions section of the Export dialog.
- `file` (string): The name of the Lua file (*action definition script*) that provides more information about the action. The script is executed when the export operation is started; that is, when the user clicks Export in the Export dialog.
- `id` (string): An identifying string for this action, unique within this plug-in. Required if more than one action is defined in one plug-in.
- `requiresFilter` (string): Optional, the identifier for the action that performs the processing.
- `supportsVideo` (Boolean): Optional. If true, this filter has access to video files. Default is false; if not supplied, video files are not processed by this filter. Supported in version 4.0 or higher.

For example, this defines three distinct actions, and the first is the one that actually performs the processing. It must be present before either of the other two can run; they simply set parameters to be used by the main action:

```
LrExportFilterProvider = {
    {
        title = "MyAction",
        file = "myAction.lua",
        id = "main",
    },
    {
```

```
        title = "Color",
        file = "colorAction.lua",
        id = "color",
        requiresFilter = "main",
    },
    {
        title = "Lines",
        file = "lineAction.lua",
        id = "lines",
        requiresFilter = "main",
    },
},
```

Defining a post-process action

An action definition script must return a table which can contain these entries. All entries are optional.

`postProcessRenderedPhotos`	A function that defines how this action processes the list of rendered photos that it receives. See "Defining post-processing of rendered photos" on page 48. Typically, only the main action in a group defines this function.
`shouldRenderPhoto`	A function that selects photos to be removed from the list of rendered photos that it receives. See "Removing photos from the export operation" on page 48.
`startDialog` `endDialog`	Initialization and termination functions that run when the action is selected or deselected in the Export dialog. See "Initialization and termination functions for services" on page 40.
`exportPresetFields`	A set of export preset settings that you define for your plug-in (in addition to the built-in settings defined by Lightroom). If provided, these fields are transferred from the Export dialog's property table to any export preset that is created when this post-process action is selected for the current Export session. See "Remembering user choices" on page 59.

`sectionForFilterInDialog`	A function that defines a new section in the Export dialog which appears when the action is selected, allowing the user to make choices that affect the operation. Any action in a group can define a dialog section. The function must conform to this prototype: `sectionForFilterInDialog = function( viewFactory, propertyTable ) ... end,` This function is defined in the same way as `sectionsForTopOfDialog` and `sectionsForBottomOfDialog`, except that the function returns a single section definition, not an array of section definitions; see "Adding custom sections to the Export or Publishing Manager dialog" on page 56. The section is always added at the bottom of the Export dialog, but above any "bottom" sections defined for the active Export Service Provider. The property table passed to this function is shared among all Export Filter Providers and Export Service Providers defined by this plug-in.

Removing photos from the export operation

If you wish to remove photos from the list of those to be exported, based on criteria of your choosing, your action definition script can define a function named `shouldRenderPhoto()`. This function receives two parameters, the *export settings* and the *current photo*, and is called successively on each entry in the list of rendered photos passed to the action.

```
shouldRenderPhoto = function ( exportSettings, photo )...end,
```

The function should return true if the photo should remain in the list and be passed to the next action or exported, or false if it should be removed from the list.

For a publish operation, if a post-process action returns false for a potentially publishable photo, that photo is not published, but remains in the "New Photos to Publish" state.

This example is for a simple filter that removes photos that do not have a minimum star rating:

```
function RatingExportFilterProvider.shouldRenderPhoto( exportSettings, photo )
    local minRating = exportSettings.min_rating or 1
    local shouldRender

    local rating = photo:getRawMetadata( 'rating' )
    shouldRender = rating and rating >= minRating

    return shouldRender
end
```

Defining post-processing of rendered photos

To specify exactly how each photo should be modified after it is rendered and passed to the action, the action-definition script for your main action defines a function named `postProcessRenderedPhotos()`.

```
postProcessRenderedPhotos = function( functionContext, filterContext ) ... end,
```

This function takes two parameters, a *function context* and a *filter context*. It can retrieve each photo from the `filterContext.renditionsToSatisfy` property, and process it as desired. The list of renditions provides access to the export settings with which the photos were originally rendered, and can check or modify those settings and rerender the photos as needed.

The processing is typically performed by an external application. You can build a command string and pass it to a platform-specific shell for execution, using `LrTasks.execute()`.

This processing function typically looks something like this:

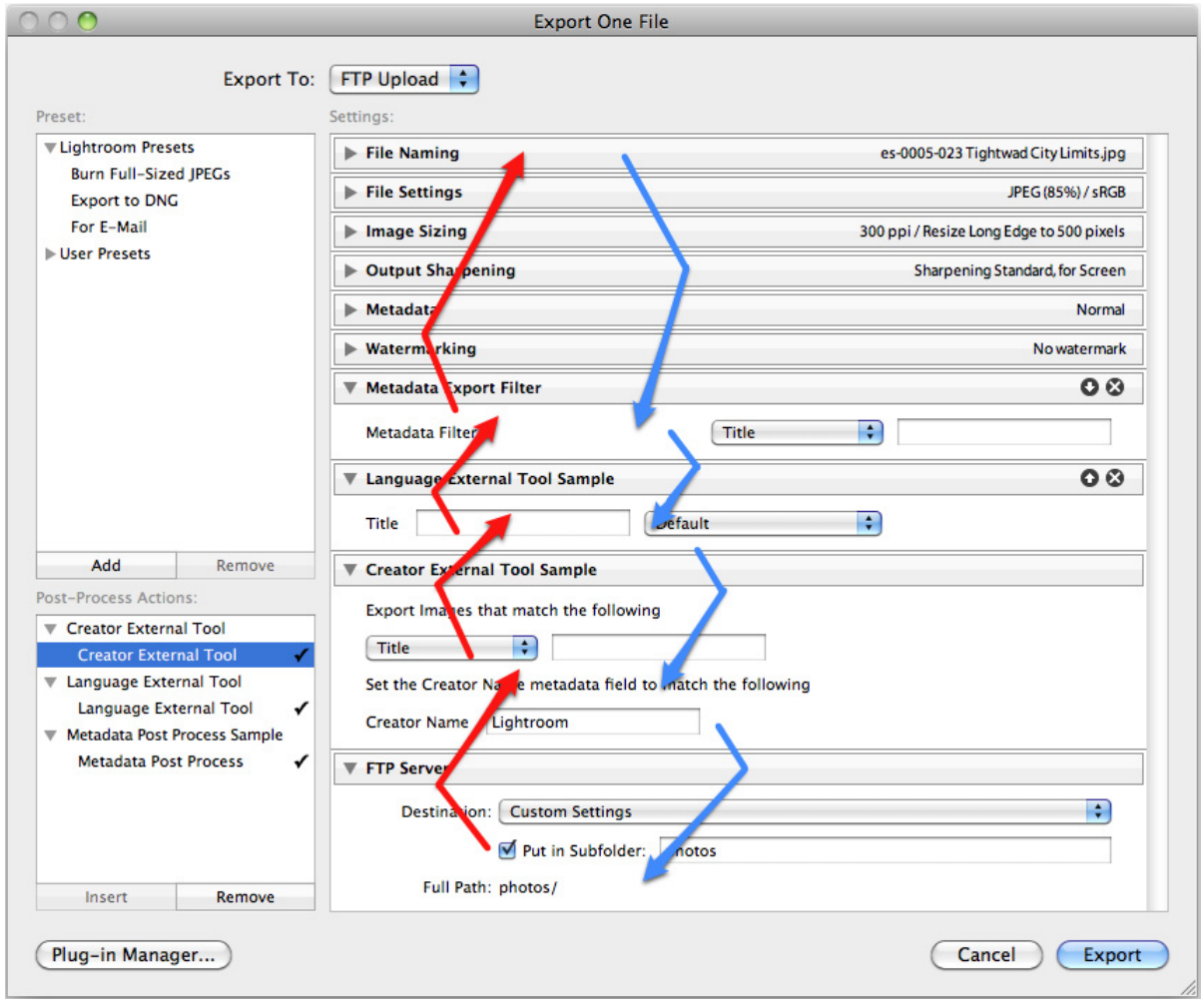
```
function SimpleExternalToolFilterProvider.postProcessRenderedPhotos( functionContext,
filterContext )

    -- Optional: If you want to change the render settings for each photo
    -- before Lightroom renders it, write something like the following.
    -- If not, omit the renditionOptions definition, and also omit
    -- renditionOptions from the call to filterContext:rendition()
    local renditionOptions = {
        filterSettings = function( renditionToSatisfy, exportSettings )
            exportSettings.LR_format = 'TIFF'
            return os.tmpname()
            -- ... if you wanted Lightroom to generate TIFF files
            -- and override the configured filename when providing
            -- input images to this post-process action.
            -- By doing so, you assume responsibility for creating
            -- the file type that was originally requested and placing it
            -- in the location that was originally requested in your
            -- filter loop below.
        end,
    }

    for sourceRendition, renditionToSatisfy in filterContext:renditions(
renditionOptions ) do
        -- Wait for the upstream task to finish its work on this photo.
        local success, pathOrMessage = sourceRendition:waitForRender()
        if success then
            -- Now that the photo is completed and available to this filter,
            -- you can do your work on the photo here.
            -- It would look somethinglike this:
            local status = LrTasks.execute( 'mytool "' .. pathOrMessage .. '"' )
                -- (This tool is hypothetical.)
                -- You may need to use escapes in the file name so that
                -- special characters are not interpreted by the OS shell
                -- (cmd.exe in  Windows or bash in Mac OS).
                -- In Windows, enclose the whole command in double quotes.

            -- If your tool cannot process the photo as intended, use
            -- something like this to signal a failure for this rendition only:
            if status ~= (desired status) then
                renditionToSatisfy:renditionIsDone( false, "error message" )
            end
            -- (Replace "error message" with a user-readable string explaining why
            -- the photo failed to render.)

            -- It is neither necessary nor harmful to call renditionIsDone if the
            -- rendition has completed successfully.
            -- The iterator for filterContext:renditions calls it
            -- automatically if you have not already done so.
        end
```

```
    end
end
```

How post-process actions are executed

During an export operation, rendition requests and state information are passed up a chain of processors from the Export Service Provider to the Lightroom built-in render engine, and rendered photos are passed back down again. The sequence is followed for each photo in the list of photos to be exported. This section explains the sequence of calls, and what information is passed at each point.

This Export Dialog shows three Export Filter Providers, whose actions have all been inserted in the processing queue. The Export Service Provider for FTP Upload (one of the sample plug-ins included in the SDK) has been selected.



The post-process actions are always invoked in the order in which they appear in the dialog, but the export operation traverses the stack several times, either top-to-bottom (blue arrows), or bottom-to-top (red arrows). In this discussion, the terms *upstream* and *downstream* refer to the downward flow; for example, when photos that have been rendered by the built-in render engine (using information passed up from the providers) are passed back down to be modified, and finally exported:

- An *upstream provider* means the post-process action immediately above the current one in the dialog, which provides a rendered photo to the current action for further processing. When there are no more actions, the final upstream provider is Lightroom's built-in rendering engine.
- A *downstream consumer* means the post-process action immediately below the current one in the dialog, which receives a rendered photo from the current one, its upstream provider. When there are no more actions, the final downstream consumer is the Export Service Provider that sends the rendered photo to the final destination.

When the user starts the export operation by clicking Export, Lightroom constructs an `LrExportSession` object with the settings and photos chosen in the dialog.

It then calls `exportSession:doExportOnCurrentTask()`, which performs the following operations. (If you wish to start an export without using the export dialog, you can make the same calls yourself.)

In this discussion, an Export Service Provider is called a *service*, and an Export Filter Provider is called a *filter*.

Stage 1: Deciding how the photos should be rendered

If the service has defined an `updateExportSettings()` function, it is called. This function takes one argument, the export-settings table, and allows the service to force certain render settings to its preferred or required settings. For example, to force a specific size for the exported photos, you could use this definition:

```
updateExportSettings = function( exportSettings )
    exportSettings.LR_size_maxHeight = 400
    exportSettings.LR_size_maxWidth = 400
end
```

The filters, if any, are not involved at this stage.

Stage 2: Deciding what photos should be rendered

The filters are invoked from top-to-bottom. For each filter:

- If it is defined, the `shouldRenderPhoto()` function is called for each photo. If it returns false, the photo is removed from the list of photos to export (and thus does not get passed to the downstream consumer). If it returns true, the photo remains in the list and is passed to the downstream consumer.
- If `shouldRenderPhoto()` is not defined for the filter, all photos are passed to the downstream consumer.

Stage 3: Requesting renditions

1. The export session (`LrExportSession`) generates rendition request objects (`LrExportRendition`) for every photo that was not removed in Stage 2. (The actual rendering of the photos does not start yet.)

   During this stage, Lightroom can show a dialog message if a photo already exists at the proposed destination location. You can control this behavior using the `LR_collisionHandling` setting in the export settings table.

2. The service's `processRenderedPhotos()` function is called. If no such function exists, a default one is provided that performs the steps described below.

IMPORTANT: Each of the providers (the export service, filters, and Lightroom's built-in rendering engine) runs in its own task (using `LrTasks`), so these loops operate in parallel. It is likely that each provider will be running simultaneously. Several photos can be in process simultaneously by different providers. This allows the overall export operation to complete much more quickly than it would if every photo had to go through all of the steps before the processing of the next photo could begin.

3. The `processRenderedPhotos()` function calls `exportContext:renditions()` and then waits for each rendition to be completed by calling `rendition:waitForRender()`. (We will discuss the completion of this loop in Stage 4.)

4. The service's rendition requests are sent to its upstream provider (that is, the bottom-most filter, or if there are no filters, Lightroom's built-in render engine).

5. For each filter, if a `postProcessRenderedPhotos()` function is defined, it is called. The function is called only once, regardless of the number of photos being exported.

   This function should enter into a loop of the form:

   ```
   for sourceRendition, renditionToSatisfy in filterContext:renditions() do ... end
   ```

   - The filter context object generates a new rendition request (`LrExportRendition`) for each of the renditions provided this filter.

   - The `renditions()` iterator provides two values: `sourceRendition` (the new rendition to be satisfied by the upstream provider) and `renditionToSatisfy` (the corresponding rendition that this filter is expected to satisfy for its downstream consumer).

ADVANCED: If the filter provider wishes to request a different file format than it is expected to satisfy, it can do so using the `renditionOptions/filterSettings` code snippet shown in "Defining post-processing of rendered photos" on page 48. This might be a good idea as a way to avoid re-encoding (and thus degrading) JPEG files. If you do this, you are still responsible for providing an output file in the exact format and location required by `renditionToSatisfy`.

   - The filter must wait for each rendition to be completed by its upstream provider by calling `sourceRendition:waitForRender()`. (We will discuss the completion of this loop in Stage 4.)

   If there is no `processRenderedPhotos()` function defined by this filter, the filter is simply left out of the loop and all rendition requests instead go to the upstream provider.

IMPORTANT: Each filter must generate a photo conforming exactly to the specifications provided to it. In particular, it must provide a suitable photo file in the specified format and at the exact path specified by the downstream consumer. If it cannot do so, it must use `renditionToSatisfy:renditionIsDone( false, message )` to indicate why not. It must never provide a file of a different format than that requested.

6. After all of the filters have had an opportunity to intercept the rendition requests, the requests are finally passed to Lightroom's built-in rendering engine.

Stage 4: Processing rendered photos

As Lightroom completes each rendition request, it signals completion by allowing the corresponding `rendition:waitForRender()` call to complete.

The rendering loops described in stage 3 then finish in top-to-bottom sequence for each photo. For each filter that defines the `postProcessRenderedPhotos()` function:

1. The `waitForRender()` call completes, meaning that the upstream provider has completed its attempt to render the photo. If that attempt was successful, a valid photo file conforming to the specifications requested by this filter is present at the path specified by the `sourceRendition`; that is, specified by this filter when it requested the rendition from its upstream provider.

2. The filter can now do whatever processing it needs to do on that file. This typically means invoking a third-party application using `LrTasks.execute()`.

ADVANCED: If the filter has changed the file format or location on disk using `renditionOptions`, it must now perform the appropriate operations on the file to convert it so that it now satisfies the request as specified in `renditionToSatisfy`.

3. When the processing operation is finished, the filter must report its status on the rendition by calling `renditionToSatisfy:renditionIsDone( success, message )`.

   This is done automatically by the `filterContext:renditions()` iterator if you have not already done so explicitly. The iterator verifies that a file exists at the expected path and signals success or failure accordingly. If the file is missing, it uses a generic "an unknown error occurred" message. If you want to provide a more meaningful message, make an explicit call to `renditionIsDone( false, message )`. Typically, you need only call `renditionIsDone()` on failure.

4. The call to `renditionToSatisfy:renditionIsDone()` allows the downstream consumer's `waitForRender()` call to complete.

5. Meanwhile, the task for this filter continues on and waits for the next rendition.

6. Once all of the filters have finished processing a photo, the `waitForRender()` call in the service's `processRenderedPhotos` loop completes. The service does whatever processing it needs to (in this example, uploading with FTP), and then waits for the next photo to be available.

7. If the "Add to this Catalog" checkbox was selected, Lightroom adds the new photo to the catalog at this point.

Stage 5: Error reporting and clean up

Once the service's `processRenderedPhotos` loop completes, Lightroom takes the following clean-up actions:

- If the photo files were rendered into a temporary folder, Lightroom deletes the folder and its contents.
- If the export was triggered through the Export dialog:
  - Plays the export completion sound, if any.
  - If any photos failed to export, shows an error dialog that summarizes all of the errors encountered while exporting.
  - Creates a temporary "Previous Export" collection with the source photos that were exported.

These behaviors are not available when an export is initiated by calling `LrExportSession` directly.

Final processing of rendered photos

The `processRenderedPhotos()` service entry allows you to control what happens to each exported photo after it is rendered by Lightroom and after all post-process actions have been applied to it. This is

the function that is responsible for transferring the image file to its destination, as defined by your plug-in. The function that you define is launched within a cooperative task that Lightroom provides. You do not need to start your own task to run this function; and in general, you should not need to start another task from within your processing function. (See "Defining function contexts and tasks" on page 20.)

```
processRenderedPhotos = function( functionContext, exportContext )
    ...
end,
```

The parameters `functionContext` and `exportContext` are instances of `LrFunctionContext` and `LrExportContext` respectively, which Lightroom creates and passes to your function.

- Use the function-context object to define cleanup handlers for this call.
- Use the export-context object to gain access to the setting chosen by the user (in `exportContext.propertyTable`), and the list of photos to be exported.
- Use the export-context object to configure the progress indicator to update as photos are displayed. This would look like this example adapted from the Flickr plug-in:

```
local progressScope = exportContext:configureProgress {
   title = nPhotos > 1 and
    LOC( "$$$/Flickr/Publish/Progress=Publishing ^1 photos to Flickr", nPhotos )
    or LOC "$$$/Flickr/Publish/Progress/One=Publishing one photo to Flickr",
}
```

The function that you define typically contains a loop of this form:

```
for i, rendition in exportContext:renditions() do
    -- Wait until Lightroom has finished rendering this photo.
    local success, pathOrMessage = rendition:waitForRender()
    -- Do something with the rendered photo.
    if success then
        -- when success is true, pathOrMessage contains path of rendered file
        local uploadStatus, uploadMessage = uploadToSomewhere( pathOrMessage )
        if not uploadStatus then
            rendition:uploadFailed( uploadMessage )
        end
    else
        -- Report waitForRender failure
        rendition:uploadFailed( pathOrMessage )
        end
    end
end
```

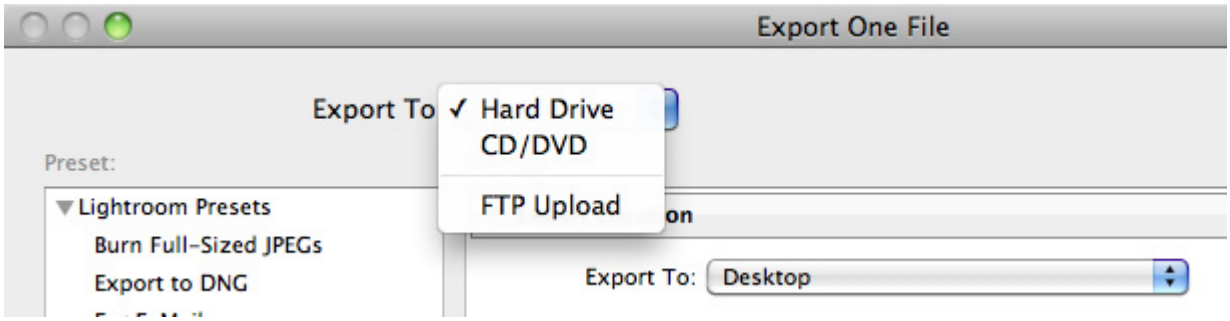
- Lightroom renders the photos in a separate background thread, so it is likely that your upload processing will overlap subsequent rendering operations to some extent.
- The function `exportContext:renditions()` automatically updates the progress indicator in the upper-left corner of the Lightroom catalog window, using a text message that you set by calling `exportContext:configureProgress()`.

Customizing the Export and Publishing Manager dialogs

Customizing the export destination

After Lightroom has completed the rendering of an individual photo for export, and also completed any post-processing of that rendered photo by any selected actions, it completes the export operation by sending the resulting image file to the export-service code, which handles the actual export to the user-selected destination. By default, Lightroom provides export services for a user-selected location in the local file system, or the CD/DVD drive.

The drop-down menu at the top of the Export dialog, labeled "Export To," gives you a choice of destinations:



Your plug-in can provide an export service that allows the user to choose a different destination, such a remote site, and define how the image files are sent to that destination; by FTP upload, for example.

A plug-in that includes an Export Service Provider gives the user a further choice of destinations for the export operation. For example, a plug-in can add a web service such as Flickr as the destination, so that the export operation uploads the selected files to Flickr, using the settings that the user selects in the rest of the dialog. When the plug-in is loaded, the user can select the new destination using the up or down arrow at the right.

If you provide a new export destination, you typically also need to add settings that are meaningful for your customized export operation. Your plug-in can define customizations for other parts of the Export dialog, which are shown when the user selects your Export destination.

- If your upload operation requires more complex user choices, you can add new sections to the Export dialog, with the UI elements that the user will need to make those choices. See "Adding custom sections to the Export or Publishing Manager dialog" on page 56.
- The User Presets list at the left can include presets that you define and include with your plug-in, as well as those defined by Lightroom for export operations; see "Remembering user choices" on page 59.

Adding custom sections to the Export or Publishing Manager dialog

Your plug-in can define one or more sections to be displayed in the Export or Publishing Manager dialog. The custom sections can be shown above or below the Lightroom standard sections for the dialog.

To customize the dialog, define a function that returns a table of sections, defined using `LrView` objects. The function is the value of one of these service entries:

```
sectionsForTopOfDialog = function( viewFactory, propertyTable ) ... end,
sectionsForBottomOfDialog = function( viewFactory, propertyTable ) ... end,
```

> **NOTE:** Similar functions can be defined in a Plug-in Info Provider, to customize the Plug-in Manager dialog, and in an Export Filter Provider, to add a section to the Export dialog when a post-process action is selected. See "Adding custom sections to the Plug-in Manager" on page 34 and "Adding an export post-process action" on page 43.

Lightroom passes your function a factory object which allows you to create the `LrView` objects that define the elements of your sections; that is, UI controls, such as buttons and text, and containers that group the controls and determine the layout. For additional details of the dialog elements you can create with `LrView`, see Chapter 5, "Creating a User Interface for Your Plug-in".

The function that you define here returns a table of tables, where each table defines one dialog section:

```
sectionsForTopOfDialog = function( viewFactory, propertyTable )
    return {
              { ...section entry ... },
              { ...section entry ... },
           ...
           }
    }
end
```

A section entry table defines the contents of an implicit container, which Lightroom creates to hold your view hierarchy.

- Each section entry sets a `title` and `synopsis` for the section; the section is identified by the `title` text on the left, and is collapsible. When in the collapsed state, the `synopsis` text is shown on the right:
- The rest of the table entry creates the UI elements that are shown when the section is expanded. To create the UI elements, use the `LrView` factory passed to your top-level `sectionsFor...` function. This process is explained in more detail in Chapter 5, "Creating a User Interface for Your Plug-in".

When adding sections to the Export dialog, the *propertyTable* parameter for both functions is the property table containing the plug-in and Lightroom-defined export settings. When your plug-in is being used as a publish service provider, the property table contains additional values that tell you about the publishing status; see "Publish Service properties" on page 70.

You can add your own program data values to this table (see "Remembering user choices" on page 59), and create *bindings* between the UI elements and the data values, so that the UI text is dynamically tied to your plug-in data. This is shown in the example below, and explained more fully in "Binding UI values to data values" on page 97.

A custom section example

This sample code creates a section in the destination dialog with two UI controls, an editable text field and a button. These are in a container, a `row` element which controls the placement of its child nodes, but is not drawn on the screen.

The value of the edit field is tied to a data key in the property table. The enabled state of the button is also tied to a data key, so that the button is only enabled when the user types into the edit field, thus setting the data value.

In this example, the `synopsis` text is also dynamic, bound to the same data key as the edit field value. (Currently, you cannot bind the `title` value.) Notice that for `synopsis`, you must specify the bound table explicitly. This is because it is not part of the view hierarchy, where the `propertyTable` is automatically assigned as the default bound table.

All of these concepts and techniques are explained more fully and described in more detail in Chapter 5, "Creating a User Interface for Your Plug-in".

```
sectionsForTopOfDialog = function( viewFactory, propertyTable )
    local LrDialogs = import "LrDialogs" -- get the namespaces we need
    local LrView = import "LrView"
    local bind = LrView.bind -- a local shortcut for the binding function

    propertyTable.user_text = "" -- add program data, no initial value
    propertyTable.buttonEnabled = false -- button starts out disabled

    return { -- create section entries
              { -- first section entry
                 title = "Section title",

                 -- bind the synopsis to a variable for a dynamic description
                 -- of this section that appears when it is collapsed
                 synopsis = bind { key = 'user_text', object = propertyTable },

                 -- create the view hierarchy that appears in the open section
                 viewFactory:row { -- create root container node
                    spacing = viewFactory:control_spacing(), -- default spacing

                    viewFactory:edit_field { -- create edit field
                       value = bind 'user_text', -- bound to property
                       immediate = true, -- update value w/every keystroke
                       validate = function( view, value )
                          if #value > 0 then -- check length of entered text
                             -- any input, enable button
                             propertyTable.buttonEnabled = true
                          else
                             -- no input, disable button
                             propertyTable.buttonEnabled = false
                          end
                          return true, value
                       end
                    },
```

```
                    viewFactory:push_button { -- create button
                        enabled = bind 'buttonEnabled',-- enable when property is set
                        title = "My Button", -- text in button
                        action = function( button )
                            LrDialogs.message( "You typed: ", propertyTable.user_text)
                        end
                    }
                }
            }
        }
    end
```

This code creates this custom section at the top of the Export dialog (when defined by an `LrExportServiceProvider` entry):

Restricting existing service functionality

Your service can restrict the built-in services offered by the Export and Publishing Manager dialogs, by hiding some of the built-in sections that are normally displayed, or by limiting the options offered by the dialog.

A single toggle entry controls whether users can select measurement units:

`hidePrintResolution = Boolean`	When true, the options for sizing in the Image Sizing section are shown only in pixel units; all mention of print units such as inches, centimeters, and pixels-per-inch are hidden. Default is false.

The rest of the service table entries that restrict existing functionality in the dialog come in positive and negative forms; that is, you can list the features to be included, or you can list the features to be excluded. For each such pair, you can provide only one of the entries, not both. If you provide neither, all default elements in that category appear.

For example, you can choose which of the built-in sections to display in the dialog. If you use the positive form, you list the sections to be shown:

```
showSections = { 'fileNaming', 'imageSettings' },
```

This causes the File Naming and Image Sizing sections to be visible, and hides all of the other built-in sections. If you use negative form, you list the sections to be hidden. For example, this hides the Export Location section, and shows all other built-in sections:

```
hideSections = { 'exportLocation' },
```

These are the service-table entry pairs for each type of restriction:

Restriction	Service table entries	Allowed values

File formats	`allowFileFormats = { 'format'[, ...] },`
	`disallowFileFormats = { 'format'[, ...] },`
		These still-photo file formats are recognized:
		`JPEG`
		`PSD`
		`TIFF`
		`DNG`
		`ORIGINAL`
		Video file formats are determined elsewhere.

Color spaces	`allowColorSpaces = { 'colorspace'[, ...] },`
	`disallowColorSpaces = { 'colorspace'[, ...] },`
		These color spaces are recognized:
		`sRGB`
		`AdobeRGB`
		`ProPhotoRGB`

Section display	`showSections = {'section'[, ...] },`
	`hideSections = { 'section'[, ...] },`
		These built-in sections are defined:
		`exportLocation`
		`fileNaming`
		`fileSettings`
		`imageSettings`
		`outputSharpening`
		`metadata`
		`video`
		`watermarking`

- When you hide a section, all of the preset values set in that section are excluded from any presets that the user creates when your plug-in is activated.
- If you hide the `exportLocation` section (the topmost section in the default dialog), Lightroom renders the photos into a temporary location, then deletes that directory and its contents after your `processRenderedPhotos` function terminates.

NOTE: In the Lightroom 1.3 SDK, there was an additional option, `postProcessing`, which was removed in Lightroom 2.0. This section is now only available with the built-in "Export to Files on Disk" service. If specified, it is ignored by Lightroom 2.0 or newer.

Remembering user choices

You typically define properties for your own export or publish operation, whose key names, values, and usage are entirely defined by your plug-in.

Some of these are simply transient or local data (see "Creating observable property tables" on page 100), but some are intended to be *export settings*. Export settings (both plug-in-defined and Lightroom-defined) are persistent properties whose values are saved from one invocation of the Export or Publishing Manager

dialog to the next, and across Lightroom sessions. The most recent settings values are passed to your initialization function when the plug-in is invoked, to various service-script functions (such as `startDialog`, `sectionsForTopOfDialog`, and so on), and to your plug-in's termination function.

To declare persistent properties and their default values, the service definition script for an export service, publish service, or post-process action can return this item:

`exportPresetFields`	A table of keys and associated default values. These are automatically stored in both Lightroom preferences and Export presets. The default values are used only on the first invocation of your plug-in; after that, the previously set values are restored. Although export presets are not available for publish-service plug-ins, this description is used to describe which values from the publish service are remembered from one invocation to the next of the Publish Manager dialog.

For example:

```
exportPresetFields = {
    { key = 'privacy',          default = 'public'   },
    { key = 'privacy_family',   default = false      },
    { key = 'privacy_friends',  default = false      },
    { key = 'safety',           default = 'safe'     },
    { key = 'hideFromPublic',   default = false      },
    { key = 'type',             default = 'photo'    },
    { key = 'addTags',          default = ''         },
}
```

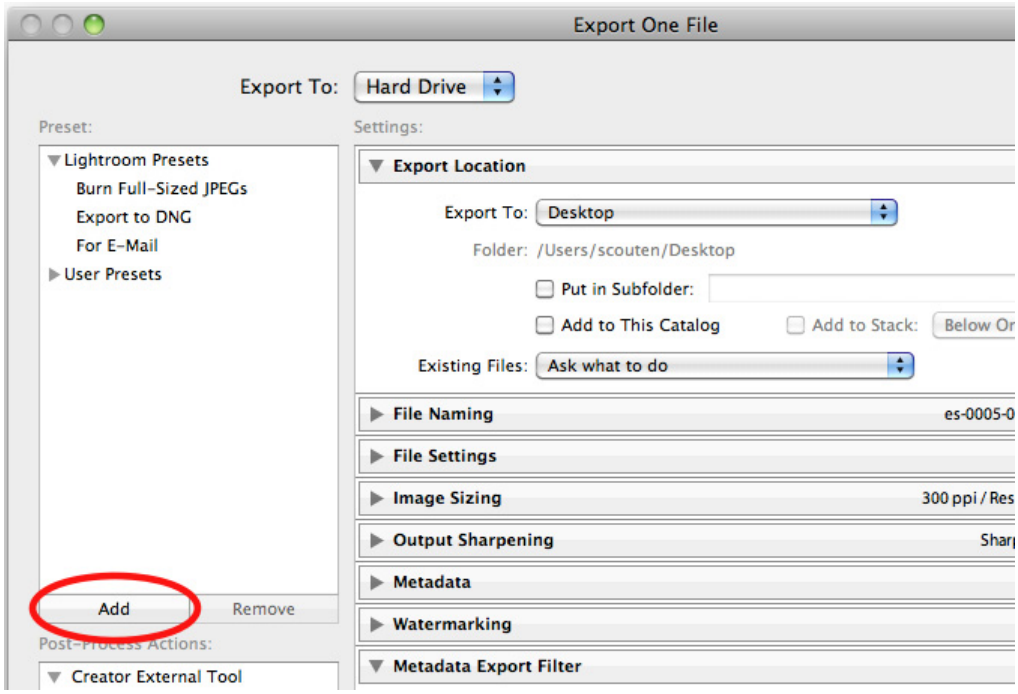
The settings you declare here are automatically saved along with the export settings already defined by Lightroom (see "Lightroom built-in property keys" on page 61). The first time your plug-in is activated, the default value is used to initialize your settings. On the second and subsequent activations, the settings chosen by the user in previous sessions are restored.

Export presets

For an export service, the user can choose to save a particular configuration of settings values as an *export preset*. A preset contains all of the current settings values, including Lightroom-defined fields, plug-in-defined fields, and post-process action-defined fields. It does *not* contain any publish-specific setting that are shown only in the Publishing Manager dialog.

If you wish to create a predefined preset for your export-service plug-in, to be loaded along with your plug-in and appear in the Lightroom Presets list, you must:

1. Make the value selections in the Export dialog.
2. Use **Add** to save a User Preset.



3. Right-click (control-click in Mac OS) on the newly-created Preset to find the preset file. Move the preset file from that folder to the subfolder that you specified using the `builtInPresetsDir` entry for the `LrExportServiceProvider` entry in the `Info.lua` file for your plug-in.

Settings for publish services and post-process actions

Although you can define persistent setting for a publish service using the `exportPresetFields` item, and those values are saved as part of any publish service the user creates in the Publishing Manager dialog, you cannot associate a preset with a publish service. The settings for the Publishing Dialog cannot be loaded from a preset, they can only be set directly by the plug-in (using defaults or a `startDialog` callback) or interactively in the UI.

Similarly, post-process actions cannot have their own presets, but user choices can be remembered if you add an `exportPresetFields` entry to the action's `info.lua` file; see "Defining a post-process action" on page 47. If the user has chosen to activate the post-process action for a particular export operation, user choices for that action are captured in any preset created for that export service.

Lightroom built-in property keys

Your export operation makes use of the settings that you have defined for your plug-in (see "Remembering user choices" on page 59), and can also check any of the Lightroom-defined properties, as

set in the Export dialog. These are all in the property table passed to your initialization function (see "Initialization and termination functions for services" on page 40).

The Lightroom built-in keys apply to specific sections of the Export dialog and Library module window, as follows:

Export Location section properties

`LR_export_destinationType`	String. The value of the "Export To:" pop-up. One of:
	`chooseLater`
	`desktop`
	`documents`
	`home`
	`pictures`
	`sourceFolder`
	`specificFolder`
	`tempFolder`

`LR_export_destinationPathPrefix`	String. Destination folder.
`LR_export_useSubfolder`	Boolean. True when the "Put in Subfolder" option is checked, false when it is unchecked. Cannot be used if destination type is `tempFolder`.
`LR_export_destinationPathSuffix`	String. Subfolder name. Valid only if use-subfolder is true.
`LR_reimportExportedPhoto`	Boolean. True when the "Add to This Catalog" option is checked, false when it is unchecked. Cannot be used if destination type is `tempFolder`.
`LR_reimport_stackWithOriginal`	Boolean. True when the "Stack With Original " option is checked, false when it is unchecked. This is only meaningful if destination type is `sourceFolder` and use-subfolder is false. Ignored otherwise.

`LR_collisionHandling`	String. The value of the "Existing Files:" pop-up. One of:
	`ask`
	`rename`
	`overwrite`
	`skip`

File Naming section properties

`LR_extensionCase`	String. The case to use for the file extension. One of:
	One of
	`uppercase`
	`lowercase`

LR_initialSequenceNumber	Number. The initial value for the sequence number, when files are renamed on export using a sequence number.
LR_renamingTokensOn	Boolean. When true, the files are renamed on export as described below.
LR_tokenCustomString	String. A naming template to use when files are renamed on export.
LR_tokens	String. A file-naming pattern. The string is used as the file name, but can contain a pattern within double curly braces which is replaced by information about the photo being exported. For example, the string {{image_name}} is replaced by the file name. See allowed token patterns below.

Token patterns

This is the complete list of patterns that are allowed in the LR_tokens string, as presented in the pop-up menu in the File Naming Template dialog:

Image Name and Custom patterns

Pattern	Replaced by
{{image_name}}	File name
{{image_filename_number_suffix}}	File number suffix
{{image_folder}}	Folder name
{{image_originalName}}	Original file name
{{image_originalName_number_suffix}}	Original number suffix
{{copy_name}}	Copy name
{{custom_token}}	Custom Text (the value of LR_tokenCustomString)

Sequence and Date patterns

Pattern	Replaced by
{{naming_sequenceNumber_1Digit}}	Sequence # (1)
{{naming_sequenceNumber_2Digits}}	Sequence # (01)
{{naming_sequenceNumber_3Digits}}	Sequence # (001)
{{naming_sequenceNumber_4Digits}}	Sequence # (0001)
{{naming_sequenceNumber_5Digits}}	Sequence # (00001)
{{naming_operationSequence_1Digit}}	Image # (1)
{{naming_operationSequence_2Digits}}	Image # (01)

{{naming_operationSequence_3Digits}}	Image # (001)
{{naming_operationSequence_4Digits}}	Image # (0001)
{{naming_operationSequence_5Digits}}	Image # (00001)
{{naming_sequenceTotal_1Digit}}	Total # (1)
{{naming_sequenceTotal_2Digits}}	Total # (01)
{{naming_sequenceTotal_3Digits}}	Total # (001)
{{naming_sequenceTotal_4Digits}}	Total # (0001)
{{naming_sequenceTotal_5Digits}}	Total # (00001)
{{date_LocalEncoding}}	Date (Month DD, YYYY)
{{date_YYYYMMDD}}	Date (YYYYMMDD)
{{date_YYMMDD}}	Date (YYMMDD)
{{date_YYYY}}	Date (YYYY)
{{date_YY}}	Date (YY)
{{date_Month}}	Date (Month)
{{date_Mon}}	Date (Mon)
{{date_MM}}	Date (MM)
{{date_DD}}	Date (DD)
{{date_Julian}}	Julian Day of the Year
{{date_Hour}}	Hour
{{date_Minute}}	Minute
{{date_Second}}	Second

Metadata patterns

Pattern	Replaced by
{{com.adobe.title}}	Title
{{com.adobe.caption}}	Caption
{{com.adobe.copyright}}	Copyright
{{com.adobe.keywords}}	Keywords
{{com.adobe.creator}}	Creator
{{com.adobe.creatorJobTitle}}	Creator Job Title
{{com.adobe.creatorAddress}}	Creator Address
{{com.adobe.creatorCity}}	Creator City

Pattern	Replaced by
{{com.adobe.creatorState}}	Creator State / Province
{{com.adobe.creatorZip}}	Creator Postal Code
{{com.adobe.creatorCountry}}	Creator Country
{{com.adobe.creatorWorkPhone}}	Creator Phone
{{com.adobe.creatorWorkEmail}}	Creator E-Mail
{{com.adobe.creatorWorkWebsite}}	Creator Website
{{com.adobe.descriptionWriter}}	Description Writer
{{com.adobe.iptcSubjectCode}}	IPTC Subject Code
{{com.adobe.intellectualGenre}}	Intellectual Genre
{{com.adobe.scene}}	Scene
{{com.adobe.location}}	Location
{{com.adobe.isoCountryCode}}	ISO Country Code
{{com.adobe.headline}}	Headline
{{com.adobe.city}}	City
{{com.adobe.state}}	State / Province
{{com.adobe.country}}	Country
{{com.adobe.jobIdentifier}}	Job Identifier
{{com.adobe.instructions}}	Instructions
{{com.adobe.provider}}	Provider
{{com.adobe.source}}	Source
{{com.adobe.rightsUsageTerms}}	Rights Usage Terms
{{com.adobe.copyrightInfoURL}}	Copyright Info URL
{{com.adobe.imageFileDimensions}}	Dimensions
{{com.adobe.imageCroppedDimensions}}	Cropped
{{com.adobe.exposure}}	Exposure
{{com.adobe.focalLength}}	Focal Length
{{com.adobe.focalLength35mm}}	Focal Length 35mm
{{com.adobe.brightnessValue}}	Brightness Value
{{com.adobe.exposureBiasValue}}	Exposure Bias
{{com.adobe.subjectDistance}}	Subject Distance
{{com.adobe.ISOSpeedRating}}	ISO Speed Rating

Pattern	Replaced by
{{com.adobe.flash}}	Flash
{{com.adobe.exposureProgram}}	Exposure Program
{{com.adobe.meteringMode}}	Metering Mode
{{com.adobe.make}}	Make
{{com.adobe.model}}	Model
{{com.adobe.serialNumber}}	Serial Number
{{com.adobe.artist}}	Artist
{{com.adobe.software}}	Software
{{com.adobe.lens}}	Lens
{{com.adobe.GPS}}	GPS
{{com.adobe.GPSAltitude}}	Altitude
{{com.adobe.rating.string}}	Rating
{{com.adobe.colorLabels.string}}	Label

File Settings section properties

LR_format	String. File format for still photos. One of: JPEG PSD TIFF DNG ORIGINAL
LR_export_colorSpace	String. Color space. One of sRGB AdobeRGB ProPhotoRGB
LR_export_bitDepth	Number. Bit depth for TIFF or PSD. Ignored for other formats. One of: 8 16
The following apply only if LR_format **=** JPEG	
LR_jpeg_quality	Number. JPEG quality [0..1]. A percentage value where 1 is the best quality.

LR_jpeg_useLimitSize	Boolean. True when the "Limit File Size To:" option is checked, false when it is unchecked. When true, Lightroom chooses the quality setting that best matches the desired file size.
LR_jpeg_limitSize	Number. When file size is limited, the target size in kilobytes.
The following apply only if LR_format **=** TIFF	
LRtiff_compressionMethod	String. TIFF compression method. One of: compressionMethod_None compressionMethod_LZW compressionMethod_ZIP
The following apply only if LR_format **=** DNG	
LR_DNG_previewSize	String. JPEG preview size. One of: none medium large
LR_DNG_compatability	Number. The oldest version of Adobe Camera Raw/Photoshop that can read these files. One of: 33816576 (Camera Raw 2.4 and later) 67174400 (Camera Raw 4.1 and later) 67502080 (Camera Raw 4.6 and later) 84148224 (Camera Raw 5.4 and later)
LR_DNG_conversionMethod	String. Image conversion method. One of: preserveRAW convertToLinear
LR_DNG_embedRAW	Boolean. True to embed original raw file.

Image Sizing section properties

LR_size_doConstrain	Boolean. True to constrain maximum size.
LR_size_doNotEnlarge	Boolean. True to prevent enlargement of the image size.
LR_size_maxHeight	Number. Height constraint in units specified by LR_size_units.
LR_size_maxWidth	Number. Width constraint in units specified by LR_size_units.
LR_size_megapixels	Number. When resize type is megapixels, the number of megapixels.

LR_size_resizeType	String. The resize method. One of: wh (width and height) dimensions longEdge shortEdge megapixels For the long-edge and short-edge methods, the value is constrained by LR_size_maxHeight.
LR_size_resolution	Number. Resolution, in units specified by LR_size_resolutionUnits.
LR_size_resolutionUnits	String. Resolution units. One of: inch cm
LR_size_units	String. Size constraint units. One of inch cm pixels

Output Sharpening section properties

LR_outputSharpeningOn	Boolean. True when the "Sharpen For" option is checked, false when it is unchecked.
LR_outputSharpeningMedia	String. The destination media for the sharpening operation. One of: screen matte glossy
LR_outputSharpeningLevel	Number. The amount of sharpening. One of 1 (low), 2 (medium), or 3 (high).

Metadata section properties

LR_minimizeEmbeddedMetadata	Boolean. True when the "Minimize Embedded Metadata" option is checked, false when it is unchecked. This property is superseded by LR_embeddedMetadataOption.
LR_metadata_keywordOptions	String. Corresponds to the state of the “Keywords as Lightroom Hierarchy” checkbox. Value can be lightroomHierarchical (checked) or flat (unchecked). Ignored when LR_minimizeEmbeddedMetadata is true.

`LR_removeLocationMetadata`	Boolean. True when "Remove Location Info" box is checked, false when it is unchecked. When true, Lightroom strips all GPS location metadata from the photo on export.
`LR_embeddedMetadataOption`	String. Corresponds to the state of the "Include" popup menu. Value can be `copyrightOnly`, `copyrightAndContactOnly`, `allExceptCameraInfo`, or `all`.

Video section properties

`LR_includeVideoFiles`	Boolean. True when the "Include Video Files" option is checked, false when it is unchecked. When true, export any video files that are selected. When false, omit those files and show an error message when the export operation is completed. Considered only when the plug-in sets `canExportVideo = true` in its service description script, to indicate that it can process video. Otherwise, this is ignored and the corresponding checkbox is not shown.

NOTE: To dynamically set the format and preset to use for video export, use the method `LrExportSettings.applyVideoExportPreset()`. See the API Reference for details.

Watermarking section properties

`LR_useWatermark`	Boolean. True to enable watermarking on export.
`LR_watermarking_id`	When watermarking is enabled, the unique identifier of the watermark to use. Lightroom assigns this ID when the user creates a new watermarking preset. You cannot modify it, or create a watermarking preset programmatically.

Post-Processing Filter section properties

`LR_exportFiltersFromThisPlugin`	Table with a key that is the ID of each enabled filter, and a corresponding value that is the index of that filter in the overall filter stack. For example:
	
	`LR_exportFiltersFromThisPlugin.myFirstFilter.1`
	`LR_exportFiltersFromThisPlugin.mySecondFilter.2`
	
	If your plug-in contains multiple filters, this tells you which ones were selected by the user.

General export properties

This property allows your plug-in to disable the Export button in the Export dialog, or the Save button in the Publishing Manager dialog.

`LR_cantExportBecause`	String, optional. If present, describes why an export cannot be initiated. Appears in the bottom of the dialog near the disabled (dimmed) Export or Save button. When the error condition is fixed, your plug-in should set this value to nil. In response to that change, Lightroom enables the Export or Save button (if no other error conditions exist). Ignored in versions earlier than 2.0.

Publish Service properties

These properties are part of the settings table passed to the following callbacks, when they are defined as part of a publish service provider:

```
startDialog
endDialog
sectionsForTopOfDialog
sectionsForBottomOfDialog
```

`LR_publish_connectionName`	String. The descriptive name of a connection, as assigned by the user in the Publishing Manager dialog.
`LR_isExportForPublish`	Boolean, read-only. True when the current export operation is part of a publish service, either currently running or being edited in the Publishing Manager.
`LR_editingExistingPublishConnection`	Boolean. True when the user is editing an existing publish service.
`LR_publishService`	`LrPublishService`. When editing an existing publish service, the service object.

4 Working with Metadata

- A plug-in can define customized metadata fields for photos by declaring a Metadata Provider in the information file, and defining type and version information for each field. See "Adding custom metadata" on page 71.
- A plug-in can specify metadata *tagsets* that affect the display of custom metadata fields in the Library module's Metadata panel. See "Adding custom metadata tagsets" on page 76.
- Plug-ins can define complex searches that find photos according to metadata values. See "Searching for photos by metadata values" on page 78.

Adding custom metadata

Your plug-in can define custom metadata fields for photos that are imported into Lightroom. These fields can be visible and even editable in Lightroom's Metadata panel, or can be invisible and used to store private data.

Declaring a Metadata Provider

Like an Export Service or Export Filter Provider, you declare a Metadata Provider in the information file (`Info.lua`) for your plug-in. See "Writing standard plug-ins for Lightroom" on page 23.

To declare a Metadata Provider, include an `LrMetadataProvider` entry in the `Info.lua` file; for example:

```
return {
    LrSdkVersion = 5.0,
    LrToolkitIdentifier = 'com.adobe.lightroom.metadata.sample',
    LrPluginName = LOC "$$$/CustomMetadata/PluginName=Metadata Sample",

    LrMetadataProvider = 'SampleMetadataDefinition.lua',
}
```

The information file that declares a Metadata Provider can also declare metadata tagsets (see "Adding custom metadata tagsets" on page 76), export services and/or filters, but need not do so.

Limitations of custom metadata in this release

In the current implementation, custom metadata defined by a plug-in has these limitations, which will be addressed in future versions of the Lightroom SDK:

- Values stored in custom metadata fields are stored only in Lightroom's database. In the current release, a plug-in cannot link custom metadata fields to XMP values or save them with the image file.
- A plug-in cannot specify complex data types. You can define simple fields per photo, but you cannot define a whole spreadsheet per photo.

Defining metadata fields

The script for your Metadata Provider defines specific metadata fields. The metadata definition script returns a table that describes the fields to be added to Lightroom's metadata schema. It contains the following fields:

`metadataFieldsForPhotos`	table	Required. Defines new data fields to be stored for each photo. The table is an array of field definitions. Each entry in the `metadataFieldsForPhotos` array describes a single field which can be associated with photos in the catalog. Each field can hold only one value per photo. See "Metadata field entries" on page 73.
`schemaVersion`	number	Required. Allows for versioning of the property definition schema. Typically this number starts at 1 and you increment it whenever you release a new version of the schema or need to run the update function.
`updateFromEarlierSchemaVersion`	function	Optional. Allows your plug-in to update data when a new schema version is in place. The function takes three parameters; *`catalog`*, *`previousSchemaVersion`*, and *progressScope*. When the plug-in is first installed, *`previousSchemaVersion`* is nil. See example below.
`noAutoUpdate`	Boolean	Optional. When false (the default), Lightroom attempts to automatically update metadata from old field definitions to new. If you prefer to handle this in your own plug-in (using the `updateFromEarlierSchemaVersion` callback), set this to true.

Metadata field entries

Each of the entries in the `metadataFieldsForPhotos` array is a table that describes one metadata field; each metadata field describes a photo in the catalog. Each field can have only one value per photo. The following entries are recognized within each table:

`id`	string	Required. A unique identifier that allows a plug-in to access this field. The name must conform to the same naming conventions as Lua variables; that is, it must start with a letter, followed by letters or numbers, case is significant.
`version`	number	Optional. If present, defines a version number specifically for this field, distinct from the version number defined by `schemaVersion` in the outer metadata definition script. If you make a change to a field definition that is incompatible with the previous definition (for example, changing the value of `searchable`), you must bump the field's version number. A migration script can search for photos that contain the old version of the field and manually migrate values.
`title`	string	Optional. If this field is displayed in the Metadata panel, this is the localizable display name. This name should be relatively short, since space in the Metadata panel is at a premium. A name longer than about 100 pixels is likely to be truncated on display; however, the full text is shown in the tooltip when the cursor hovers over the name. If this item is omitted, the field does not appear in the Metadata panel. This can be useful for storing private, per-image plug-in information, such as the image's ID at an on-line service that is the export destination, or other cross-reference information.
`dataType`	string	Optional. If this field is present, Lightroom disallows any other data type from being stored in this field. Nil is always permitted. You cannot require that a field have a value. The value is one of these strings: **`string`** — The field value must have a string value. **`enum`** — The field value must have one of the allowed values specified in the `values` entry. In the Metadata panel, allowed values are shown as a pop-up menu for the field. **`url`** — The field value must have a string value. In the Metadata panel, the text field is acccompanied by a button that treats the text value as a URL, opening it in the user's preferred web browser.

values	table	Required when dataType = "enum", otherwise disallowed. An array of allowed values. Each entry in the array is a table that must contain a *value* and a *title*. The title is shown in the popup menu; the corresponding value (which must be a string, number, or Boolean, or nil) is written to the database. The values table can have only one entry where value = nil. If such an entry is present, the corresponding label is used when no value has been assigned to this property for a photo. The values table can also have an entry allowPluginToSetOtherValues = true. • If present, your plug-in can store values outside of the enumerated values in this field. • If not, an attempt to set such a value triggers a Lua error and does not change the value stored in the database.
readOnly	Boolean	Optional. Use only when title is provided. When true, the field is visible in the Metadata panel, but not editable by the user. The value can still be set programmatically, using LrPhoto:setPropertyForPlugin().
searchable	Boolean	Optional. Use only when title is provided. When true, this field is stored in a separate table and indexed for faster searching; this also means that the field can be chosen by a user as a search criterion for smart collections. Strings stored in this field must not exceed 511 bytes. Default is false.
browsable	Boolean	Optional. Use only when title is provided and searchable is true. When true, this field can be used as a filter in the Library metadata browser.

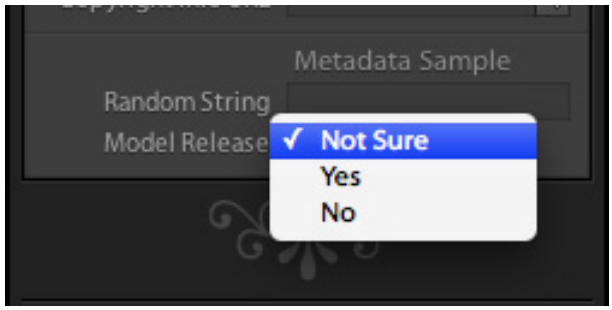
Custom metadata example

This sample Metadata Provider script defines three metadata fields of representative types.

```
return {
    metadataFieldsForPhotos = {

        {
            id = 'siteId',
            -- This field is not available in the metadata browser because
            -- it does not have a title field. You might use a field like this
            -- to store a photo ID from an external database or web service.
        },

        {
            id = 'randomString',
            title = LOC "$$$/Sample/Fields/RandomString=Random String",
            dataType = 'string', -- Specifies the data type for this field.
        },

        {
            id = 'modelRelease',
            title = LOC "$$$/Sample/Fields/ModelRelease=Model Release",
            dataType = 'enum',
            values = {
                {
                    value = nil,
```

```
                    title = LOC "$$$/Sample/Fields/ModelRelease/NotSure=Not Sure",
                },
                {
                    value = 'yes',
                    title = LOC "$$$/Sample/Fields/ModelRelease/Yes=Yes",
                },
                {
                    value = 'no',
                    title = LOC "$$$/Sample/Fields/ModelRelease/No=No",
                },
                -- optional: allowPluginToSetOtherValues = true,
            },
        },

    },

    schemaVersion = 1,
        -- must be a number, preferably a positive integer

    updateFromEarlierSchemaVersion = function( catalog, previousSchemaVersion,
progressScope )

    -- When the plug-in is first installed, previousSchemaVersion is nil.

    -- As of Lightroom version 3.0, a progress-scope variable is available; you can
    -- use it to signal progress for your upgrader function.
    -- Note: This function is called from within a  catalog:withPrivateWriteAccessDo
    -- block. You should not call any of the with___Do functions yourself.

        catalog:assertHasPrivateWriteAccess(
               "SampleMetadataDefinition.updateFromEarlierSchemaVersion" )
        local myPluginId = 'com.adobe.lightroom.metadata.sample'
        if previousSchemaVersion == 1 then
            local photosToMigrate = catalog:findPhotosWithProperty( myPluginId,'siteId')
            -- optional: can add property version number here
            for i, photo in ipairs( photosToMigrate ) do
                local oldSiteId = photo:getPropertyForPlugin( myPluginId, 'siteId' )
                -- add property version here if used above
                local newSiteId = "new:" .. oldSiteId
                -- replace this with whatever data transformation you need to do
                photo:setPropertyForPlugin( _PLUGIN, 'siteId', newSiteId )
            end
        elseif previousSchemaVersion == 2 then
              -- and so on...
        end
    end,
}
```

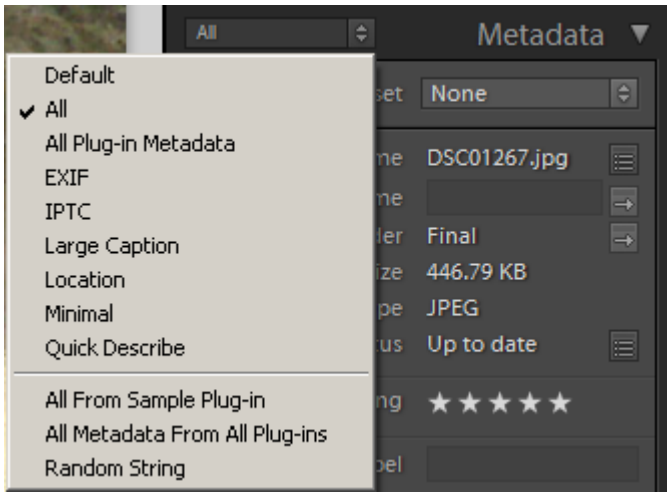
This is how these fields appear in the Metadata panel, when the user has chosen a metadata tagset that contains them, or one of the default metadata tagsets "All" or "All Plug-in Metadata" (see "Adding custom metadata tagsets" on page 76). The user-visible custom fields from plug-ins are shown after all of the built-in metadata fields.



- Notice that the field "siteId" does not appear in the panel because no `title` is defined for it; it is an invisible field, internal to the plug-in.
- The field "randomString" appears with the localized `title` value, "Random String", as the display label. Because it is a plain string value, it appears as an editable text field.
- The field "modelRelease" also appears with the `title` value, "Model Release", as the display label. Because it is an enumerated value, clicking it pops up a menu of the allowed values, each shown using its own localized `title` value as the display string.

Adding custom metadata tagsets

The drop-down menu at the top left of the Metadata panel allows users to filter what is shown in the panel, by selecting a *metadata tagset* to be displayed. A metadata tagset is a predefined set of metadata fields. When you select a tagset, the Metadata panel displays only the fields included in that set.



There are predefined tagsets, and you can also create your own. Your plug-in can define a named metadata tagset, which can include fields defined by your plug-in, by other plug-ins, or by Lightroom.

This is the `Info.lua` file for a minimal plug-in that defines a tagset:

```
return {
    LrSdkVersion = 5.0,
    LrToolkitIdentifier = 'com.adobe.lightroom.metadata.sample',
    LrPluginName = LOC "$$$/CustomMetadata/PluginName=Metadata Sample",
    LrMetadataTagsetFactory = 'SampleTagset.lua',
}
```

The metadata-tagset provider can appear in the same plug-in with export-service and export-filter providers, and with simple Metadata Providers.

The metadata-tagset provider is a Lua file that returns a tagset definition. You can use the `LrMetadataTagsetFactory` entry to specify more than one such file in a single plug-in. For example:

```
LrMetadataTagsetFactory = { 'Tagset1.lua', 'Tagset2.lua', 'Tagset3.lua' },
```

Defining a metadata tagset

Each tagset definition file must return a table listing the fields that should appear in the Metadata panel when that tagset is selected. The file can return an array of such tables to define multiple tagsets.

The table contains these entries:

`id`	string	Required. An identifier for this tagset that is unique within this plug-in. The name must conform to the same naming conventions as Lua variables; that is, it must start with a letter, followed by letters or numbers. Case is significant.
`title`	string	Required. The localizable display name of the tagset, which appears in the popup menu for the Metadata panel.
`items`	table	Required. An array of metadata fields that appear in this tagset, in order of appearance.

Each entry in the `items` array identifies a field to be included in the Metadata menu. It can be a simple string specifying the field name, or an array that specifies the field name and additional information about that field:

fieldname	string	The first element in the array is the unique identifying name of the field, or one of the special values described below.
`label`	string	Optional. When the field name is the special value '`com.adobe.label`', this is the localizable string to use as the section label.

Certain special values are defined for field names. See the API Reference documentation for metadata plug-in-defined functions, and the metadata sample plug-in.

- You can include all visible metadata from a plug-in by specifying the field name with the wild-card character "`*`"; for example, "`com.mycompany.uploader.*`". The visible fields are included in the sequence in which they are defined in the definition script. The fields for each plug-in are preceded by a dividing line and the plug-in's name.

  If you refer to a plug-in that is missing or that defines no visible metadata, it is not an error; the block and separator for that plug-in are simply not displayed.

- You can include all visible metadata from a plug-in by specifying the special field name "`com.adobe.allPluginMetadata`". This is the field name used by the built-in "All Plug-in Metadata" preset.

- The special name '`com.adobe.separator`' inserts a dividing line in the Metadata panel before the first field from this plug-in.

- The special name '`com.adobe.label`' inserts a section label in the Metadata panel, specified by a `label` entry in an array with this name. A label is typically used below a separator.

Custom metadata tagset example

This sample metadata-tagset provider script defines a set of metadata fields.

```
return {
    title = LOC "$$$/SampleTagset/Title=Sample Tagset from Plug-in",
    id = 'sampleTagset',

    items = {
        'com.adobe.filename',
        'com.adobe.copyname',
        'com.adobe.folder',

        'com.adobe.separator',

        'com.adobe.title',
        { 'com.adobe.caption', height_in_lines = 3 },

        'com.adobe.separator',
        { 'com.adobe.label', label = LOC "$$$/Metadata/SampleLabel=Section Label" },

        'com.adobe.dateCreated',
        'com.adobe.location',
        'com.adobe.city',
        'com.adobe.state',
        'com.adobe.country',
        'com.adobe.isoCountryCode',

        'com.adobe.GPS',
        'com.adobe.GPSAltitude',

        'com.adobe.lightroom.metadata.sample.randomString',

    },
}
```

Searching for photos by metadata values

The `LrCatalog` object provides a function, `findPhotos()`, that allows you to search through the catalog for photos with particular metadata values. You pass this function a *search descriptor* to define the search, which is a table containing a metadata field (the search *criteria*), a matching *operation* (which depends on the datatype of the field), and a *value* to match against.

This function must be used within a background task. For example:

```
import "LrTasks".startAsyncTask( function()

    local catalog = import "LrApplication".activeCatalog()

    local photos = catalog:findPhotos {
        searchDesc = {
            criteria = "rating",
            operation = ">",
            value = 3,
```

```
        },
    }

    for _, photo in ipairs( photos ) do
        -- do somethingwith path
    end

end )
```

This simple usage is straightforward, although the function allows many matching operations, depending on the datatype of the metadata field to be considered.

criteria The allowed values for `criteria` correspond to choices in the Edit Smart Collection dialog:

`rating`	(number)
`pick`	(enum) Value must be one of: 1 (flagged), 0 (unflagged), -1 (rejected)
`labelColor`	(enum) Value must be one of: 1 (red), 2 (yellow), 3 (green), 4 (blue), 5 (purple), "custom" (any label not currently assigned to a color), "none"
`labelText`	(string, can be empty) User-assigned name of color label
`folder`	(string) Name of folder, including all parent folders shown in the Folders panel
`collection`	(string) Name of any collection containing this photo
`all`	(string) Any searchable text
`filename`	(string)
`copyname`	(string, can be empty) Copy Name assigned in Metadata panel
`fileFormat`	(enum) Value must be one of: "DNG", "RAW", "JPG", "TIFF", "PSD"
`metadata`	(string) Any searchable metadata
`title`	(string, can be empty)
`caption`	(string, can be empty)
`keywords`	(string, plural, can be empty)
`iptc`	(string) Any IPTC metadata; that is, any text in a field that is indexed by Lightroom.
`exif`	(string) Any EXIF metadata; that is, any text in a field that is indexed by Lightroom.
`captureTime`	(date)
`touchTime`	(date) Edit Date
`camera`	(string, with exact match)
`cameraSN`	(string, with exact match) Camera Serial Number
`lens`	(string, with exact match)
`isoSpeedRating`	(number)
`hasGPSData`	(Boolean)

`country`	(string, with exact match)
`state`	(string, with exact match)
`city`	(string, with exact match)
`location`	(string, with exact match)
`creator`	(string, with exact match)
`jobIdentifier`	(string, with exact match)
`copyrightState`	(enum) Value must be one of: true (Boolean, copyrighted), false (Boolean, public domain), "unknown"
`hasAdjustments`	(Boolean)
`developPreset`	(enum) Value must be one of: "default", "specified", "custom"
`treatment`	(enum) Value must be one of: "grayscale", "color"
`cropped`	(Boolean)
`aspectRatio`	(enum) Value must be one of: "portrait", "landscape", "square"

You can search plug-in defined fields, using these special `criteria` values:

`"allPluginMetadata"`	Any searchable plug-in-defined metadata.
`sdktext:`*`plugin_id.field_name`*	A specific, searchable, plug-in-defined field (with datatype text or enum).
`sdktext:`*`plugin_id`*`.*`	All searchable fields defined by a specific plug-in (with datatype text or enum).

operation The allowed values for `operation` depend on value type of the criteria field, and also correspond to selectable values the Edit Smart Collections dialog.

- For string values, one of:

`any`	"contains"
`all`	"contains all"
`words`	"contains words"
`noneOf`	"does not contain"
`beginsWith`	"starts with"
`endsWith`	"ends with"
`empty`	"are empty", only valid for items that can be empty,
`notEmpty`	"are not empty", only valid for items that can be empty,

==	"is", only valid for items that can have an exact match,
!=	"is not", only valid for items that can have an exact match

- For Boolean values, one of "isTrue", "isFalse"
- For enumerated values, one of == (is), != (is not)
- For number and rating values, one of:

==	"is"
!=	"is not"
>	is greater than"
<	"is less than"
>=	"is greater than or equal to"
<=	"is less than or equal to"
in	"is in range" End value of range specified in value2 parameter.

- For date values, one of:

==	"is"
!=	"is not"
>	"is after"
<	"is before"
inLast notInLast	"is in the last" "is not in the last" With unit specified in value_unit parameter, one of: hours days weeks months years
in	"is in the range" End value of range specified in value2 parameter.
today	"is today"
yesterday	"is yesterday"
thisWeek	"is in this week"

`thisMonth`	"is in this month"
`thisYear`	"is in this year"

value The value to match against must be of the type indicated for the criteria. Additional parameters `value2` and `value_unit` are used with specific types and operations, as mentioned above.

Combining search criteria

You can create a more complex search descriptor by using a `combine` entry to specify how to combine the results of several criterion tables:

```
import "LrTasks".startAsyncTask( function()

    local catalog = import "LrApplication".activeCatalog()

    local photos = catalog:findPhotos {
        searchDesc = {
            combine = "intersect",
            {
                criteria = "rating",
                operation = ">",
                value = 3,
            },
            {
                criteria = "captureDate",
                operation = ">",
                value = "2007-01-01",
            }
        },
    }


    for _, photo in ipairs( photos ) do
        -- do something with path
    end

end )
```

There are three ways to combine criteria:

`combine = "union"`	Any of the criteria match.
`combine = "intersect"`	All of the criteria match.
`combine = "exclude"`	None of the criteria match.

A `combine` entry is followed by an array of elements to be combined. This array can contain nested `combine` entries, so the search can become quite complex. For example:

```
    {
            combine = "union",
            {
                    combine = "intersect",
                    {
                            criteria = "rating",
```

```
                operation = ">=",
                value = 1,
            },
            {
                criteria = "labelColor",
                operation = "==",
                value = 1,
            },
        },
        {
            criteria = "rating",
            operation = "==",
            value = 5,
        },
}
```

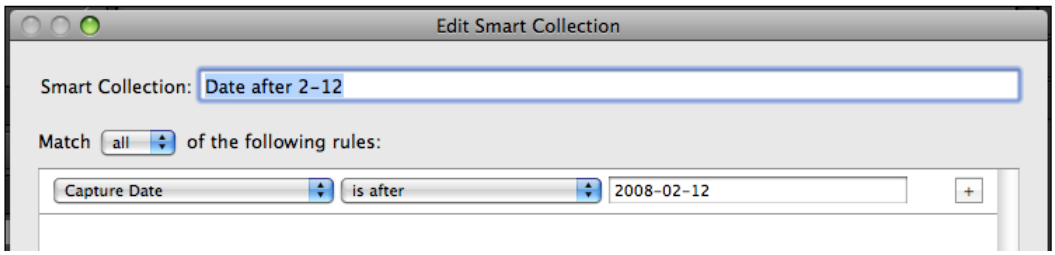
This renders the following statement of Boolean logic:

```
photos where ( rating >= 1 AND labelColor == 1 ) OR ( rating == 5 )
```

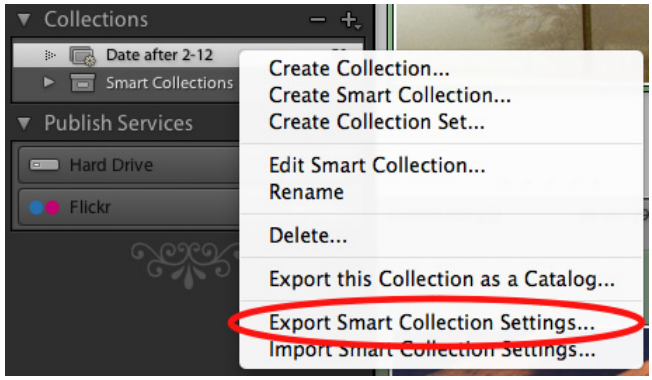
Creating searches interactively

If you are unsure how to construct a particular search, you can make Lightroom build it for you. To do this:

1. Construct the search as a Smart Collection in Lightroom:



2. Right click the resulting collection in the Collections panel and choose "Export Smart Collection Settings."



3. Open the resulting `.lrsmcol` file in a text editor.

4. Select and copy the `value` entry:

```
s = {
    id = "C5510D75-282B-40E4-91BD-18C9F0541553",
    internalName = "Date after 2-12",
    title = "Date after 2-12",
    type = "LibrarySmartCollection",
    value = {
        {
            criteria = "captureTime",
            operation = ">",
            value = "2008-02-12",
        },
        combine = "intersect",
    },
    version = 0,
}
```

5. Edit the resulting code to change `value` to `searchDesc`, and include it in your call to `findPhotos()`.

6. Make any other appropriate changes in the code. In this example, for instance, you would not need the `combine` element. If you remove it, you can also promote the parameter table to the top level in `searchDesc`:

```
searchDesc = {
    criteria = "captureTime",
    operation = ">",
    value = "2008-02-12",
}
```

5 Creating a User Interface for Your Plug-in

You can define a user interface to your plug-in with these tools:

- Your plug-in can define one or more custom sections to be displayed in the Plug-in Manager dialog or Export dialog, above and/or below the Lightroom standard sections. The custom sections are displayed when the user chooses your export destination. You define the UI elements of a custom section using `LrView` objects; see "Adding custom dialog views" on page 85.
- You can call the functions of the `LrDialog` namespace to display messages, prompts, and errors to users in predefined dialogs. See "Displaying predefined dialog boxes" on page 86.
- You can use the functions in the `LrDialog` and `LrView` namespaces to create your own dialog boxes. You can display them when users choose your custom menu items, invoke them from tasks, or invoke them in response to selections in controls you have added to the Export dialog. See "Creating custom dialog boxes" on page 87.

The `LrView` class models a node tree, where each node is a UI element, represented by a specific type of `LrView` object. A node can be a *container* or parent of other nodes, or a *control*, an individual UI element such as a checkbox, which displays a value and can allow user input. Containers and controls are arranged in a node tree, or *view hierarchy*. A view hierarchy has a top-level container node, additional child containers if needed, and leaf nodes that are the controls.

The `LrView` namespace and class provides a set of interface elements, with functionality to lay out and localize the display, and a binding mechanism that lets you tie the displayed values to your plug-in data and settings.

- "User interface elements" on page 87 introduces the UI elements you can create with `LrView`.
- "Binding UI values to data values" on page 97 explains the binding mechanism, with examples of how to create various relationships between your data and your display.
- """Determining layout" on page 110 explains the placement options and gives examples of various layout techniques.

Adding custom dialog views

You can create custom sections to be displayed in the Plug-in Manager or Export dialog using these service-script entries:

```
sectionsForTopOfDialog = function( viewFactory, propertyTable ) ... end,
sectionsForBottomOfDialog = function( viewFactory, propertyTable ) ... end,
```

- The function that you define here is slightly different for the two dialogs; see "Adding custom sections to the Plug-in Manager" on page 34 and "Adding custom sections to the Export or Publishing Manager dialog" on page 56.
- For an Export Filter Provider, a very similar function, `sectionForFilterInDialog`, creates only one section, rather than multiple sections. See "Defining a post-process action" on page 47.

In any case, however, the function must define the UI to be displayed when each dialog-box section is expanded. To do so, use the `viewFactory` object to construct all of the elements of a view hierarchy.

To create the containment hierarchy, use the view factory to create a container, and within that call, use it to create the child containers and controls:

```
viewFactory:group_box {
    ...initial property settings...
    viewFactory:row {-- a row of controls within the box
        ...initial property settings...
        viewFactory:static_text { -- a text label, contained in the row
            ...initial property settings...
        viewFactory:button { -- a button that responds to a click, contained in the row
            ...initial property settings...
...
```

Control nodes have properties that define a tooltip for the node, control the visibility, and affect the size, displayed font, and enabled state. Additional properties apply to controls of specific types; for instance, a pop-up menu has an `items` property, which contains a table of the selectable menu items to display. Each item is in turn a table containing a `title` (displayed string) and `value` (the value returned when the item is selected):

```
viewFactory:popup_menu {
    title = "myPopup",
    items = { -- the menu items
        { title = "First item",  value = 1 },
        { title = "Second item", value = 2 },
    },
    value = LrView.bind( "myPopup_value" ), -- the control value
    size = 'small'
},
```

- The types of containers and controls and their view properties are listed and described in "User interface elements" on page 87.
- Certain properties describe node layout; that is, the sizing and placement of each node with respect to its container and sibling nodes. You can set layout values individually, or use `LrView` functions to set spacing and margin values for an entire node tree. The layout properties and functions are described in "Determining layout" on page 110.
- Display strings in all containers and controls (generally specified in the `title` property) can be localized to different languages by using the `LOC` function to specify the string value; for details, see Chapter 7, "Using ZStrings for Localization."

Using dialog boxes

The `LrDialogs` namespace provides functions that you can use to display simple messages in predefined dialog boxes, or to define a completely customized dialog box. All dialog boxes are modal, meaning that when the dialog is invoked, no other actions can be taken in the Lightroom UI until the dialog is dismissed.

Displaying predefined dialog boxes

The predefined dialog boxes display:

- Messages

  Message dialogs display your text message to the user. They have a single OK button that dismisses the dialog; you can specify the button text. One version has a "Don't show again" checkbox, so that the user can prevent this message from being displayed next time the same situation occurs.

- Confirmations and prompts

  In addition to your text message, these dialogs have configurable OK and Cancel buttons. These return different values to the invocation function, which you use to decide on the action to be taken. Again, there is a "Don't show again" version.

  These dialogs are extensible; you can define an optional third button, or a small UI section that you define using `LrView`; see "Creating custom dialog boxes" on page 87.

- Errors

  You can display a simple error message with a single OK button, or you can wrap an error dialog around a function context, so that if the wrapped function throws an error, the dialog appears. See "Using function contexts for error handling" on page 18.

- Platform Open File and Save File

  You can bring up the platform-defined file-selection dialogs, so that the user can choose a file system location.

Creating custom dialog boxes

You can use the `LrDialogs.presentModalDialog()` function to create a completely customized dialog box, which you can, for example, invoke from a menu item that your plug-in adds to the Lightroom menu bar, using one of the menu service items: `LrExportMenuItems`, `LrLibraryMenuItems`, or `LrHelpMenuItems`.

Most of the contents of this dialog are defined by an `LrView` hierarchy that you define. To build the contents of a custom dialog, obtain a factory object using the `LrView` namespace function `LrView.osFactory()`. Like the confirmation dialogs, this dialog automatically contains configurable OK and Cancel buttons.

You can choose to make this dialog user-resizeable, and can also choose to save its most recent frame size as one of your plug-in settings. The location of the dialog is also saved, if the user moves it.

The example code in "Building a basic dialog" on page 113 demonstrates how to build and invoke a custom dialog within a function context.

Floating (non-modal) dialogs can be created using the `LrDialogs.presentFloatingDialog()` function in a similar manner to that described for `LrDialogs.presentModalDialog()` above. For specifics, see the *Lightroom SDK API Reference*. Use floating dialogs when it is desirable to let the user interact with Lightroom's main window, as well as the dialog itself, while the dialog box is open.

User interface elements

This section provides details of the types of container and control nodes you can create with an `LrView` factory object.

Containers

When creating a dialog or a section for the Plug-in Manager or Export dialog, you generally begin with a top-level container, then, within that container, create its children. Depending on the complexity of your

interface, the children can be nested containers (such as a tabbed view that contains tabbed pages), placement containers (rows and columns), or the visible controls (such as text and buttons).

- All containers have the shared view properties listed in "General view properties" on page 94, except as mentioned.
- All containers except `spacer` have the layout properties listed in "Determining layout" on page 110.

Types of containers are:

Container type	Description	Type-specific properties
`view`	A basic containment frame for a set of controls, with no visual representation.	
`group_box`	A visible containment frame for a set of controls. Can have a localizable `title`, which is displayed near the top left corner of the frame.	**`title`** and **`font`**: See "Control node view properties" on page 95 **`show_title`**: True to display the title. Default is true.
`tab_view`	A container of tabbed pages. The containing `tab_view` draws the frames for its `tab_view_item` children, but has no title. The font is used for the tab text of the children.	**`font`** and **`size`**: See "Control node view properties" on page 95 **`value`**: The identifier of the currentlly selected tab.
`tab_view_item`	A tabbed page in a `tab_view`. The localizable `title` text is displayed in the tab.	**`title`**: The display text for the tab. **`identifier`**: A unique identifier string for this page, used to select the current tab in the parent tab view container. Required. *Note: Previous versions of this document incorrectly stated that the identifier could be any type.*
`column` `row`	These group controls for layout purposes, but do not otherwise affect the child nodes.	These do not have any non-layout properties, such as `visible`. Otherwise, a `column` or `row` is the same as a `view` with `place = vertical` or `horizontal`.
`spacer`	This is a row that contains no child nodes. It is used only for spacing.	**`width`**, **`height`**: The size in pixels.

Controls

You can use the `LrView` factory to create visible controls of types common to Windows and Mac OS interface systems. If the creation function is called within the creation of a container, the control is a child of that container.

For complete details of how to create the controls and specify their appearance and behavior, see the *Lightroom SDK API Reference*. The following table summarizes the available control types and lists their type-specific properties.

- All controls have the shared view properties listed in "General view properties" on page 94 and "Control node view properties" on page 95.
- All controls have the layout properties and functions described in "Determining layout" on page 110.

Control type	Description	Type-specific properties
`checkbox`	Displays the `title` text with a platform-style checkbox button. A checkbox is checked (selected) when its `value` is equal to its `checked_value`, and unchecked (deselected) when its `value` is equal to its `unchecked_value`. If its `value` has any other value, the button shows a mixed state. See "Binding checkbox selections" on page 101.	**`title`**: Display label. **`value`**: The control value. **`checked_value`**: When the box is selected (checked) this becomes the control value. Default is true. **`unchecked_value`**: When the box is deselected (unchecked) this becomes the control value. Default is false. All text properties. See "Text view properties" on page 97.
	NOTE: The comparison of property values is very specific; the values 0, false, nil, and "" (the empty string) are all distinct.	
`color_well`	Displays a color, and responds to a click by displaying a predefined color-selection UI.	**`value`**: The value, an `LrColor` object.
`combo_box`	An editable text field with a pop-up menu of predefined text values. User can enter any text, or select from the menu. When an item is selected from the menu, its `value` becomes the control `value`, and is displayed in the text field. See "Binding combo box selections" on page 104.	**`items`**: An array of entry values to appear in the menu, or a function that returns such an array. The values are not localizable in place; to localize, you must build the array with localized strings. All edit and text properties. See "Edit-field view properties" on page 95 and "Text view properties" on page 97.

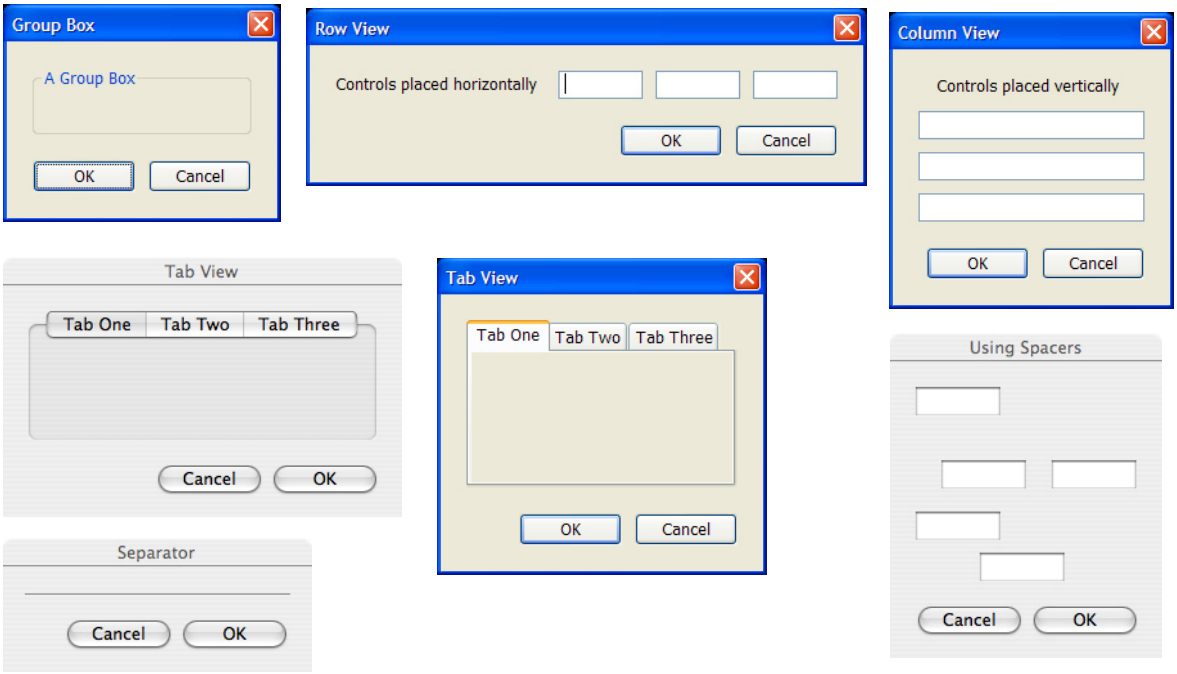
Control type	Description	Type-specific properties
`edit_field`	An editable text field. An edit field accepts keyboard input when it has the input focus. User input is committed (that is, the `value` is updated) with every keystroke if `immediate` is true. If `immediate` is false, input is committed when the control loses focus. There is a platform difference in the focus behavior: • In Windows, the control loses focus when the user clicks outside it. • In Mac OS, it loses focus when the user uses TAB to shift the focus, not when the user clicks outside the control.	All edit and text properties. See "Edit-field view properties" on page 95 and "Text view properties" on page 97.
	NOTE: When the user types ENTER/RETURN in an edit field, the default button of the containing dialog is invoked. In the case of an Export dialog section, this is the Export button. If you do not want that to happen, disable the default button until the user indicates that input is done.	
`password_field`	An editable text field that obscures the entered text, displaying only bullet characters.	All edit and text properties. See "Edit-field view properties" on page 95 and "Text view properties" on page 97.
`picture`	A static image or icon. **TIP:** you can typically get the path to the image file using this function: `_PLUGIN:resourceId('myPic.png')`	**`value`**: The full path to the JPG or PNG image file. **`frame_width`**: Pixel width of a frame to draw around the image. Default: 0. **`frame_color`**: An `LrColor` object. Default is black.
`popup_menu`	A pop-up menu of choices, each with a `title` and `value`. When the user pops up the menu and makes a choice, the selected item's `title` and `value` become those of the control. The current `title` text is displayed in the control when the menu is not open. See "Binding pop-up menu selections" on page 103 for example of how to specify `items` and use the `value_equal` function.	**`title`**: Display label. **`value`**: The value of the currently selected item. **`items`**: A table of items to appear in the menu. Each selectable item entry contains a `title` and a `value`. The `title` text is displayed when the menu is open. An entry of `separator=true` creates a separator, an unselectable line in the menu. **`value_equal`**: A function that compares the current control `value` to each item's `value` in turn, to determine the selection. All text properties. See "Text view properties" on page 97.

Control type	Description	Type-specific properties
`push_button`	A button that responds to a click with an action. Drawn in platform-standard style with a rounded appearance.	**`title`**: Display label. **`action`**: A function defining the action to be taken when the button is clicked, in the form `myAction( button )`. All text properties. See "Text view properties" on page 97.
`radio_button`	Displays the `title` text with a platform-style radio button. The button is checked (selected) when its `value` is equal to its `checked_value`, and unchecked (deselected) when its `value` has any other value, except nil. When the `value` is nil, the button shows a mixed state. Within a container, only one of a set of radio buttons should be selected. Selecting one button should deselect all others in the set. You must enforce this in the way you bind the button values. It is not automatic. See "Binding radio button selections" on page 102.	**`title`**: Display label. **`value`**: The control value. **`checked_value`**: A value that indicates the selected state.
`slider`	A draggable "bug" that changes an associated numeric value.	**`value`**: The current numeric value. **`min`**: The low end of the range. **`max`**: The high end of the range. **`integral`**: True to change only by integer increments. Default is false.
`static_text`	Text that typically does not respond to user input, such as a label or instructions. Although the user cannot change the text, it can still be made dynamic by binding the `title` to a data value; see "Binding UI values to data values" on page 97.	**`title`**: Display label. **`truncation`**: Where to truncate the text if needed, `"head"`, `"middle"`, or `"tail"`. **`selectable`**: True to make text selectable (in Mac OS only). **`alignment`**: Text alignment, `"left"`, `"center"`, or `"right"`. **`text_color`**: An `LrColor` object. Default is black. **`mouse_down`**: A function called in response to this event; takes one argument, the view object that was clicked. All text properties. See "Text view properties" on page 97.

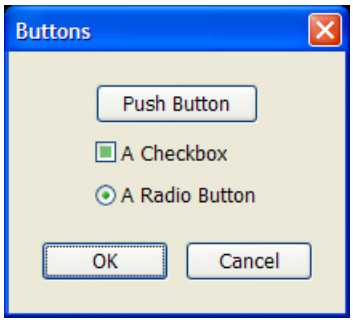
Control type	Description	Type-specific properties
`separator`	Draws a line across its container, but has no other behavior. The line is always 2 pixels in width, and is drawn either vertically or horizontally, depending on the fill value. If both values are set, the larger value determines the direction of the line.	**`fill_horizontal`**: The width of the horizontal line, a percentage of the parent's width in the range [0..1]. **`fill_vertical`**: The height of the vertical line, a percentage of the parent's height in the range [0..1].

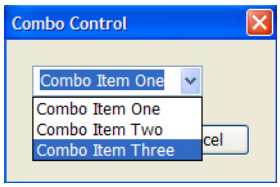
The following figures show examples of the various control and container types. The appearance is appropriate to the platform; these examples show some of each.

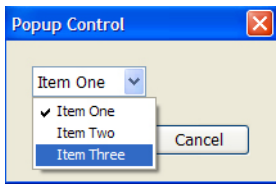
Containers and placement controls

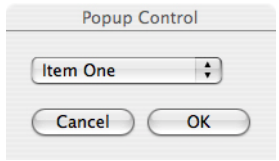


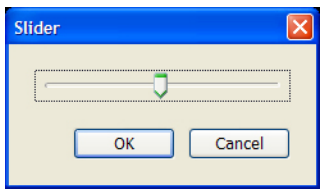
Buttons. selection, edit and text controls

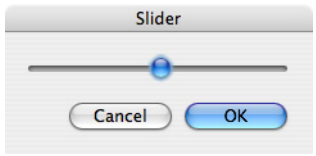




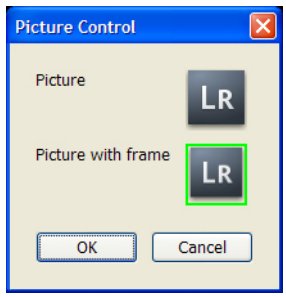


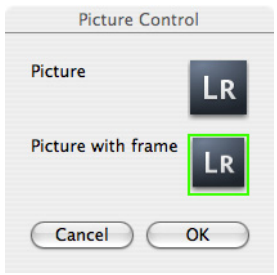


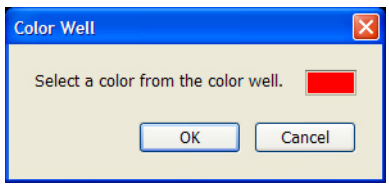


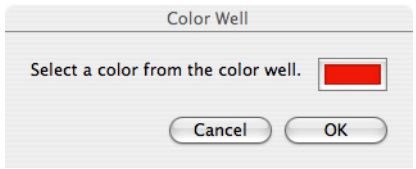


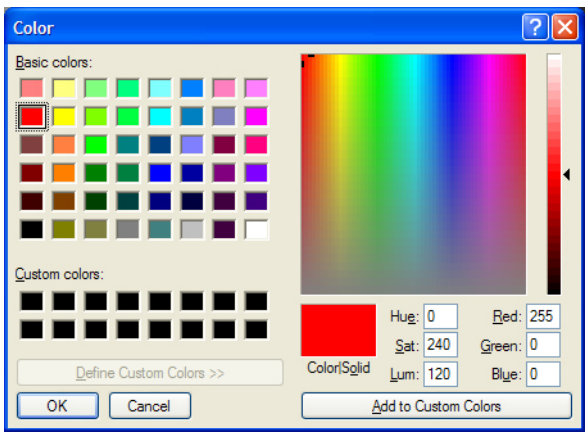
Other controls

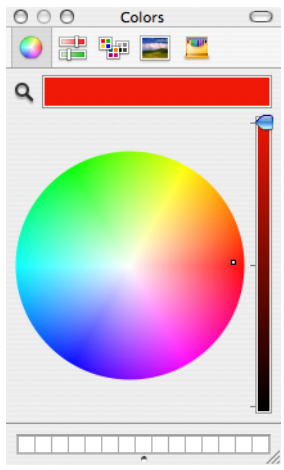












View properties

Properties in container and control nodes affect the layout of the controls, and their appearance. Layout properties, and certain view properties, are available to all nodes, both containers and controls. Other view properties are available only in control nodes. Most types of controls have additional view properties specific to their type; these are reflected in the creation parameters.

General view properties

Of the properties that are available in both containers and controls, many are connected with layout behavior; these are discussed separately in "Determining layout" on page 110. The following view properties are available in all containers and controls except the layout containers, `row` and `column`:

View property	Datatype	Description

`bind_to_object`
—or—
`object`
	table	The default bound property table for this object and its children. The default can be overridden at any level of the node hierarchy, or for individual property bindings. See "Binding UI values to data values" on page 97.
		
		In views created with `sectionsForTopOfDialog` and `sectionsForBottomOfDialog`, this is set automatically to the property table passed along with the view factory. This creates a binding between all nodes in the view and the settings table, so that any node can observe any setting.

`tooltip`	string	A help string that appears when the cursor hovers over a container or control. Default nil.
`visible`	Boolean	Determines whether a container or control is shown or hidden. This is not the same as being enabled or disabled; the disabled state is only applied when a control is visible. Default is true. • When true in a container, the container is visible, and its children are visible according to their individual visibility values. • When false in a container, the container and all of its child nodes are hidden, regardless of the value in each child node. • When true in a control, the control is visible if its parent is visible. When false, it is hidden even if its parent is visible. Value must be `true` or `false`; do not use `nil`. **TIP:** An item still affects layout, even when it is hidden.

Control node view properties

These properties are available in control nodes of all types, but not in containers.

Control node property	Datatype	Description
`enabled`	Boolean	When true, the control is drawn normally and is sensitive to user input. When false, it is drawn with a grayed appearance and does not respond to input. Value must be `true` or `false`; do not use `nil`.

`font`	string or table	The font to be used for this control, if it contains text. Can be:
		
		• A string with the name of the font.
		
		• One of these strings:
		`<system>`
		`<system/small>`
		`<system/bold>`
		`<system/small/bold>`
		
		• A table with the keys `name` and `size` (see `size` property)

`size`	string	The size of text in the control (if not otherwise determined by the `font` specification) and of other visual features in non-text controls. For example, affects the track and thumb size in a slider.
		
		One of:
		`regular` (the default)
		`small`
		`mini`

Edit-field view properties

These properties are available in control nodes that contain editable text; these include `edit_field`, `combo_box`, and `password_field`.

Property	Datatype	Description
`value`	any	Value to be displayed.
`min`	number	The minimum value allowed. If specified, the field is numeric.
`max`	number	The maximum value allowed. If specified, the field is numeric.
`precision`	number	The number of decimal places to display. Default is 2. If specified, the field is numeric.

Property	Datatype	Description
`alignment`	string	Alignment of text in frame, `left` (the default), `center`, or `right`.
`text_color`	`LrColor`	The color of displayed text. Default is black.
`immediate`	Boolean	If true, the field commits its value as the user is typing, and the validate function is called for every change. Default is false, validation occurs on loss of focus.
`auto_completion`	Boolean	True if the field should auto complete as the user types. Default is false.
`completion`	table or function	A table of strings for completion, or a function that returns a table of strings: `myCompletion( view, partialWord )`
`increment`	number	If field is numeric, the amount to increment the value (without SHIFT key). If the precision is 0, default is 1; otherwise default is 0.1.
`large_increment`	number	If field is numeric, the amount to increment the value when the SHIFT key is held down. If the precision is 0, default is 10; otherwise default is 1.
`validate`	function	A function called to validate the value: `myValidate( view, value )` Returns *result*, *value*, *message*: *result*: (Boolean) True if value was valid. *value*: (any) The new value. *message*: (string) An error message to be displayed if result is false.
`value_to_string`	function	An optional conversion function, called to convert a non-string value to a display string. Takes arguments *`view`* (this control) and *`value`* (the entered value), and returns a string.
`string_to_value`	function	An optional conversion function, called to convert the display string to a non-string value. Takes arguments *`view`* (this control) and *`string`* (the string), and returns a value of the required type.
`wraps`	Boolean	True to wrap text. Default is true.

Text view properties

These properties apply to any control that displays text, including `popup_menu`, `static_text`, and `push_button`, as well as the editable text controls.

Property	Datatype	Description
`width_in_chars`	number	Calculates the minimum width using this as the number of m characters that should fit. Considered together with `width_in_digits`. Default is 15 for editable text fields.
`width_in_digits`	number	Calculates the minimum width using this as the number of 0 digits that should fit. Considered together with `width_in_chars`. Default is 0.
`height_in_lines`	number	Calculates the minimum height using this as the number of lines that should fit within the field. Default is 1.

Binding UI values to data values

Bindings allow you to make your UI dynamic by specifying a relationship between an `LrView` object's state and current values in an *observable table* (see "Creating observable property tables" on page 100). This can be the export-settings property table provided by the API, or a table that you create for local program data that you define.

When you create a binding, the value or state of the UI element reflects the data value, and the data value reflects the UI element state. This is a two-way relationship; when the binding is established, the data value from the table is pushed to the view, and when the user changes the bound value in the view (by selecting a checkbox, for instance, or entering a value in a text field), the table is notified and the corresponding data key value or values change accordingly. Similarly, when your program changes a value in the table, the bound UI elements are updated to display the new value.

To create bindings:

1. Specify a bound table at some level of the view hierarchy. Set the table as the value of the `bind_to_object` property (you can also use the name `object` for this property). The bound table of a parent container is inherited by its children, but can be overridden.

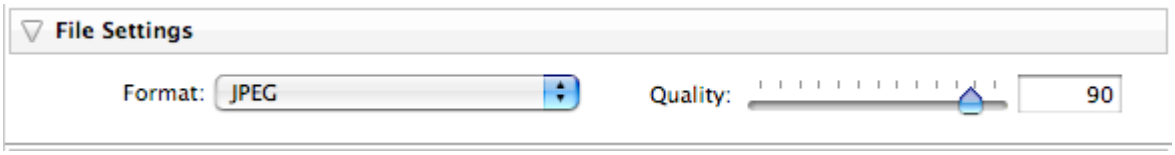
    - When you create a dialog box, you must set the bound table explicitly.
    - When you create a section for the Plug-in Manager or Export dialog using the `sectionsForTopOfDialog` or `sectionsForBottomOfDialog` functions, the settings table for your plug-in is passed to those functions as the *propertyTable* value. This table contains both export settings that you have defined for your plug-in (see "Remembering user choices" on page 59) and Lightroom-defined export settings (see "Lightroom built-in property keys" on page 61).

      The *propertyTable* is automatically set as the default bound table for all of the UI elements in the view hierarchy for that section. However, the bindable `synopsis` for the section is not part of the view hierarchy; if you want to make that value dynamic, you must specify the table explicitly. See "Adding custom sections to the Plug-in Manager" on page 34.

2. For each specific UI element, set the value of each dynamic property using the `LrView.bind()` function to associate that value with a specific key in the bound table.

    - The simplest binding simply mirrors the key value and the property value; for instance, setting one value to true sets the other value to true.
    - You can use the `LrBindings` functions to create other common mappings between the bound key value and the view property value. See "Specifying bindings" on page 98.
    - For more complex mappings, see "Transforming values" on page 106 and "Binding multiple keys" on page 108.

A typical example is a binding between the `visible` property and a particular settings value, so that a control is only shown when the appropriate setting is present. For example, in the File Settings section of the Export dialog, the control that appears next to the Format combo box changes according to the selected format.



When JPEG is selected, there is a slider for setting the Quality. When you select the TIFF format, the slider is hidden and a Compression pop-up menu is shown. For PSD and DNG, both of these controls (and their labels) are hidden.

To accomplish this, Lightroom binds the Format pop-up menu's `value` property to the `LR_format` setting. Then the `visible` property of the slider and its label are bound to the `JPEG` value of `LR_format`. The example code in "Changing the contents of a view dynamically" on page 115 shows how to use bindings in this way, setting the visibility state of different sets of controls, based on the selection in a pop-up menu.

Notice in this example that two control values are related by being bound to the same key value; this is how you bind control values to one another.

> **NOTE:** Bindings are used to create dynamic text in `LrView` objects only. The title of a dialog box, for example, is not part of an `LrView` object, and you cannot bind it. Similarly, the title of an Export dialog section cannot be bound.

Specifying bindings

The `LrView` namespace function `LrView.bind()` creates a direct association between a key or keys in an observable table and a view property value. Use this function when creating the view or control, to specify the view property value. For example:

```
visible = LrView.bind( "LR_export_useSubfolder" )
```

Many of the sample plug-ins create a shortcut to the function:

```
local bind = LrView.bind -- shortcut for bind() method
```

You can then use the shortcut to specify dynamic property values:

```
...
    viewFactory:static_text {
                title = bind 'mySetting',
...
```

The required argument of `LrView.bind()` is the key name; by default, this is in the table that is already bound to the UI element; that is, the value of `bind_to_object` in the same UI element. This is inherited in the view hierarchy, but can be overridden at any level.

You can override the bound table for a specific binding by passing the `LrView.bind()` function a table containing both the key and the table it comes from:

```
visible = LrView.bind { key = "mySetting", bind_to_object = "myTable" }
```

This allows you to bind different properties in one view object to keys in different tables.

The bound table is typically the export-settings table, since your UI is typically a way for your user to see and set these values. The SDK makes this default case easy for sections that you define for the Export dialog. In views created with `sectionsForTopOfDialog` and `sectionsForBottomOfDialog`, the value of `bind_to_object` for the entire view hierarchy is set automatically to the export-settings table passed along with the view factory. See "Adding custom sections to the Plug-in Manager" on page 34.

Simple bindings

The simplest binding is between a property in the `LrView` object and a settings key of the same datatype, and simply keeps the two synchronized. For example:

```
visible = LrView.bind( "LR_export_useSubfolder" )
```

In this case, both the local property (`visible`) and the bound table item (a Lightroom-defined export setting) have Boolean values. Setting the use-subfolder preference to true (in the Export dialog, for instance) makes the control visible.

For some other common types of binding, you can use an `LrBinding` function as the value assignment; for example:

```
visible = LrBinding.negativeOfKey( "LR_export_useSubfolder" )
```

This binds the property to the opposite of the table value; that is, setting the use-subfolder preference to true hides the control. The binding works in both directions; that is, hiding the control would also set `LR_export_useSubfolder` to true. This function can be used to negate numeric as well as Boolean values; for example, a value of 2 would become -2.

Although `negativeOfKey()` works both ways, and with numeric values, the other `LrBinding` functions can be used only with Boolean values, and work only in one direction; a change in the bound table sets the bound property value, but not the reverse. The `LrBinding` functions allow you to:

- Set a Boolean property to the opposite of a Boolean key value, or a numeric property to the negation of a numeric key value (`LrBinding.negativeOfKey`).
- Set a Boolean property when a key value is or is not present (`LrBinding.keyIsNil`, `keyIsNotNil`).
- Set a Boolean property when a key value is or is not equal to a specific value (`LrBinding.keyEquals`, `keyIsNot`).

- Set a Boolean property when a set of Boolean keys are either all true, or when any one is true (`LrBinding.andAllKeys`, `orAllKeys`); for more information on how this works, see "Binding multiple keys" on page 108.

For details of the `LrBinding` functions, see the *Lightroom SDK API Reference*.

Creating observable property tables

The Lightroom SDK defines a notification mechanism based on *observable tables*. When a value changes in an observable table (such as the export-settings table), all registered observers (typically `LrView` objects) are notified. A plug-in uses this mechanism to make UI controls in the plug-in's user interface respond to changes in data properties.

The `LrView` objects that define your UI elements in an Export dialog section are automatically registered as observers of the export-settings table that is passed on creation; see "Adding custom sections to the Plug-in Manager" on page 34.

To use export settings in another context, or to define additional program data, use the function `LrBindings.makePropertyTable()` to create an observable table, and populate it with your own plug-in settings or any other program data.

An observable table must be created with a *function context*, so that Lightroom can clean up the notifications if anything goes wrong. (See "Using function contexts for error handling" on page 18.) A function context is created using `LrFunctionContext.callWithContext()`. This passes a function-context object to its main function; you pass that object on to your table-creation function. For example:

```
LrFunctionContext.callWithContext("showCustomDialog", function( context )
    local properties = LrBinding.makePropertyTable( context )
    properties.url = "http://www.adobe.com" -- create a settings value
    -- add code to take create dialog contents
end)
```

When you create a new table, it is initially empty. You can explicitly add keys and values, as in the example. However, it is not necessary to add a key to a table before you reference it in a binding; if it is not yet in the table, its value is nil. The example in "Transforming values" on page 106 shows how a control's `value` is bound to a key that is not yet in the table. When the control first gets a value, the key is put into the table with that value.

TIP: You can use a naming convention to distinguish program data from persistent export settings (that is, those specified in the `exportPresetFields` table; see "Remembering user choices" on page 59). For example, you might use an underscore prefix, "`_tempUrl`," to indicate a local data property.

Adding observers to tables

You can create a general and flexible response to a change in an observable table by adding an *observer*. An observer associates a function that you define with a key in the table, so that whenever the key value changes, the function is called.

To receive notification of changes in the table you create, use this function to register an observer of the table:

```
propertyTable:addObserver( key, func )
```

For example:

```
LrFunctionContext.callWithContext("showCustomDialog", function( context )
    local myPropTable = LrBinding.makePropertyTable( context )
    mypropTable:addObserver( 'mySetting', function( properties, key, newValue )
        -- do something when this value changes
        end )
    -- add code to create dialog contents
    end)
```

The handler function you specify for your observer takes as arguments the observed table (so you can access other data values), the key whose value change triggered the notification (in case you are using the same handler function for multiple keys), and the new value of that key.

You can define a function to handle more than one key notification, using the `key` argument to distinguish which key changed. If you do, you must pass the function to a separate `addObserver()` call for each key.

For examples of how and why to add an observer to a table, see "Binding combo box selections" on page 104, and Chapter 9, "Getting Started: A Tutorial Example".

Bindings for selection controls

Controls that have a selection state include checkboxes, radio buttons, pop-up menus, and (to some extent) combo boxes. You can use bindings to keep track of the selection state, and to create dependencies between what is selected in one of these controls and what is shown elsewhere in the UI, or what actions are taken.

Binding checkbox selections

The `value` property of a radio button or checkbox controls and reflects the current selection state:

- In both, if the user checks the button, the `checked_value` becomes the new control `value`.
- In the checkbox, if the user unchecks the button, the `unchecked_value` becomes the new control `value`.

EXAMPLE 1: This example shows how bindings work in checkboxes:

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrView = import "LrView"
local bind = LrView.bind -- shortcut for bind() method
LrFunctionContext.callWithContext( 'bindingButtonsExample', function( context )
    local f = LrView.osFactory()  -- obtain view factory
    local properties = LrBinding.makePropertyTable( context ) -- make prop table
    -- create some keys with initial values
    properties.checkbox_state = 'checked' -- for checkbox
    properties.my_value = 'value_1' -- for radio buttons and pop-up menu
    local contents = f:column { -- create view hierarchy
        fill_horizontal = 1,
        spacing = f:control_spacing(),
        bind_to_object = properties, -- default bound table is the one we made
        f:group_box {
            title = "Checkboxes", -- (only one here)
            fill_horizontal = 1,
            spacing = f:control_spacing(),
```

```
            f:checkbox {
                title = "Value will be string",
                value = bind 'checkbox_state', -- bind to the key value
                checked_value = 'checked', -- this is the initial state
                unchecked_value = 'unchecked', -- when the user unchecks the box,
                                               -- this becomes the value, and thus
                                               -- the bound key value as well.
            },
            f:static_text {
                fill_horizontal = 1,
                title = bind 'checkbox_state', -- bound to same key as checkbox value
            },
        },
-- (add radio button container here for example 2)
-- (add pop-up container here for example 3)
    local result = LrDialogs.presentModalDialog( -- invoke the dialog
        {
            title = "Binding Buttons Example",
            contents = contents,
        }
    )
end )
```

Binding radio button selections

The user cannot uncheck a radio button; a selected button is deselected only when another button in the group is selected. Simply putting the buttons in the same container does not enforce this usage; to arrange it, bind the value of each button in the set to a different value of the same key.

EXAMPLE 2: Add this to the previous code for an example of binding in a set of radio buttons:

```
        f:group_box { -- the buttons in this container make a set
            title = "Radio Buttons",
            fill_horizontal = 1,
            spacing = f:control_spacing(),
            f:radio_button {
                title = "Value 1",
                value = bind 'my_value', -- all of the buttons bound to the same key
                checked_value = 'value_1',
            },
            f:radio_button {
                title = "Value 2",
                value = bind 'my_value',
                checked_value = 'value_2',
            },
            f:radio_button {
                title = "Value 3",
                value = bind 'my_value',
                checked_value = 'value_3',
            },
            f:static_text {
                fill_horizontal = 1,
                title = bind 'my_value',
            },
        },
```

Binding pop-up menu selections

The pop-up menu and the menu component of a combo box allow you to specify a set of choices, using an `items` table; each item entry is a table containing a `title` and `value`. The `title` is localizable display text, that appears in the menu (see Chapter 7, "Using ZStrings for Localization").

```
items = { { title = "First item",     value = 1 },
          { title = "Second item",    value = 2 },
          { title = "Third item",     value = 3 }, },
```

The value of the item that the user selects from the menu becomes the control's `value`. For the pop-up menu, the `title` becomes the control's `title`, and is displayed in the control when the menu is not shown. (For the combo box, the displayed text is the `value`, or the result of the `value_to_string` function; see "Edit-field view properties" on page 95.)

EXAMPLE 3: This code fragment adds a pop-up menu to the previous example, with the currently selected value from the menu similarly bound to a static text value:

```
        f:group_box {
            title = "Popup Menu",
            fill_horizontal = 1,
            spacing = f:control_spacing(),
            f:popup_menu {
                value = bind 'my_value', -- current value bound to same key as static text
                items = { -- the menu items and their values
                    { title = "Value 1", value = 'value_1' },
                    { title = "Value 2", value = 'value_2' },
                    { title = "Value 3", value = 'value_3' },
                }
            },
            f:static_text {
                fill_horizontal = 1,
                title = bind 'my_value', -- bound to same key as current selection
            },
        },
    }
```

You can bind the `items` property to a settings key to create a dynamic menu. However, you can only set the whole menu at once; you cannot bind individual item values.

EXAMPLE 4: This code binds the currently selected value from a pop-up menu to the same key as an editable text value. The user can change this value by entering any text in the edit field; the entered text shows up immediately as the value of the pop-up control.

However, since the user can enter any text, that text might not match the menu items. This code shows how to use the pop-up control's `value_equal` function to do a case-insensitive comparison of the user-entered value with the item values. The function is called for each item until it returns true, or has gone through all the items.

- If the entered text matches one of the item values (that is, the function returns true), the matching item becomes the selected item in the pop-up menu, and the item's `title` text is displayed in the pop-up control.
- If the function goes through all the items without finding a match, the pop-up control shows no selection; that is, it appears blank, and the next time the user pops up the menu, none of the items is in the selected state. The entered value remains in the pop-up control's `value` property.

```
local LrDialogs = import "LrDialogs"
```

```
local LrFunctionContext = import "LrFunctionContext"
local LrStringUtils = import "LrStringUtils"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'bindingExample', function( context )
    local f = LrView.osFactory()
    local properties = LrBinding.makePropertyTable( context )
    properties.format = "jpeg"
    local contents = f:column {
        spacing = f:control_spacing(),
        bind_to_object = properties,
        f:popup_menu {
            items = {
                { title = "JPEG", value = "jpeg" },
                { title = "TIFF", value = "tiff" },
            },
            value = LrView.bind 'format',
            value_equal = function( value1, value2 )
                return LrStringUtils.lower( value1 ) == LrStringUtils.lower( value2 )
            end,
        },
        f:edit_field {
            immediate = true,
            value = LrView.bind 'format',
        },
    }
    local result = LrDialogs.presentModalDialog(
        {
            title = "Dialog Example",
            contents = contents,
        }
    )
end )
```

Binding combo box selections

For a combo box, the user can enter text in the edit-field portion, which becomes the new control `value`. If you select an item from the menu portion, that item value becomes the control `value`; this provides an input shortcut for the user. Unlike the pop-up menu, the combo box menu items are simple values; if you need to localize them, you must do so when building the item array.

This example shows how to create a dynamic menu for a combo box that gives previously-entered values as menu choices. This code:

- Binds the `value` and `items` of the combo box to data properties `storeValue` and `storeItems`.
- Creates an observer for the `storeValue` property, so that a change in that property (caused by entering a new value in the combo box) calls a function.
    - The observer function checks to see if the current value is already in the items list (stored in `storeItems`), and if it is not, adds it to the list.
    - Because of the binding, any change the function makes to the `storeItems` property is automatically reflected in the combo box `items`.

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrStringUtils = import "LrStringUtils"
```

```
local LrView = import "LrView"
-- Create an observable table within a function context.
LrFunctionContext.callWithContext( 'bindingExample', function( context )
    -- Obtain the view factory.
    local f = LrView.osFactory()
    -- Create the observable table.
    local properties = LrBinding.makePropertyTable( context )
    -- Add an observer of the storeValue property.
    properties:addObserver( 'storeValue', function( properties, key, newValue )
        local items = properties.storeItems -- get current items list from this table
        if items == nil then
            items = {}
        end
        -- Check if current value is already in the list.
        local inList
        for i, v in ipairs( items ) do
            if v == newValue then
                inList = true
                break
            end
        end
        -- If not, add it.
        if not inList then
            items[ #items + 1 ] = newValue
        end
        -- Reset data value with current items list
        properties.storeItems = items
    end )
    -- Create the view hierarchy for the dialog.
    local contents = f:column {
        spacing = f:control_spacing(),
        bind_to_object = properties, -- bound to the table we created
        f:combo_box {
            value = LrView.bind 'storeValue', -- bind to observed key
            items = LrView.bind 'storeItems', -- bind to data value that the
                                              -- observer modifies
        },
    }
    -- Display the dialog.
    local result = LrDialogs.presentModalDialog(
        {
            title = "Dialog Example",
            contents = contents,
        }
    )
end )
```

Complex bindings

The `LrBinding` functions provide a particular, limited set of value transformations. To specify more complex bindings, the argument to `LrView.bind()` can be a table with these items:

`key`	A key name in the bound table. The value can be mapped to a value for the local property by the `transform` function.

`bind_to_object`
—or—
`object`
	Optional. The name of an observable table which overrides the value of the `bind_to_object` view property.

`transform`	Optional. A function that maps the key or key values to the local property value. See "Transforming values" on page 106. This function is called immediately when the value changes in either the bound view property or the bound table key.

Here is an example of binding to keys in two different tables in a single view object:

```
...
visible = LrView.bind("myBooleanSetting"), -- simple binding between two
                                           -- Booleans in the default table
enabled = LrView.bind( {
    key = "mySetting", -- a single key
    bind_to_object = mySettingsTable, -- a non-default bound table
    transform = function( value, fromTable ) -- a mapping function
            ...
        end
    }
)
...
```

Transforming values

The transformation function that you specify for a binding maps the value of a key in the bound table to a value in the bound property. If the `LrBinding` functions do not provide mapping that you need, define your own transformation function. It is passed these parameters:

`value`: The new value of the key or property that changed.

`fromTable` (Boolean): True if the change that triggered this notification was in the bound table, false if the change was in the bound view property.

Your function should return the new value for the destination property or key.

This simple example creates a slider with a range of 0-110, then reports when the value goes over 100, by using a transformation function. The slider `value` and the `visible` property of a text box are bound to the same key. For the text box, the transform function returns true (making `visible` true) only when the value is over 100.

```
sectionsForTopOfDialog = function(viewFactory, propertyTable)
    return {
        {
            title = "Section Title",
            viewFactory:slider {
                min = 0,
```

```
            max = 110,
            value = LrView.bind "slider_value",
            title = "slider title",
        },
        viewFactory:static_text {
        title = "You're over a hundred",
        visible = LrView.bind {
            key = "slider_value",
            transform = function(value, fromTable)
                return value > 100
                end
            }
        }
    }
}
end
```

Transformations can work in both directions; changes in the bound property affect the bound table key, and changes in the table key affect the property. If you write a custom function for a one-way transformation, return the value `LrBinding.kUnsupportedDirection` to indicate that one or the other direction is not supported by your transformation.

Here is an example of a one-way transformation. This example shows a transformation that makes a text display visible only when text is entered in an edit field. The transform function checks for a value of nil or the empty string in the key to which both controls are bound. This example pops up a dialog, so it needs to create an observable table to hold the data.

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'bindingExample', function( context )
    local f = LrView.osFactory()  -- obtain the view factory
    local properties = LrBinding.makePropertyTable( context )  -- make a settings table
    -- the new table is initially empty
    local contents = f:column {-- create view hierarchy for dialog
        spacing = f:control_spacing(),
        bind_to_object = properties, -- default bound table is the one we made
        f:row {
            fill_horizonal = 1,
            spacing = f:label_spacing(),
            f:static_text {
                title = "Type anything:",
                alignment = 'right',
            },
            f:edit_field {
                fill_horizonal = 1,
                width_in_chars = 20,
                immediate = true,
                value = LrView.bind( 'text' ), -- creates a key 'text'
                    -- the initial value of the new key is nil
                    -- setting its value (by entering text in the control)
                    -- puts it into the table
            },
        },
        f:static_text {
            place_horizontal = 0.5,
            title = "This is only visible when there is text in the edit field",
            visible = LrView.bind {
                key = 'text', -- bind to the same key
```

```
                transform = function( value, fromTable )
                    if fromTable then
                        return value ~= nil and value ~= '' -- check that key has a value
                    end
                    return LrBinding.kUnsupportedDirection
                end,
            }
        },
    }
    local result = LrDialogs.presentModalDialog( -- invoke the dialog
        {
            title = "Binding Example",
            contents = contents,
        }
    )

end )
```

Binding multiple keys

To specify even more complex bindings, between a property in a view object and multiple keys in one or more bound tables, the value part of a `binding` key-value pair can be a table with these items:

`keys`	A table specifying one or more keys. The table can have these entries: • `key`: A key name in the bound table. • `bind_to_object` or `object`: Optional. The name of an observable table which overrides both the default `bind_to_object` value and this binding's `bind_to_object` value. • `uniqueKey`: Optional. Because you can specify keys in different tables, the names might overlap. This provides a unique name that is used to identify this key in the values table passed to your `operation` function.

`bind_to_object`
—or—
`object`
	Optional. An observable table which overrides the value of the `bind_to_object` view property.

`operation`	Required. A function defining an operation to perform on the key values; the result of this operation is passed to the `transform` function. This function is called when any specified key value changes. The function you define receives three parameters: `binder`: For internal use. `values`: A special look-up table of key-value pairs with the current values of all specified keys. The `key` portion of the pair uses the `uniqueKey` name, if provided. (This is not a general-purpose table; you cannot iterate over the values.) `fromTable` (Boolean): True if the change that triggered this notification was in the bound table, false if the change was in the bound view property. This function is not called immediately, but at the end of an event cycle; this means that, if the change is in the bound table, more than one key value can have changed. If changes occur in both directions, the function is called twice.
`transform`	Optional. A function that maps the return value of the `operation` function to the local property value. See "Transforming values" on page 106.

This example shows multiple binding. The dialog contains two edit fields, each with its `value` bound to a different key. A static text box below them has its `visible` property bound to both keys; the `operation` makes it true only when both values are equal (meaning that the same text has been typed into both edit fields, or they are both empty).

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'multiBindingExample', function( context )
    local f = LrView.osFactory() -- get view factory
    local properties = LrBinding.makePropertyTable( context ) -- make empty table
    local contents = f:column { -- create view hierarchy
        spacing = f:control_spacing(),
        bind_to_object = properties, -- default bound table is the one we made
        f:row {
            fill_horizonal = 1,
            spacing = f:label_spacing(),
            f:static_text {
                title = "Type anything:",
                alignment = 'right',
                width = LrView.share( 'label_width' ),
            },
            f:edit_field {
                fill_horizonal = 1,
                width_in_chars = 20,
                immediate = true,
                value = LrView.bind( 'text1' ), -- bind to the first key
            },
        },
        f:row {
            fill_horizonal = 1,
```

```
            spacing = f:label_spacing(),
            f:static_text {
                title = "Type more:",
                alignment = 'right',
                width = LrView.share( 'label_width' ),
            },
            f:edit_field {
                fill_horizonal = 1,
                width_in_chars = 20,
                immediate = true,
                value = LrView.bind( 'text2' ), -- bind to the second key
            },
        },
        f:static_text {
            place_horizontal = 0.5,
            title = "This is only visible when the text in the two fields are equal",
            visible = LrView.bind {
                keys = { 'text1', 'text2' }, -- bind to both keys
                operation = function( binder, values, fromTable )
                    if fromTable then
                        return values.text1 == values.text2 -- check that values are ==
                    end
                    return LrBinding.kUnsupportedDirection
                end,
            },
        },
    }
    local result = LrDialogs.presentModalDialog( -- invoke dialog
        {
            title = "Multi Binding Example",
            contents = contents,
        }
    )

end )
```

Determining layout

Both the initial layout of a container, and subsequent automatic layout operations, use a set of parameters set by properties in both the container and its child nodes. These values control the initial layout, and, if the containing dialog is resizeable, the way the layout changes if the dialog size changes.

Properties control these broad categories of placement and sizing:

- Spacing values determine how child nodes are placed relative to one another. See "Relative placement of sibling nodes" on page 111.
- Margin values determine how a node is placed and sized within its parent node. See "Placement within the parent" on page 111.
- You can obtain default layout values using "Factory functions for obtaining layout values" on page 112.

Relative placement of sibling nodes

These properties determine how child nodes are placed relative to one another. They apply only to containers. A margin is the interior margin of a container, the distance between the edge of the container and its children; spacing is the distance between children. All numeric values are in pixels.

Layout property	Datatype	Description
`place`	string	The placement style. One of: `vertical` (default): Children are placed in a column top down. `horizontal`: Children are placed in a row left to right. `overlapping`: Children are placed on top of one another.
`margin`	number	Space around children within the containing node.
`margin_horizontal`	number	Overrides the `margin` value for both the right and left sides.
`margin_vertical`	number	Overrides the `margin` value for both the top and bottom.
`margin_left`, `margin_right`	number	Overrides the `margin` value for the left and right sides, respectively.
`margin_top`, `margin_bottom`	number	Overrides the `margin` value for the top and bottom, respectively.
`spacing`	number	The amount of space placed between each child. Ignored if `place` is `overlapping`.

Placement within the parent

These can be set on any view or control. These properties determine how child nodes are placed and sized within the parent node. All numeric values are percentages, between 0 and 1.

Layout property	Datatype	Description
`fill_horizontal` `fill_vertical`	number [0..1]	The amount of free space that the node is sized to fill in the given direction. These determine how a node is sized relative to its siblings. No node is made smaller than its minimum size. Each child's fill size is first treated as a proportion of the total space desired; that is, 0.25 makes the node 25% of the parent's size. If any of the child node fill needs cannot be met, they are given a percentage of the extra space in the proportion to how much they specified. For instance, if three nodes specify 0.2, 0.2, and 0.4, and there is not enough extra space, the nodes get 25%, 25% and 50% of the extra space that is available.

Layout property	Datatype	Description
`fill`	number [0..1]	The default fill value, if a specific horizontal or vertical value is not provided.
`place_horizontal` `place_vertical`	number [0..1]	The place properties determine how a node is placed in any extra space within its parent node; that is, extra space available after the fill properties have been considered. The percentage value determines how much of the extra space is placed to the left or above the node. Space allocated on a first-come first-served basis; if the first child has a `place_horizontal` value of 1, it consumes all of the extra horizontal space and there is none left for its siblings.
`width` `height`	number	The minimum size for this node in pixels, when it is automatically resized. If both are specified, the minimum size for the node is not automatically calculated. If only one is specified, the minimum size can be calculated in the other direction.

Factory functions for obtaining layout values

The `LrView` factory object defines a set of functions that you can use to obtain appropriate values for the layout properties of individual containers and controls. For example, this sets a spacing property to a recommended value for a control that is used either as a label or as the labeled object:

```
spacing = viewFactory.label_spacing()
```

Call these functions from the view factory passed to the `sectionsForTopOfDialog` or `sectionsForBottomOfDialog` function, or obtained using the `LrView` namespace function `LrView.osFactory()`.

Default layout function	Description
`dialog_spacing()`	The number of pixels between elements that is appropriate for top-level items in a dialog, such as views or group boxes.
`control_spacing()`	The number of pixels between controls or groups of controls.
`label_spacing()`	The number of pixels between a label and its control.

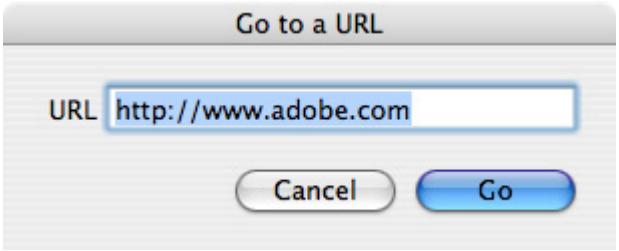
Layout examples

The following examples show how to build a basic dialog with an initial layout, how to make labels line up properly, and how to set the dialog up to take advantage of automatic layout on resize.

Building a basic dialog

The following code creates a basic dialog within a function context. (See "Using function contexts for error handling" on page 18.)

- It creates a properties table with a plug-in defined property, `url`, which contains a URL.
- It defines the contents of the dialog box using an `LrView` factory: a label, and an edit field that shows the property value.
- It invokes a modal dialog with `LrDialogs.presentModalDialog()`, passing in the defined view. Based on the result of the invocation, it opens the web page using `LrHttp`.



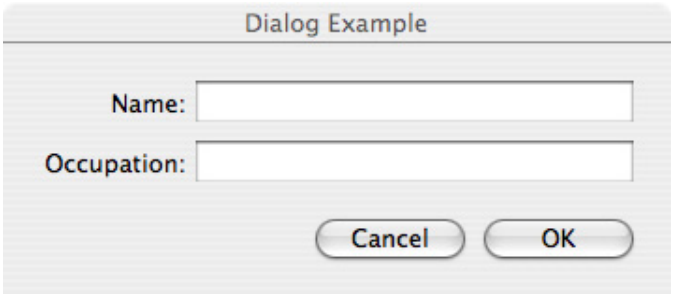
This code demonstrates a very simple layout, where the topmost and only container is a `row` view, which uses default values to place its two children, a label and an edit field.

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrHttp = import "LrHttp"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'dialogExample', function( context )
    local f = LrView.osFactory()  --obtain a view factory
    local properties = LrBinding.makePropertyTable( context ) -- make a table
    properties.url = "http://www.adobe.com" -- initialize setting
    local contents = f:row {   -- create UI elements
        spacing = f:label_spacing(),
        bind_to_object = properties,  -- default bound table is the one we made
        f:static_text {
            title = "URL",
            alignment = 'right',
        },
        f:edit_field {
            fill_horizonal = 1,
            width_in_chars = 20,
            value = LrView.bind( 'url' ),-- edit field shows settings value
        },
    }
    local result = LrDialogs.presentModalDialog(  -- invoke a dialog box
        {
            title = "Go to a URL",
            contents = contents,   -- with the UI elements
```

```
            actionVerb = "Go",   -- label for the action button
        }
    )
    if result == 'ok' then -- action button was clicked
        LrHttp.openUrlInBrowser( properties.url )
    end
end )
```

Making labels line up

Typically, a dialog contains vertical sets of controls and their labels. The following code demonstrates how make right-aligned labels on the left side of the dialog, with matching left-aligned controls on the right side.



To make this happen, the example uses the `alignment` property and the `LrView.share()` function.

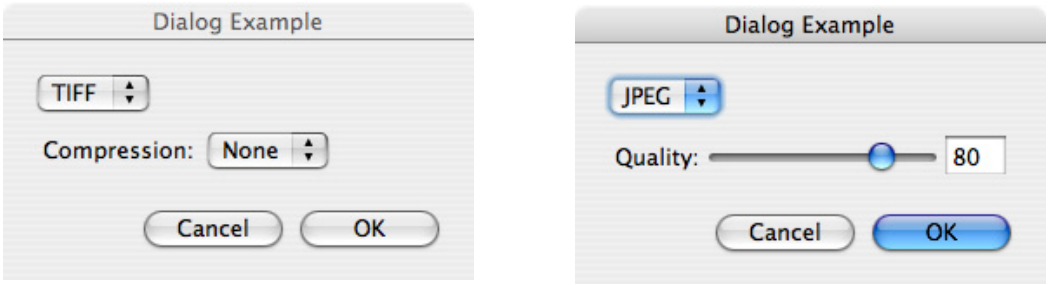
- The `alignment` property determines whether a control is right-aligned, left-aligned, or centered. Since at least one of these labels is wider than the text it is showing, the labels need to be right-aligned. Labels should generally be right-aligned in any case, because if the dialog is translated, the size of the text changes.
- The namespace function `LrView.share()` binds a property value to an identifier that has no value of its own, but indicates that this property value is to be shared across the hierarchy with other properties that share the same identifier. In this case, the `width` of both labels is shared because they use the same identifier, `label_width`. When layout occurs, the largest `width` value of the two labels is used as the width for both of them.

```
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'bindingExample', function( context )
    local f = LrView.osFactory() -- obtain view factory
    local contents = f:column {  -- define view hierarchy
        spacing = f:control_spacing(),
        f:row {
            spacing = f:label_spacing(),
            f:static_text {
                title = "Name:",
                alignment = "right",
                width = LrView.share "label_width", -- the shared binding
            },
            f:edit_field {
                width_in_chars = 20,
            },
        },
        f:row {
```

```
            spacing = f:label_spacing(),
            f:static_text {
                title = "Occupation:",
                alignment = "right",
                width = LrView.share "label_width", -- the shared binding
            },
            f:edit_field {
                width_in_chars = 20,
            },
        },
    }
    local result = LrDialogs.presentModalDialog( -- invoke the dialog
        {
            title = "Dialog Example",
            contents = contents,
        }
    )
end )
```

Changing the contents of a view dynamically

This simple example of dynamic layout shows one set of controls and hides another set, based on the selected value in a pop-up menu. The dialog contains the popup and three views, each containing an alternate set of controls. When the user makes a selection in the pop-up menu, one of the views is shown, and the other two are hidden. For example:



This technique makes use of the `overlapping` placement style, and demonstrates *binding* of a property in one node to a property in another, so that changing one also changes the other.

The `overlapping` value for the `place` property causes all of the children of a node to be placed in the same space. The parent views are made big enough to enclose the largest child in any view, and all of the children are placed within that space.

If all of the children were visible at the same time, they would display on top of one another. To make sure only one view is visible at a time, we bind the `visible` value of each alternative view to a unique value of the pop-up menu. When the user makes the selection that has this value, the view bound to that value is shown, and the other views (bound to different values) are hidden.

- You only need to set the visibility of the parent view; when the parent is hidden, all of its child nodes are also hidden, regardless of their individual visibility settings.
- The `LrBindings.whenKeyEquals()` function sets `visible` to true only when the specified value of the bound property is set. You could choose to bind the true value to, for example, a logical OR or AND of several key values.

This example creates the overlapping views shown in the figure, where the controls shown below the format pop-up depend on the selection in the pop-up menu.

```
local LrBinding = import "LrBinding"
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrView = import "LrView"
LrFunctionContext.callWithContext( 'bindingExample', function( context )
    local f = LrView.osFactory()  -- obtain the view factory
    local properties = LrBinding.makePropertyTable( context ) -- make settings table
    -- add some keys with initial values
    properties.format = "jpeg"
    properties.jpeg_quality = 80
    properties.tiff_compression = "none"
    local contents = f:column { -- define the view hierarchy
        spacing = f:control_spacing(),
        bind_to_object = properties,  -- default bound table is the one we made
        f:popup_menu {
            items = {
                { title = "JPEG", value = "jpeg" },
                { title = "TIFF", value = "tiff" },
            },
            value = LrView.bind 'format', -- bind selection to the format key
        },
        f:column { -- place two views in the same space
            place = "overlapping",
            -- JPEG view
            f:view {
                -- shown only when format selection is JPEG
                visible = LrBinding.keyEquals( "format", "jpeg" ),
                margin = 3,
                f:row {
                    spacing = f:label_spacing(),
                    f:static_text {
                        title = "Quality:",
                    },
                    f:slider {
                        min = 0,
                        max = 100,
                        value = LrView.bind 'jpeg_quality', -- sets a JPEG value
                        fill_horizontal = 1,
                        place_vertical = 0.5,
                    },
                    f:edit_field {
                        width_in_digits = 3,
                        min = 0,
                        max = 100,
                        precision = 0,
                        value = LrView.bind 'jpeg_quality', -- sets a JPEG value
                    },
                },
            },
            -- TIFF view
            f:view {
                -- shown only when format selection is TIFF
                visible = LrBinding.keyEquals( "format", "tiff" ),
                margin = 3,
                f:row {
                    spacing = f:label_spacing(),
                    f:static_text {
```

```
                    title = "Compression:",
                },
                f:popup_menu {
                    items = {
                        { title = "None", value = 'none' },
                        { title = "LZW", value = 'lzw' },
                        { title = "ZIP", value = 'zip' },
                    },
                    value = LrView.bind 'tiff_compression',-- sets a TIFF value
                },
            },
        },
    },
}
local result = LrDialogs.presentModalDialog( -- invoke the dialog
    {
        title = "Dialog Example",
        contents = contents,
    }
)
end )
```

6 Writing a Web-engine Plug-in

This chapter describes the web-engine plug-in mechanism in the Lightroom SDK. This mechanism allows you to define new HTML web engines for the Web module. A web engine controls how a photo gallery is generated.

Web-engine plug-ins use a different architecture from standard plug-ins, and are not managed by the Plug-in Manager dialog. All available web engines appear in the Engine panel at the upper right of the Web module of Lightroom, including those predefined by Lightroom and any defined by plug-ins.

Your plug-in can also customize the control panels in the Web module, so that the controls map to user-customizable features of your own web engine.

Creating a web-engine plug-in

A web-engine plug-in consists of:

- A *manifest* file named `manifest.lrweb`, which maps LuaPage source files to the HTML output files that make up a photo gallery. This file uses a special command set; see "Web SDK manifest API" on page 130.
- An information file named `galleryInfo.lrweb`, which defines the *data model* and customized UI for your gallery type. See "Defining the data model" on page 119.
- One or more *web-page templates*, in the form of LuaPages; that is, HTML pages with embedded Lua code that is evaluated for display in the preview browser, or on publication, to generate dynamic content. See "LuaPage syntax" on page 136.
- Additional resources and supporting files, such as images, style sheets, string dictionaries for localization, and code files that define special behaviors.

Collect these files into a single folder, which you must place in the following directory according to your operating system:

IN MAC OS: *userhome*`/Library/Application Support/Adobe/Lightroom/Web Galleries/`

IN WINDOWS: *LightroomRoot*`\shared\webengines`

The name of the plug-in folder must end with `.lrwebengine`; for example, `myWebPlugin.lrwebengine`.

Folder contents

Here are the contents of a sample web-engine folder named `default_html.lrwebengine`:

Root	`default_html.lrwebengine/`
Template information	`manifest.lrweb` `galleryInfo.lrweb`

LuaPage templates	about.html detail.html foot.html grid.html head.html
Resources	resources/
style sheets	css/ie6.css ie7.css master.css
JavaScript	js/live_update.js
images	misc/icon_fullsize.png shadow-grid.gif shadow.png
Localization dictionaries	strings/ de/TranslatedStrings.txt en/TranslatedStrings.txt fr/TranslatedStrings.txt ja/TranslatedStrings.txt
Iconic preview	iconic_preview/flash_gallery_preview.swf
source files (not needed for delivery)	flash_gallery_preview.as flash_gallery_preview.fla

Defining the data model

The folder that defines your gallery type must define the data model in an information file named `galleryInfo.lrweb`. The file defines various parameters for the gallery, using this simple Lua-table format:

```
return {
    property_name_1 = "value string",
    property_name_2 = "value string",
    ...
}
```

Top-level property names are predefined, as shown in "GalleryInfo top-level entries" on page 120.

- The top-level `model` property is extensible, containing both predefined and plug-in-defined sections to create a complex data model. Sections are grouped, using brackets and dot notation to specify a complex property name. See "Data model entries" on page 121.
- The top-level `views` property is a function that customizes the user interface for your web engine, creating UI controls in the Web module control panels and binding them to the data model that your plug-in defines. See "Defining a UI for your model" on page 123.

GalleryInfo top-level entries

The following top-level properties are defined in the `galleryInfo.lrweb` file:

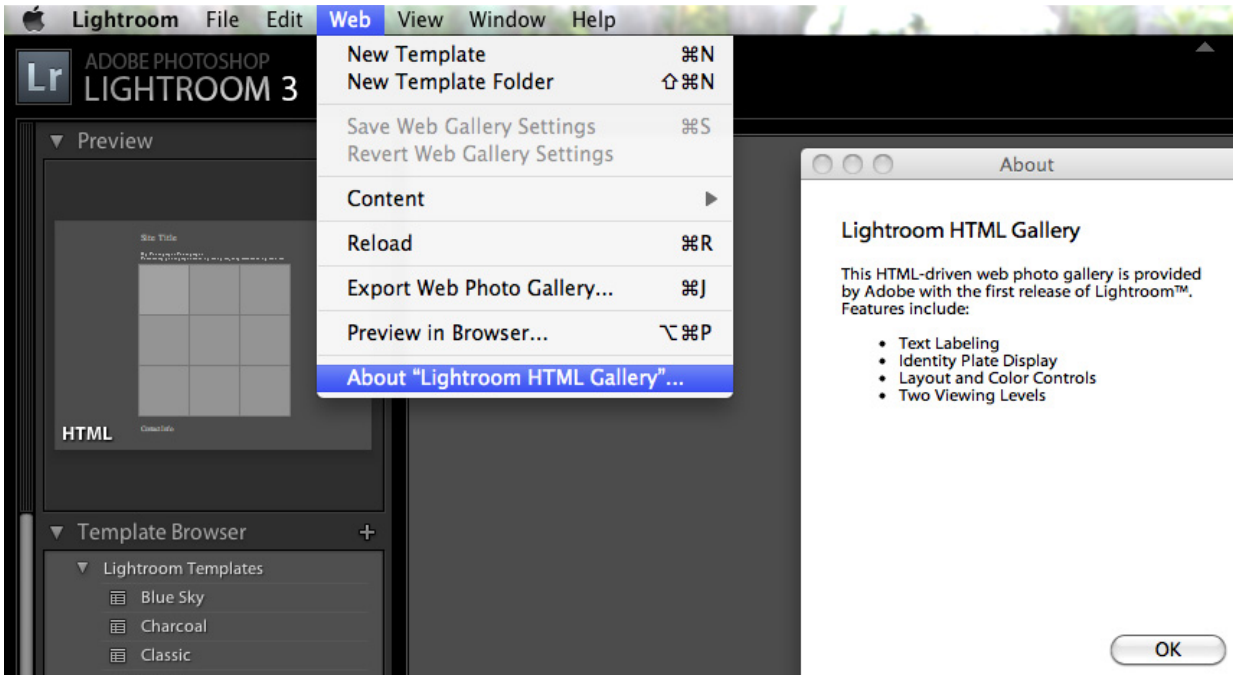
`title`	A localizable title string for the gallery type, which appears in the Web module's Engine list. You can localize the title string using the `LOC` function.
`id`	Each gallery must have a unique identifying string. By convention, use reverse-domain nomenclature, as for Java packages. For example, `com.myCompany.myDivision.myGallery`.
`galleryType`	The type of gallery. Currently only one type is allowed: • `"lua"` — An HTML gallery that uses Lua Server Pages.
`maximumGallerySize`	The maximum number of photos this gallery can reasonably display.
`model`	A table of user-configurable options for this web engine such as colors, labels, dimensions, image quality settings, grid row and column specifications, and so on. The keys in this table are strings that use dot-separated notation to break into separate areas; for example, `"model.nonDynamic.numRows"`. See"Data model entries" on page 121.
`views`	A function that returns a table of view descriptions by name, with entries for "labels", "colorPalette", "appearanceConfiguration", and "outputSettings". Each of these corresponds to a panel in the Web module, and each entry defines new UI controls to be added to that panel. See "Defining a UI for your model" on page 123.
`iconicPreview`	A table of information for controlling the live preview movie for a gallery in the Preview panel. See "Creating a preview" on page 128.
`aboutBoxFile`	The name of an HTML file to be displayed in the About box for this web engine. The file must be simple, self-contained HTML that does not reference any other resources (such as CSS or images). The About box is displayed when the user chooses **Web > About *[thisEngine]***.
`supportsLiveUpdate`	Boolean, true if this web engine supports the Live Update mechanism. See "Web HTML Live Update" on page 142.

This example shows the top-level entries from the `galleryInfo.lrweb` file of the built-in HTML gallery:

```
return {
    LrSdkVersion = 5.0,
    LrSdkMinimumVersion = 2.0, -- minimum SDK version required by this plugin
```

```
    title = LOC "$$$/AgWPG/Templates/HTML/Title=Lightroom HTML Gallery",
    id = "com.adobe.wpg.templates.jardinePro",
    galleryType = "lua",
    maximumGallerySize = 50000,
    aboutBoxFile = "about.html",
    supportsLiveUpdate = true,
    model = {...}
}
```

Here is the About box for the built-in HTML Web Gallery:



Data model entries

The `model` entry in the table returned by the `galleryInfo.lrweb` file defines the data model for your web engine. The `model` entry contains both predefined sections such as `photoSizes`, and plug-in-defined sections for local data, such as the one named `metadata` in the following example.

```
return {
    ...
    model = {
              [ "photoSizes.large.width" ] = 250,
              ...
              [ "photoSizes.thumb.width" ] = 130,
              ...
              ["appearance.textColor.color"] = "#166AF2",
              ["appearance.textColor.cssID"] = ".textColor",
              ...
              ["lightroomApplication.identityPlateExport"] = "(main)",
              ["lightroomApplication.jpegQuality"] = 70,
              ...
              ["metadata.siteTitle.value"] = LOC
                    "$$$/Templates/HTML/Defaults/props/SiteTitle=Site Title",
              ...
    }
```

```
}
```

Within the predefined `photoSizes` section, the size-class names (in this example, `large` and `thumb`) are defined by the plug-in. Within each size class, however, there are a set of predefined properties such as `width` and `height`.

Model properties can have simple number, string or color values, but to make a property dynamic, you can make the value a function definition. See "Creating a dynamic data model" on page 127.

These are the predefined properties for the `model` table:

`appearance.`*`cssClass`*`.`*`cssProp`*
 	
`appearance.`*`cssClass`*`.cssID`
	For an HTML gallery, entries in the `appearance` section are available in CSS form automatically, if they follow the correct convention. Specify each key is in this format:
	
	`[ "appearance.`*`cssClassName`*`.`*`cssPropertyName`*`" ] = `*`value`*
	
	In addition, for each unique name, you must also add an entry that tells Lightroom what CSS selector to use for that name, in this form:
	
	`[ "appearance.`*`cssClassName`*`.cssID" ] = `*`selectorName`*
	
	For example, to control the background color of all elements that belong to CSS class `myClass`, and provide an initial default value of red, add these two entries to the model:
	
	`[ 'appearance.myClass.background-color' ] = "#ff0000",`
	`[ 'appearance.myClass.cssID' ] = '.myColor',`

`photoSizes.`*`sizeClass`*`.`*`property`*
`width`
`height`
`maxWidth`
`maxHeight`
`metadataExportMode`
`tracking`
	These entries specify size classes of rendered JPEGs Lightroom should create. Some galleries might have only one size, such as a thumbnail-only gallery, but typically a gallery has layers with different image sizes (such as thumbnail, small, full-size). The size-class names are defined by your plug-in.
	
	For each named size class, you can specify properties; for example:
	
	`[ "photoSize.small.width" ] = 50,`
	
	Each size class has these properties:
	
	`width`, **`height`**—The size in pixels, as chosen by the user
	`maxWidth`, **`maxHeight`**—The largest allowed size in pixels
	`metadataExportMode`—What metadata to include, either 'copyright' or 'all'.
	`tracking`—Binds to the image-size slider in the control panel, so that the image can be resized interactively during preview. Lightroom sets this to true while resizing is in progress.

`lightroomApplication.`*`property`*
`identityPlateExport`
`jpegQuality`
`useWatermark`
`outputSharpeningOn`
`outputSharpening`
	These properties control how Lightroom behaves when creating the gallery:
	`identityPlateExport`—If your gallery can encorporate a PNG version of the identity plate, use the value "(main)".
	`jpegQuality`—The quality of the rendered JPEGs. Range is [0..100], where 100 is the best quality.
	`useWatermark`—True if rendered JPEGs should include a copyright watermark.
	`outputSharpeningOn`—True if rendered JPEGs should be sharpened. Default is true.
	`outputSharpening`—The type of sharpening, one of 1 (low), 2 (standard, the default), or 3 (high).

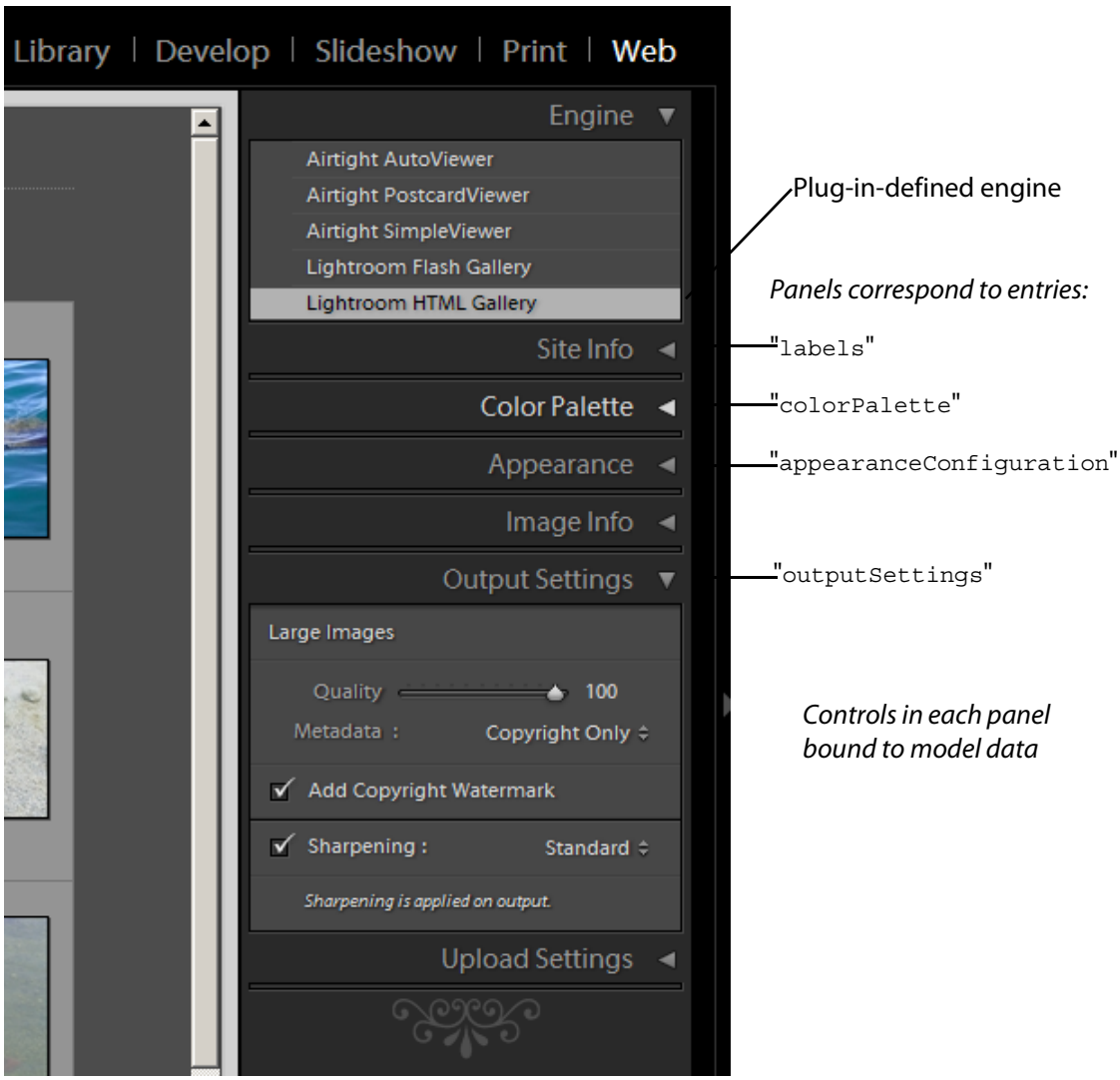
`perImageSetting.`*`property`* *`setting_names`*	A table that defines a per-image text description. Specifying one of these entries customizes the Image Info panel in the Web module. Each `perImageSetting` entry defines a checkbox, label, and edit text control, like those for the built-in per-gallery Title and Caption. Each table has these entries: **`title`**—The localizable display name of this setting, which appears in the label. **`value`**— Text to associate with each image. A string that can contain text-token placeholders in double curly braces. The user can edit this text. **`enabled`**— When true, the checkbox is checked. These values are stored as properties of `image.metadata`. Your template for the image details page can reference the text in order to display it with the image. For example, if you name a setting `description`, access the value using `$image.metadata.description.value`. See "Customizing per-image text" on page 126.
`plug-in-defined properties`	You can define additional properties to store gallery-wide text labels; for example, a site title or collection title. You can also define properties to store appearance parameters that control the look of the gallery, but do not work through CSS; for example, how many rows or columns in the grid, or color properties that need to be accessed dynamically by JavaScript. The names of such properties are defined by your plug-in for its own use.

Defining a UI for your model

The `views` entry in the table returned by the `galleryInfo.lrweb` file defines the user interface for your web engine. It is a function that is passed two arguments, a `controller` (which is an observable table that contains your model data) and a `webViewFactory` object that allows you to create and populate UI elements (as described in Chapter 5, "Creating a User Interface for Your Plug-in.")

The function returns a table of view descriptions by name, with entries that correspond to the control panels at the right of the Web module.

`labels`	The Site Info panel, which allows users to specify text to be associated with the site.
`colorPalette`	The Color Palette panel, which allows users to adjust the colors of various elements of the site.
`appearanceConfiguration`	The Appearance panel, which allows users to adjust the appearance of individual photos.
`outputSettings`	The Output Settings panel, which allows users to adjust various output parameters such as image quality and metadata inclusion.



Within each entry, you can use the view factory object to create UI controls. Set the bound table to be the `controller` table, and bind control values to data values you have defined in your model.

Here is an example of the format of the `views` function for your web engine:

```
return {
    ...
    views = function( controller, f )
        local LrView = import "LrView"
        local bind = LrView.bind
        local multibind = viewFactory.multibind

        return {
            labels = f:panel_content { -- returned item identifies panel
                bind_to_object = controller, -- bound table is passed controller
                f:subdivided_sections {
                    f:labeled_text_input  { -- create controls in the sections
                        title = "Site Title",
                        value = bind "metadata.siteTitle.value", -- bind to model data
                    },
                ...additional content...
            },
            colorPalette = f:panel_content {
                bind_to_object = controller,
                ...define content...
            },
            appearanceConfiguration = f:panel_content {
                bind_to_object = controller,
                ...define content...
            },
            outputSettings = f:panel_content {
                bind_to_object = controller,
                    ...define content...
                },
            },
        }
    end,
    ...
}
```

Using web view factories

Notice that the view factory passed to `views` function is an extension of the standard view factory described in Chapter 5, "Creating a User Interface for Your Plug-in." It is an object of type `LrWebViewFactory`, and it defines these additional functions for creating UI content suitable to the Web module (see the Lightroom SDK API documentation for details):

`panel_content`	Creates a top-level panel in the Web module, which contains sections divided by heavy black lines.
`subdivided_sections`	Creates a section within a panel in the Web module. Within the section, control rows and columns are separated by light gray lines.
`header_section_label`	Creates a text label for a section within a panel, with suitable formatting.

`content_column`
`slider_content_column`
`checkbox_and_color_row`
`color_content_column`
`content_section`
`header_section`
	These create column-style containers for controls within a section. Some are generic, and some are specialized to particular types of row content, with suitable formatting.

`row`
`popup_row`
`slider_row`
`checkbox_and_color_row`
`label_and_color_row`
`checkbox_row`
`labeled_text_input`
	These create row-style containers within column-style containers. Some are generic, and some contain specific sets of controls, with suitable formatting.

`metadataModeControl`
`warning_icon`
`identity_plate`
`row_column_picker`
	These create individual controls of types appropriate to web-engine usage. These can be placed in unspecialized column or row containers.

Customizing per-image text

The Image Info panel allows the user to specify text for use in gallery pages. Each text label is named with a *label*, such as Caption or Title, and can be enabled with a checkbox. A menu of preset values (the *custom settings* menu) allows the user to get dynamic text from the current image's metadata. The presets can be further customized using the Text Template Editor, which allows users to define text that incorporates dynamic values from metadata. The user can also save customized text templates as new presets.

Your page templates can access the user's text choices using this syntax:

You can use the `model` entry `perImageSetting` to add text labels to your model. Each setting is identified by a property name that you define. Each setting adds a row of controls to the Image Info panel, which allows the user to choose the text value of that label.

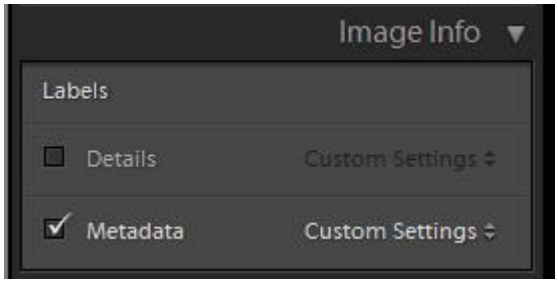
Your page templates can access the user's text choices for it using this syntax:

```
$image.metadata.propertyName
```

For example, the following defines a simple per-image description and title:

```
model = {
    ...
    ["perImageSetting.details"] = {
       enabled = true,
       value = "Default Custom-Text value",
       title = LOC "$$$/WPG/HTML/CSS/properties/ImageDetails=Details",
       },
    ["perImageSetting.datatext"] = {
       enabled = true,
       value = "Default Custom-Text value",
       title = LOC "$$$/WPG/HTML/CSS/properties/ImageData=Metadata",
    },
}
```

This definition creates these controls in the Image Info panel:



The localized `title` text appears as the display name for the label. The checkbox and presets menu are supplied by Lightroom. The value of the `value` entry appears as the default value for the Custom Text preset choice.

To incorporate the user's choice of text in the image-detail template page, use code like this:

```
<html>
    <body>
        <lr:ThumbnailGrid>
            <lr:GridPhotoCell>
                <pre>
                    $image.metadata.datatext, $image.metadata.details
                </pre>
            </lr:GridPhotoCell>
            <lr:GridRowEnd>
                <br>
            </lr:GridRowEnd>
        </lr:ThumbnailGrid>
    </body>
</html>
```

Localizing the UI

Strings that appear in the UI, either in predefined LIghtroom controls or menus, or in those you define, can be localized using the `LOC` function, as described in Chapter 7, "Using ZStrings for Localization.

The `LOC` function looks up localized values in string dictionaries; your plug-in must supply these as part of the plug-in folder. To add string dictionaries to your plug-in, create a `strings` resource folder in your main plug-in folder, and name the subfolders with the appropriate language codes. For example:

```
myWebPlugin.lrwebengine/strings/de/TranslatedStrings.txt
myWebPlugin.lrwebengine/strings/fr/TranslatedStrings.txt
...
```

Localization occurs when the user publishes the gallery. To get different language versions, the user must run Lightroom in the desired locale, and publish another version of the gallery.

Creating a dynamic data model

Model properties can have simple number, string or color values, but to make a property dynamic, you can make the value a function definition. When it needs to access the property, Lightroom executes the function and the result is returned as the property value. The evaluation context makes your data model available in the global scope.

A typical use of dynamic data is to tie two properties together, so that changing one changes the other. For example, you might want to control the aspect ratio by making `photoSizes.mySize.width` to be equal to `photoSizes.mySize.height`. To do this, you can use a function definition as the value of one of the properties. For example:

```
["photoSizes.large.height"] = function() return photoSizes.large.width end,
["photoSizes.large.width"] = 450,
```

This function simply accesses and returns the value of another property. You can, however, define a function to perform some transformation of the related value. You can, for instance, add formatting and logic using Lua's basic math and string manipulation functions. Lightroom also provides a function, `LrColorToWebColor`, that converts an `LrColor` object to a string representation suitable for use in CSS.

Creating a preview

The *iconic preview* for a web engine is an SWF movie of a gallery that can be shown in the "Preview" panel of the Web module. It presents a dynamic preview of the gallery using icons, rather than full-size images.

You can use the `iconicPreview` top-level gallery-info entry to specify how the iconic preview for your gallery should be implemented. This entry references a simple Flash® movie (which you must implement and include in the web-engine folder) that renders an "iconic" representation of each web page, in order to convey the general look of the web gallery in a simple, stylized form.

The `iconicPreview` entry is a table with two entries:

`flashMovie`	A string value, the relative path from the root web-engine folder to a compiled Flash movie (an SWF file).
`flashvars`	A function that returns a table of values to be used in the Flash movie. These are the model properties that you want represented in the iconic preview. Each entry in this table is available at the `_root` level of the Flash movie environment. Numbers and string values are passed through without any conversions. `LrColor` objects are converted to a string representation that is easy to parse in ActionScript™.

Here is an example of an `IconicPreview` entry in the `galleryInfo.lrweb` file:

```
return {
    ...
    iconicPreview = {
        flashMovie = "iconic_preview/flash_gallery_preview.swf"
        flashvars = function()
            local iconicData = {
                foregroundColor = appearance.textColor.color,
                showLogo = appearance.logo.display,
                cellBorderColor = nonCSS.cellBorderColor,
                cellRolloverColor = nonCSS.cellRolloverColor,
                cellBackgroundColor = nonCSS.cellColor,
                bodyBackgroundColor = appearance.body[ "background-color" ],
                numRows = nonCSS.numRows,
                numCols = nonCSS.numCols,
                }
            return iconicData
        end,
    ...
}
```

Creating the Flash movie

Your web-engine folder must include a simple Flash movie that renders the "iconic" representation of each web page. The ActionScript file that defines your movie can access the global variables provided by the `flashvar` entry in `iconicPreview`, at the top level of the `_root` object.

Your Flash movie should do these things:

1. Set the stage behavior:

   ```
   Stage.scaleMode = "noScale";
   Stage.align = "tl";
   ```

   This ensures that your preview renders without stretching or distortion.

2. Create an external interface callback called `ready`, which Lightroom will poll waiting for your preview to finish drawing:

   ```
   _root.ready = 'no';

   _root.readyFunc = function(str:String) {
     return _root.ready;
   }
   ExternalInterface.addCallback("ready", _root, _root.readyFunc);
   ```

   Then at a later time (usually in a subsequent frame):

   ```
   _root.ready = 'yes';
   ```

   Once your ready function returns `"yes"`, Lightroom takes a screenshot of the Flash movie and terminates its execution (in order to reduce CPU usage)

3. Initialize default values, so that you can preview your movie without running it in Lightroom.

   ```
   var numCols;
   if( _root.numCols != null ) {
       numCols =parseInt( _root.numCols );
   }
   else {
       // default value
       numCols = 4;
   }
   ```

   Do this for each model property you are using in your preview.

4. Draw the preview. This can be done by rearranging existing objects that you created in Flash, or simply by using the drawing primitives of the ActionScript programming language. For example:

   ```
   var cellSize = 10;
   for( x = 0; x < numCols; x++ ) {
       _root.beginFill( cellColor, 100 );
       _root.moveTo(x*cellSize, y* cellSize );
       _root.lineTo( (x+1)* cellSize, y* cellSize );
       _root.lineTo( (x+1)* cellSize, (y+1)* cellSize );
       _root.lineTo( x* cellSize,+ (y+1)* cellSize );
       _root.endFill();
   }
   ```

   This draws as many rectangle as are specified in the `numCols` Flash variable.

Web SDK manifest API

The *manifest* is a Lua file in your plug-in's root directory named `manifest.lrweb`. It maps LuaPage source files and template files to web-engine HTML output files using a set of commands for different kinds of pages and resource files.

Define the mapping using these commands:

AddPage	Maps one source LuaPage file from the gallery template directly into the published gallery.
AddResource AddResources	Maps one resource file or a set of resource files from the gallery template directly into the published gallery. Resources can include image or icons, string dictionaries for localization, JavaScript files, and so on.
AddPhotoPages	Uses a LuaPage template to build a page for each photo in the current Lightroom selection.
AddGridPages	Uses a LuaPage template to build a page for each grid of photos in the current Lightroom selection.
AddCustomCSS	Generates a CSS file using the `appearance` properties of your data model.
IdentityPlate	Exports an identity plate as a PNG file.
importTags()	Adds custom tagsets to your gallery.

AddPage

Maps one source LuaPage file from the gallery template into the published gallery. The source file is interpreted by the LuaPage engine, resulting in an HTML file in the published gallery.

Inputs

`filename`	The path to which to write the file in the published gallery.
`template`	The path to the source LuaPages file, relative to the folder containing this manifest.

Example

```
AddPage {
    filename = "content/pages/myWebPage.html",
    template = "myWebPage.html",
}
```

AddResource

Maps one resource file from the gallery template directly into the published gallery. A resource is not interpreted, but is simply copied directly.

Inputs

`source`	The path to the resource file, relative to the gallery template.
`destination`	Optional. The path to the published gallery to which to copy the resource. By default, the destination path is the same as the source path.

Example

```
AddResource {
    source = "image.png",
    destination = "content/resources/image.png",
}
```

AddResources

Copies a set of resource files from the gallery template directly into the published gallery. A resource is not interpreted, but is simply copied directly.

Inputs

`source`	The path to the resource folder, relative to the gallery template.
`destination`	Optional. The path to the published gallery to which to copy the resources. By default, the destination path is the same as the source path.

Example

```
AddResources {
    source = "resources",
    destination = "content/resources",
}
```

Alternative syntax

Instead of passing a table of named arguments, pass a single string to be used as the source:

```
AddResources "resources"
```

AddPhotoPages

Uses a LuaPage template to build a separate page for each photo in the current Lightroom selection.

Inputs

`filetype`	Optional. A file extension for the pages. Default is `"html"`.
`variant`	Optional. A suffix to append to the file name. Useful if your gallery has several sizes of pages for each photo. Default is the empty string.
`destination`	The path to the published gallery to which to write the pages.
`template`	The path to the source LuaPage file.

LuaPages environment variables

When executing the LuaPages for `AddPhotoPages`, the following variables are defined in the environment

`filename`	The file name of the current page.
`root`	The relative path to the root of the gallery.
`gridPageLink`	If grid pages have been added to the gallery, the relative path from this page to the corresponding grid page that contains this photo.
`pageType`	The page type, `"photo"`.
`index`	The index position of the photo within the gallery.

Example

```
AddPhotoPages {
    template = 'detail.html',
    variant = '_large',
    destination = "content",
}
```

AddGridPages

Uses a LuaPage template to build a page for each grid of photos in the current Lightroom selection.

Inputs

`filetype`	(Optional) A file extension for the pages. Default is `"html"`.
`destination`	The path to the published gallery to which to write the pages.
`template`	The path to the source LuaPage file.
`rows`	The number of rows in each grid.
`columns`	The number of columns in each grid.

LuaPages environment variables

When executing a LuaPage specified with `AddGridPages()`, the following variables are defined in the environment

`filename`	The file name of the current page.
`pageType`	The page type, `"grid"`.
`page`	The position index of the current page among grid pages defined in the gallery.

In addition, if you use `AddGridPages()` to add any page, all of the LuaPages in your gallery can use these environment variables:

`numGridPages`	The number of grid pages in the gallery
`filenameOfGridPage( pageNumber )`	A function that takes a grid page number and returns the file name string for that page.
`gridPageForPhotoAtIndex( photoIndex )`	A function that take a photo index position and returns the file name for the grid page containing the photo.
`rows`	The number of rows on the grid pages.
`columns`	The number of columns on the grid pages.

Example

```
AddGridPages {
    destination='content',
    template='grid.html',
    rows=model.nonDynamic.numRows,
    columns=model.nonDynamic.numCols,
}
```

AddCustomCSS

Generates a CSS file using the `appearance` properties defined in your data model.

When you declare your data model in the `galleryInfo.lrweb` file, this command exports to CSS all entries that begin with "`appearance.`".

Inputs

`filename`	(string) The path and name of the output file.

Example

To specify the background color of the body using CSS, you need a declaration like this:

```
/* Desired CSS output */
body {
    background-color: #ff0000,
}
```

To make your web engine generate this:

1. Declare the intention to emit CSS in the manifest file, `manifest.lrweb`:

```
AddCustomCSS {
    filename='content/custom.css',
}
```

2. Define the required data model entries in the information file (`galleryInfo.lrweb`):

```
return {
    ...
```

```
    model = {
        ...
        ["appearance.body.background-color"] = "#ff0000",
        ["appearance.body.cssID"] = "body",
        ...
     },
    ...
}
```

3. To make this something the user can edit, add a corresponding control to one of the panel descriptions in the `views` section of your information file (`galleryInfo.lrweb`):

```
return {
    ...
    views = {
        ...
        myViewFactory.label_and_color_row {
        bindingValue = "appearance.body.background-color",
        title = "Background",
        },
        ...
    ...
}
```

IdentityPlate

Exports an identity plate as a PNG file, if the user chooses to use it.

During a Lightroom preview of the web gallery, the PNG file is always generated, to support a live update of the model-defined property that controls identity-plate use. If the user chooses not to use an identity plate, however, the PNG is not exported as part of any export, upload, or preview-in-browser operation.

Inputs

`destination`	(string) The path to the published gallery to which to write the image file.
`enabledBinding`	(string) The plug-in-defined entry in the data model that controls whether to export the identity plate.

Example

The `appearance` section of the model defines a `logo.display` properties:

```
["appearance.logo.cssID"] = ".logo",
["appearance.logo.display"] = false,
```

The `views` section of the model binds the property to the `identityPlateExport` checkbox (defined by the Lightroom application):

```
myViewFactory:identity_plate {
    value = bind "lightroomApplication.identityPlateExport",
    enabled = bind "appearance.logo.display",
    },
```

In the manifest, use the `IdentityPlate` command to enable this binding:

```
IdentityPlate {
    destination='content/logo.png',
    enabledBinding = [[appearance.logo.display]],
}
```

When the user selects or deselects the checkbox, this binding causes the corresponding model property (`logo.display`) to be set to true or false, and thus the corresponding CSS property (`.logo`) to be set to the correct image (`logo.png`), or to `none`. If the user does not choose to export the identity plate, the file `content/logo.png` is not generated on upload.

importTags()

Adds custom tagsets to your gallery (see "Web SDK tagsets" on page 138). This is a function which takes two ordered parameters:

`prefix`	(string) A short prefix used to identify tags belonging to this tagset. For example, "lr".
`tagsetPath`	(string) A path to the tagset definition file. The special path "com.adobe.lightroom.default" loads the default tagset.

Example

1. Create a tagset file called `myTags.lrweb`.:

```
tags = {
    fancyQuote = {
        startTag = 'write( [[<blockquote\nstyle="margin: 0 0 0 30px; '
        .. 'padding: 10px 0 0 20px; font-size: 88%; line-height: 1.5em; '
        .. 'color: #666;">]] )',
        endTag = 'write( [[</blockquote>]] )',
    }
}
```

2. In your `manifest.lrweb` file, import the tagset definition file by name:

```
importTags( "xmpl", "myTags.lrweb" )
```

3. Use that tagset in any LuaPages file, identifying each defined tag with the specified prefix:

```
<xmpl:fancyQuote>
A wise man once said:<br>
Don't count your chickens before they're hatched
</xmpl:fancyQuote>
```

4. This results in the following in the HTML output:

```
<blockquote
    style="margin: 0 0 0 30px; padding: 10px 0 0 20px; font-size: 88%;
        line-height: 1.5em; color: #666;">
    A wise man once said:<br>
    Don't count your chickens before they're hatched
</blockquote>
```

LuaPage syntax

A LuaPage is a Lua-language source file that is evaluated to produce one destination web page in your published gallery. In the manifest, use the AddPage command to map each source LuaPage to a destination file location.

Environment variables available to LuaPages

A LuaPage is evaluated in a context that provides many of the Lua functions of a default Lua installation, and also some special functions that are specific to the web photo gallery templating language.

These read-only variables are available:

Variable	Description
`getImage(imageIndex)`	A function that returns an imageProxy.
`mode`	• When the gallery is being previewed inside Lightroom, the value is the string "preview". • During export, upload or preview in browser, it is set to "publish".
`numImages`	The number of photos in the gallery.
`string` `math`	Standard Lua namespaces.
`table`	A subset of the default Lua `table` namespace; contains the `insert` function.

`ipairs`
`pairs`
`type`
`tostring`
	Standard Lua functions

`LOC`	Text values can be localized. Use this function as a string value in order to specify a string by a unique identifier; your plug-in must provide a string dictionary in which to look up the display-string value for the current system language. See "Localizing the UI" on page 127."
`LrTagFuncs`	A table of private helper functions for the `lr:` tags

Variable	Description
`includeFile()`	An execution-time function that allows a page to include another file using runtime logic to specify which file. For example: `<html>` ` <body>` `  <h1>My web gallery</h1>` `  This is the grid.html file.` `   ` `  Include a file using the include directive:` `  <%@ include file="subdir/foo.html" %>` `   ` `  Include same file using the includeFile command:` `  <% includeFile( 'subdir/foo.html' ) %>` `   ` `  Include a file that includes another file:` `  <% includeFile( 'file1.html' ) %>` ` </body>` `</html>`

LuaPage data types

These data types are defined:

imageProxy	An object that has these properties:
`exportFilename`	(string) The base name string of a JPEG that will be written to disk for this photo.
`rating`	(number) The numeric rating for the image, or nil if it has no rating
`imageID`	(string) The `id_global` string for this image.
`renditions`	(array of object) An array of imageRendition objects for the renditions of this photo.
`metadata`	(table) A table of metadata settings, based on the `perImage` settings.
`colorLabel`	(string) The localized text for a photo's `colorLabel`, such as "red", or nil.

imageRendition	An object that represents an image rendition for a photo. It has these properties:
`width`	(number) The width in pixels.
`height`	(number) The height in pixels.
`relPath`	(array of string) An array of directory names, in which the last entry is the file name.
`dir`	(array of string) An array of directory names.

`cropMode`	How to size the image. One of: `minimum` — Fits the image within the`photoSize` dimensions. `maximum` — Scales the image to be at least as big as both `photoSize` dimensions.
`metadataExportMode`	How to export metadata. One of: `copyright` — Export only copyright information. This is equivalent to the "Minimize Metadata" option in the Export dialog `all` — Export all metadata.

Web SDK tagsets

A tagset is an external file containing macro-like definitions that can be loaded by your web pages. These are similar to JSP Tag Libraries, but simpler. They allow you to extract common content and logic that appears on multiple pages into a custom set of tags. Once defined and imported, you can use the tags just like regular HTML tags.

At run time, your LuaPage, replaces the tag with its Lua-language tag definition, which it then compiles and executes to product the HTML output.

There is built-in set of tags included with the Lightroom SDK, which you can also include and use in your LuaPages.

Defining custom tags

To define a tagset in Lua, specify a `tags` table. This is a table of tables, where each element table defines one tag. The first element is the tag's unique name, and the value is a table containing a `startTag` and `endTag` element:

```
tags = {
    tagName = {
        startTag = "macroCode",
        endTag = "macroCode",
    },
}
```

The value of the `startTag` and `endTag` element is a string containing Lua code. It can use global functions and constants defined in the same page using a `globals` table. This is again a table in which each element defines one function or constant:

```
globals = {
    functionName = function( x )
        _body of function_
    end,
}
```

When the LuaPage is evaluated, the code for each tag is evaluated, and the result is substituted for the opening or closing named tag.

For example, you could define code in this format in your tagset file:

```
globals = {
    myOpenTagFunction = function( )
        --body of function
```

```
    end,
    myCloseTagFunction = function( )
        --body of function
    end,
}

tags = {
    myTag = {
        startTag = "myOpenTagFunction()",
        endTag = "myCloseTagFunction()",
    }
}
```

If you import this tag into the `xmpl` namespace, your LuaPage would reference the tag like this:

```
<xmpl:myTag>Helloworld</xmpl:myTag>
```

At run time, when the LuaPage is evaluated, the tags are replaced with the Lua code, and the contents is simply written out:

```
myOpenTagFunction() write( [[Helloworld]] ) myCloseTagFunction()
```

This code is then evaluated to produce the final HTML for your web gallery page.

Using custom tags

To use a tagset that you have defined in your web-engine plug-in:

1. Include the tagset definition file or files in the root directory of your web engine.
2. Add a line to import the tagset in your `manifest.lrweb` file:

   ```
   importTags( "lr", "pathToTagsetFile" )
   ```

   This includes all of the tags defined in the file under the namespace `lr`. The namespace definition prevents conflicts with tags of the same name defined in other tagset libraries. You can use any namespace for your own tags.
3. To load the built-in default tagset, substitute the special value "com.adobe.lightroom.default" for the path:

   ```
   importTags( "lr", "com.adobe.lightroom.default" )
   ```

   By convention, the built-in tags are imported into the `lr` namespace.
4. To use the defined tags in your LuaPages, use the namespace prefix for both the opening and closing tag. For example:

   ```
   <lr:ThumbnailGrid>...</lr:ThumbnailGrid>
   ```

Custom tag example

Here is an example that simply wraps some constant text around the text specified as the content of the tag:

1. Define the tag and its supporting function in the tagset file, `myTags.lua`:

   ```
   globals = {
       myFn = function( x )
   ```

```
            write( "You said, \"" )
            x()
            write( "!\"" )
        end,
    }
    tags = {
        exclaim {
            startTag = "myFn( function()",
            endTag = "end )",
        }
    }
```

2. Use the importTags() command in your web SDK manifest (`manifest.lrweb`) to import this into the "`xmpl`" namespace:

```
importTags( "xmpl", "myTags.lua" )
```

3. Reference the tag in a LuaPage source file:

```
<xmpl:exclaim>Helloworld</xmpl:exclaim>
```

4. When the LuaPage file is converted into Lua code, this becomes:

```
myFn( function() write( [[Helloworld]] ) end )
```

5. When the Lua code is executed, this produces text as its HTML output:

```
You said, "Helloworld!"
```

Lightroom built-in tagset

Lightroom includes a default set of tags, defined in the `"com.adobe.lightroom.default"` tagset.

This tagset is typically imported into the `lr:` tagset namespace, but you can import it into any namespace using the importTags() command of your web SDK manifest. As for all imported tags, you must reference each opening and closing tag name with the namespace prefix. For example:

```
<lr:ThumbnailGrid>
    ...
</lr:ThumbnailGrid>
```

The built-in tagset defines two groups of tags, for building thumbnail grids, and for defining navigation properties of a multi-page gallery.

Thumbnail grid tags

Use these tags to build a grid. This set of tags simplifies assembly of repeating units based on rows, columns, and the photo selection. You can use these tags only on pages that you specify in the manifest with AddGridPages.

The `ThumbnailGrid` tag is a container for the other tags, which define cells within the grid. For example:

```
<lr:ThumbnailGrid>
    <lr:GridPhotoCell>
        <img src="thumbs/<%= image.exportFilename %>.jpg" >
    </lr:GridPhotoCell>
</lr:ThumbnailGrid>
```

This defines a simple grid with only one cell, which displays a photo from the referenced file. It uses a variable, `image`, which is evaluated at run time as a reference to the currently selected photo.

The following local variables are available in the context of the `ThumbnailGrid` tag:

Variables available for grids	
`cellIndex`	Contains the 1-based index for the current cell in the grid.
`row`	Contains the 1-based row number for the current cell.
`column`	Contains the 1-based column number for the current cell.
`image`	Contains the image proxy. This is a complex data type defined within the LuaPage environment; see "LuaPage data types" on page 137.

The following grid tags are defined:

Grid tags	
`ThumbnailGrid`	Provides the definition of a thumbnail grid for pages in your gallery. Contains the remaining tags as children.

`GridPhotoCell`	Defines content to be repeated for each cell. Contained in a `ThumbnailGrid` tag; for example:
	`<lr:ThumbnailGrid>`
	`  <lr:GridPhotoCell>`
	`  <img src=`
	`  "$mypath/thumb/<%= image.exportFilename %>.jpg"`
	`  id="<%= image.imageID %>" class="thumb" />`
	`  </lr:GridPhotoCell>`
	`</lr:ThumbnailGrid>`

`GridEmptyCell`	Optional. Defines an empty cell in the grid.
`GridRowEnd`	Optional. Defines content to be placed at the end of each row.
`GridRowStart`	Optional. Defines content to be placed at the start of each row.

Pagination tags

This set of tags can be used to add page navigation buttons to your HTML pages. Predefined page-navigation buttons include one for the current page, one for direct access to other pages, and ones for the next and previous page, which can be disabled for the first and last pages. You can associate your own text or destination with each type of button. For example:

```
<% if numGridPages > 1 then %>
    <div class="pagination">
        <ul>
            <lr:Pagination>
                <lr:CurrentPage>
                    <li>$page</li>
                </lr:CurrentPage>
                <lr:OtherPages>
                    <li><a href="$link">$page</a></li>
```

```
            </lr:OtherPages>
            <lr:PreviousEnabled>
               <li><a href="$link">Previous</a></li>
            </lr:PreviousEnabled>
            <lr:PreviousDisabled>
               <li>Previous</li>
            </lr:PreviousDisabled>
            <lr:NextEnabled>
               <li><a href="$link">Next</a></li>
            </lr:NextEnabled>
            <lr:NextDisabled>
               <li>Next</li>
            </lr:NextDisabled>
         </lr:Pagination>
      </ul>
   </div>
<% end %>
```

The following local variables are available in the context of the `Pagination` tag:

Variables available for pagination	
`page`	Contains the appropriate page number. • Within a `CurrentPage` tag, this is the current page number. • Within an `OtherPages` tag, this is the number of the corresponding page.
`link`	Contains the URL to an appropriate page for a navigation button. For example, within a `PreviousEnabled` tag, the URL of the previous page.

The following pagination tags are defined:

Pagination tags	
`Pagination`	Provides the definition of pagination properties for pages in your gallery. Contains the remaining tags as children.
`CurrentPage`	Defines an icon or text for the current page.
`OtherPages`	Defines an icon or value with which to navigate directly to other pages.
`PreviousEnabled`	Defines an icon or value with which to navigate to the previous page.
`PreviousDisabled`	Defines an icon or value for the previous-page button for the first page (the case in which there is no pervious page).
`NextEnabled`	Defines an icon or value with which to navigate to the next page.
`NextDisabled`	Defines an icon or value for the next-page button for the last page (the case in which there is no next page).

Web HTML Live Update

When you preview your web gallery in Lightroom, Lightroom opens a web browser which runs independently of the main application. If, during the preview, you make changes to the gallery parameters, Lightroom must communicate with the web browser in order to reflect those changes.

While a page from your gallery is being previewed in Lightroom, a user might change a model variable using the control panel. In order to reflect the change in the previewed page, the Lightroom browser normally needs to reload the page. Lightroom clears all cached copies of the page, tells the browser to reload, and builds new HTML and CSS files in response to the browser's reload request. This process is time consuming, and can cause changes in color or other visually startling changes as the page loads. For a change as simple as a nudge in hue of a color slider, you might find this response unacceptably jarring.

Live Update is intended to avoid browser reload, which disrupts the user experience. Live Update is a mechanism by which a web engine can intercept and prevent the reload operation, using DHTML/AJAX scripting techniques to alter the web page in place. DHTML/AJAX use JavaScript, which is executed in the context of the built-in web browser (rather than the Lua scripting environment of Lightroom in general).

An HTML page in your web engine can incorporate JavaScript that uses Live Update to interact with Lightroom during a preview. This communication operates in both directions:

- Lightroom sends messages to the page, making `liveUpdate()` JavaScript function calls into the page whenever the user alters a parameter in the gallery data model. If the call is successful, Lightroom does not request a page reload.
- The page contains JavaScript that sends messages back to Lightroom in response to user events, such as a request for a text field edit, or to override a data model value.

In order to enable this functionality, your plug-in must contain JavaScript implementations of the `liveUpdate()` functions and the event-handler callbacks. There is a sample implementation in the file `live_update.js`, which you can use or modify. It is part of the sample plug-ins provided with the SDK.

To include the JavaScript file that implements Live Update in your pages, use a line such as this in your `header.html` template file:

```
<script type="text/javascript" src="$theRoot/resources/js/live_update.js">
```

Defining messages from Lightroom to a previewed page

To implement a message from Lightroom to your HTML web gallery, you define JavaScript live-update functions as properties of the JavaScript `document` object. There are two kinds of live update messages for different kinds of live update operations:

- The document.liveUpdate function handles changes that involve gallery appearance (such as CSS properties) and text labels
- The document.liveUpdateImageSize function handles changes that involve gallery image size.

Implementation of either of these functions is optional. For simple galleries, the reload solution may be adequate. If you do not add any Live Update functions to your `document` object, Lightroom uses the default reload behavior.

Returning values from live-update functions

When a change affects a page that the browser has previously cached, Lightroom must ensure that the browser reloads that page, rather than displaying the cached version. Lightroom also maintains a cache,

which may need to be cleared. Your live-update function signals Lightroom about what behavior to use by returning one of these strings:

- **`invalidateOldHTML`**: The browser cache is cleared, and all of the HTML pages in Lightroom's page cache are cleared. The exported JPEGs remain unchanged. The reload is deferred until the user navigates away from the currently previewed page.

  Return this value if the update is successful, and the change affects only the current HTML page

- **`invalidateAllContent`**: The browser cache is cleared, and all of Lightroom's page caches (both HTML and resource files) are cleared. The exported JPEGs remain unchanged. The reload is deferred until the user navigates away from the currently previewed page.

  Return this value if the update is successful, and the change affects any referenced file, such as a JavaScript or CSS file. This is typically the default case.

- **`failed`** (or any other return, or throwing an exception): Causes immediate reload, and clearing of both browser and Lightroom page caches. The exported JPEGs remain unchanged.

  Return this value if your function is unable to update the page. Lightroom then commands the embedded browser to reload the original page.

document.liveUpdate

Your HTML gallery can implement this function to respond to a change made in Lightroom to the appearance (CSS styling), or a Lightroom update to the fixed strings for the gallery. (Do *not* use it for changes to strings associated with a particular image, such as the image name or other metadata; a change of this kind always causes a reload.)

Your function manipulates the web page objects using JavaScript calls; typically, it locates the document node and alters the page appearance or content to reflect the change made to the data values. The function should return a result that indicates whether the update was successful.

The prototype is:

```
document.liveUpdate = function( path, newValue, cssId, property ) {
    var result = "failed";
    // JavaScript implementation goes here
    return result;
}
```

`path`	A dot-separated path to the node in the `appearance` portion of your `galleryInfo.lrweb` file.
`newValue`	The new value (such as `"ffffff"` for the color white).
`cssId`	The corresponding `cssId` for the node (such as `'body'`).
`property`	The CSS property that is changing on this node (such as `'margin'`).

document.liveUpdateImageSize

Implement a separate function for live update of image size. This function is called repeatedly while the mouse is held down on the image size slider. As soon as the mouse is released, a full reload of the page occurs, flushing all caches and invalidating all JPEGs.

This function must locate the image using `document.getElementById()`, and set its dimensions using appropriate DOM methods.

The function has this prototype:

```
document.liveUpdateImageSize = function( imageID, width, height ) {
    // your code here
    return "invalidateAllContent";
}
```

imageID	The unique identifier of the image.
width	The new image width in pixels.
height	The new image height in pixels.

If unable to perform the live update, return `"failed"`. In this case, Lightroom reloads the browser as often as it can while the mouse is dragged.

Example live-update implementation

The Lightroom SDK includes the source code for the default HTML web engine, which includes an example implementation of `document.liveUpdate()` in the file `live_update.js`. To include this file in your project, you must construct your data model to match the names used in the JavaScript.

- For any gallery text fields, you must place an `id` attribute on the immediately enclosing element.
- The `id` value must match the dot-separated path to the corresponding model value defined in the `galleryInfo.lrweb` file.

For example, in the default HTML gallery, the *site title* is in an `h1` element. In the template source file it looks like this:

```
<h1 onclick="clickTarget( this, 'siteTitle.text' ); "
       id="metadata.siteTitle.value"
       class="textColor">
    $model.metadata.siteTitle.value
</h1>
```

Notice that the `id` is the same as the path in the `model` definition in the `galleryInfo.lrweb` file:

```
["metadata.siteTitle.value"] = "Site Title",
```

NOTE: This example implementation issues a reload for compound `cssID` values, such as "`#myId.myClass`". Create unique classes for such cases to avoid the reload.

Defining messages from a previewed page to Lightroom

Lightroom provides a callback mechanism whereby JavaScript code running in the previewed page can communicate with Lightroom. The implementation differs slightly in Mac OS and in Windows, but the example `live_update.js` implements a wrapper function which hides this difference. This discussion assumes that you are including `live_update.js` in your web-engine folder.

To call from JavaScript into Lightroom, invoke the `callCallback()` function defined in `live_update.js`, using this syntax:

```
callCallback( "callback_name", param1, param2, ... );
```

For example, to call the in-place-edit callback defined in the sample implementation, the JavaScript makes this call:

```
callCallback( 'inPlaceEdit', target, bounds.x, bounds.y, bounds.width, bounds.height,
    font.fontFamily, font.fontSize, imageID )
```

Lightroom provides these callback functions that can be invoked from JavaScript using `callCallback()`:

`showInPhotoBin = function( id )`	Reveals a photo in the filmstrip whose `id_global` value matches the given `id` value.
`setActiveImageSize = function( size )`	Tells Lightroom which of the sizes is currently being displayed on screen. Use the same string labels that you provided in the `photoSizes` section of the `galleryInfo.lrweb` file.
`inPlaceEdit = function( target, x, y, width, height, fontFamily, fontSize )`	Edits a text field at given coordinates on the screen. See "Specifying in-place edit" on page 146.
`updateModel = function( key, value )`	Alters the data model for the given dot-separated key path.
`fetchURL = function( url, callbackName )`	Downloads the contents of a given URL and returns it as a string. This is an asynchronous operation.When the operation is complete, the result string is passed to the callback. Implement the referenced callback function in the `document` object.

Specifying in-place edit

Your JavaScript code can call `inPlaceEdit()` directly, using `callCallback()`. You must provide these arguments:

`target`	string	The dot-path identifier of a metadata property defined in your model, such as "metadata.siteTitle.value"
`x, y` `width, height`	number	The bounds of the element on the web page, in pixels. These coordinates are used to position the edit text window that is temporarily superimposed on the web page.
`fontFamily`	string	The font family to use for the edit.
`fontSize`	number	The font point size.

The JavaScript file `live_update.js` also provides an easier way to implement in-place edit, by using the `clickTarget()` function. This function gets the bounds and font information for a particular node in the page DOM, and uses it to call the `inPlaceEdit()` function.

You can add in-place editing functionality to any node containing text by adding code like this to your HTML:

```
onclick="clickTarget( this, 'target_property' );"
```

For example:

```
<p onclick="clickTarget( this, 'metadata.groupDescription.value' );"
       id="metadata.groupDescription.value"
       class="textColor">
   $model.metadata.groupDescription.value </p>
```

Notice that the target ID sent to `clickTarget()`, the ID for the node, and the path in the `$model` variable all match.

7 Using ZStrings for Localization

ZStrings are an Adobe convention for defining localization strings. You identify a string according to its usage in the user interface, and specify it in the ZString format. This enables Lightroom to look up language-specific versions of the string to display to the user.

- In Lightroom, you pass ZStrings to the built-in `LOC` function to allow for localization of your plug-in's displayed text. See "The LOC function" on page 150.
- Resolution of ZStrings depends on dictionary files that you supply, which contain the mappings from the ZString path to the localized string. See "Localization dictionary files" on page 151.

  NOTE: Reloading a plug-in interactively or automatically after export does not reload any localization dictionaries supplied with that plug-in. The translation dictionaries are read only when the plug-in is first loaded or Lightroom is restarted.

ZString format

The format of a ZString is:

```
$$$/ZString_path/stringKey=defaultValue
```

`$$$`	The ZString marker is always required to identify a ZString and distinguish it from any other 8-bit ASCII string.
`/ZString_path/stringKey=`	The path and key uniquely identifies a specific string, and is used to look up the translation in a dictionary file that you provide with your plug-in (see "Localization dictionary files" on page 151.) The path is a series of 7-bit ASCII character strings separated by the slash (/) character. You can use any strings you wish, except that no white space is allowed. The last element of the path is a specific key name, which is separated from the default value by an equal sign (=). The path groups a set of properties; for example, you might use a unique path for a particular plug-in, and within that plug-in further group all strings that appear in a particular dialog. Each plug-in has its own mapping of the context paths, so your path names will not conflict with those used by other plug-ins, or by Lightroom itself.
`defaultValue`	The string following the separator (=) is the default display string to use for this ZString. If no matching key exists in the active localization dictionary (or if no appropriate dictionary is found) this value is displayed to the user. Strings values used in ZStrings can contain escape sequences to indicate certain characters; see "ZString characters and escape sequences" on page 149.

Like any Lua string, ZStrings can be enclosed in single or double quotes. For example:

```
LOC "$$$/MyPlugin/Dialogs/Description/sectionName=Description"
LOC '$$$/MyPlugin/Dialogs/Description/Title=Document Title:'
```

ZString characters and escape sequences

ZStrings in code should consist entirely of low-ASCII characters. The key should only contain characters in the set "a-ZA-Z0-9/". The value can contain any low-ASCII character.

- ZStrings allow some common non-low-ASCII characters to be substituted for escape sequences in the strings. For example, the sequence `^T` includes the trademark symbol (™) in that location in the resulting string.
- The general-purpose sequence `^U` encodes any arbitrary Unicode character by its code point, in the form `^U+1234`.
- An escape sequence with a number is a replacement point.; the sequence to be replaced by another string supplied as an additional argument to `LOC()`; see "The LOC function" on page 150.

These substitution sequences are recognized:

Sequence	Replacement
`^r`	carriage return
`^n`	line feed
`^B`	bullet
`^C`	copyright
`^D`	degree
`^I`	increment
`^R`	registered trademark
`^S`	n-ary summation
`^T`	trademark
`^!`	not
`^{`	left single straight quote
`^}`	right single straight quote
`^[`	left double quote
`^]`	right double quote
`^'`	right single curly quote
`^.`	ellipsis ("...")
`^e`	Latin small e with acute accent
`^E`	Latin small e with circumflex

Sequence	Replacement
^d	Greek capital delta
^L	backslash ("\")
^V	vertical bar ("|")
^#	command key (in Mac OS)
^`	accent grave ("`")
^^	circumflex ("^")
^0 - ^9	Marks insertion point for additional LOC argument strings; see "The LOC function" on page 150.
^U+*xxxx*	Unicode code point U+*xxxx*

The LOC function

The global LOC function takes a ZString argument, and automatically performs the table lookup to resolve the display string for the current locale. If it cannot find a matching string in a dictionary for the current locale, or if there is no dictionary for the current locale, it returns the default string provided with the ZString.

You can use the LOC function anywhere you specify display strings:

- In the title for your Export Service Provider
- In the title for menu items that you add
- In the title and value properties of UI elements

Any of these properties can take a simple string or a LOC and ZString value. You are not required to use the LOC function if you do not need to localize the text of your plug-in.

Here is an example of localizing the text that identifies an Export Service Provider in the Export destination section of the Export dialog:

```
LrExportServiceProvider = {
    title = LOC "$$$/MyPlugin/Name=My Plug-in",
    file = 'MyPluginExportServiceProvider.lua',
    ...
    },
```

The LOC function also allows you to combine strings using placeholders in the ZString's value string, and additional string arguments to the function. The placeholders use "hat" notation with a numeric value; the first is ^1, the second ^2, and so on. You can specify up to 9 additional string arguments, which are inserted at the placeholder locations ^1 through ^9 in the localized text.

For example:

```
LOC( "$$$/Message=Could not open the file ^1 because ^2.",
     "myfile.jpg", "a disk error occurred" )
```

The placeholders are replaced by the string arguments, resulting in this string:

```
"Could not open the file myfile.jpg because a disk error occurred."
```

Localization dictionary files

To localize your plug-in's user interface, you must provide a localization dictionary file for each language, named and located as follows:

- Each translation dictionary must be in a file named `TranslatedStrings_code.txt`. The *code* is the two-letter ISO code for the language, such as `en` for English, `fr` for French, `de` for German, `ja` for Japanese, and so on.
- The dictionary files must located be in the top plug-in folder, with the `Info.lua` file.

Lightroom automatically selects the appropriate translation file based on the current language in use for the application.

Lightroom performs ZString translation when it creates the object containing the string. When the `LOC` function encounters a ZString, it looks for the localization dictionary appropriate to the current locale, and uses it to find translations for static ZString values.

- If there are no localization dictionary files, or if none is found to match the application language, the `LOC` function returns the value string found in the original ZString.
- When it does find a matching dictionary file, the `LOC` function locates the ZString in the dictionary file using the *context path* and *property name*; that is, the first part of the supplied ZString up to, but not including the `=` sign. If a matching line is found, `LOC` removes the first part of the found ZString up to and including the `=` sign, and returns the remaining string as the translation.
- If a matching line is not found in the dictionary, the function returns the value string found in the original ZString.

Localization dictionary file format

A localization dictionary file is a text file containing one ZString translation entry per line. Each ZString's final string value is in the destination language. The text must be encoded in UTF-8.

The only things allowed on a line (after the first character) are ZStrings; no newline characters or comments are allowed. The ZStrings in this file must all be enclosed with double quotes.

The following text editors are recommended for creating these files:

- In Mac OS X simply use TextEdit and make sure you save the file type as "UTF-8" (rather than UTF-16 or UCS-2, for instance).
- In Windows use Notepad, and be sure to save the file as type "UTF-8."

The file is UTF-8 formatted text. A leading UTF-8 byte-order marker (`EF BB BF`) is permitted.

Example dictionary file

Here is a small example of a German translation dictionary:

```
"$$$/MyPlugin/Size/Small=Klein"
"$$$/MyPlugin/Size/Medium=Mittel"
"$$$/MyPlugin/Size/Large=Groß"
"$$$/MyPlugin/Size/Large/Extra=Sehr groß"
"$$$/MyPlugin/Image/Title=Titel"
"$$$/MyPlugin/Image/Quality=Qualität"
"$$$/MyPlugin/Image/View=Ansicht"
"$$$/MyPlugin/Enabled=Aktiviert"
```

Supported languages

These languages are supported:

Language	Language code
German	`de`
English	`en`
Spanish	`es`
French	`fr`
Italian	`it`
Japanese	`ja`
Korean	`ko`
Dutch	`nl`
Portuguese	`pt`
Swedish	`sv`
Chinese, simplified	`zh_cn`
Chinese, traditional	`zh_tw`

8 SDK Sample Plug-ins

The Lightroom SDK includes some complete sample plug-ins that you can examine and use to familiarize yourself with the plug-in architecture, and with API and Lua usage in the Lightroom SDK.

The plug-in samples are packaged with the Lightroom SDK, in the folder *LR_SDK*/Sample Plugins/ (see "The Lightroom SDK" on page 8).

The plug-in script files are written using the Lua scripting language which have the file extension .lua. Each section in this chapter lists the program files and support files that are provided in the plug-in folder for each sample.

- "The FTP Upload sample plug-in" on page 154 demonstrates how to use the SDK API to connect to an FTP server and upload images using FTP.
- "The Flickr plug-in" on page 158 demonstrates how to use the SDK API to upload images directly to a Flickr account using HTTP.

  Each of these samples is an Export Service Provider, extending Lightroom's Export dialog by adding a new export destination. The plug-ins define their own export settings, as needed for their operations, and add one or more sections to the Export dialog that allow the user to make settings choices for the export operation.

  In addition, the samples demonstrate how to define and use independent dialogs for confirmations and actions.
- "Metadata and filtering samples" on page 164 demonstrate additional types of standard plug-in functionality. These show how to create Lightroom-specific metadata and use it together with other features, such as customizing the Plug-in Manager, creating dialog boxes, and creating an Export Filter Provider that accesses custom metadata.
- "Post-processing samples" on page 167 demonstrate more types of post-processing that can be accomplished with an Export Filter Provider.
- "Web engine sample" on page 169 demonstrates a different type of plug-in, a web engine, by creating a simple HTML gallery.

The FTP Upload sample plug-in

The FTP sample plug-in demonstrates how to customize the Export dialog with an Export Service Provider that exports images to a remote export destination. The plug-in allows you to upload images directly to an FTP Server.

Plug-in files

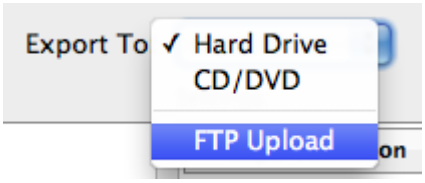
The FTP plug-in folder is *LR_SDK*/Sample Plugins/ftp_upload.lrdevplugin

Info.lua	Information file that describes the plug-in to Lightroom.
FtpUploadExportServiceProvider.lua	The service definition file.
FtpUploadExportDialogSections.lua	Defines the initialization routes and customizations for the Export dialog.
FtpUploadTask.lua	Uploads the images to the FTP Server.

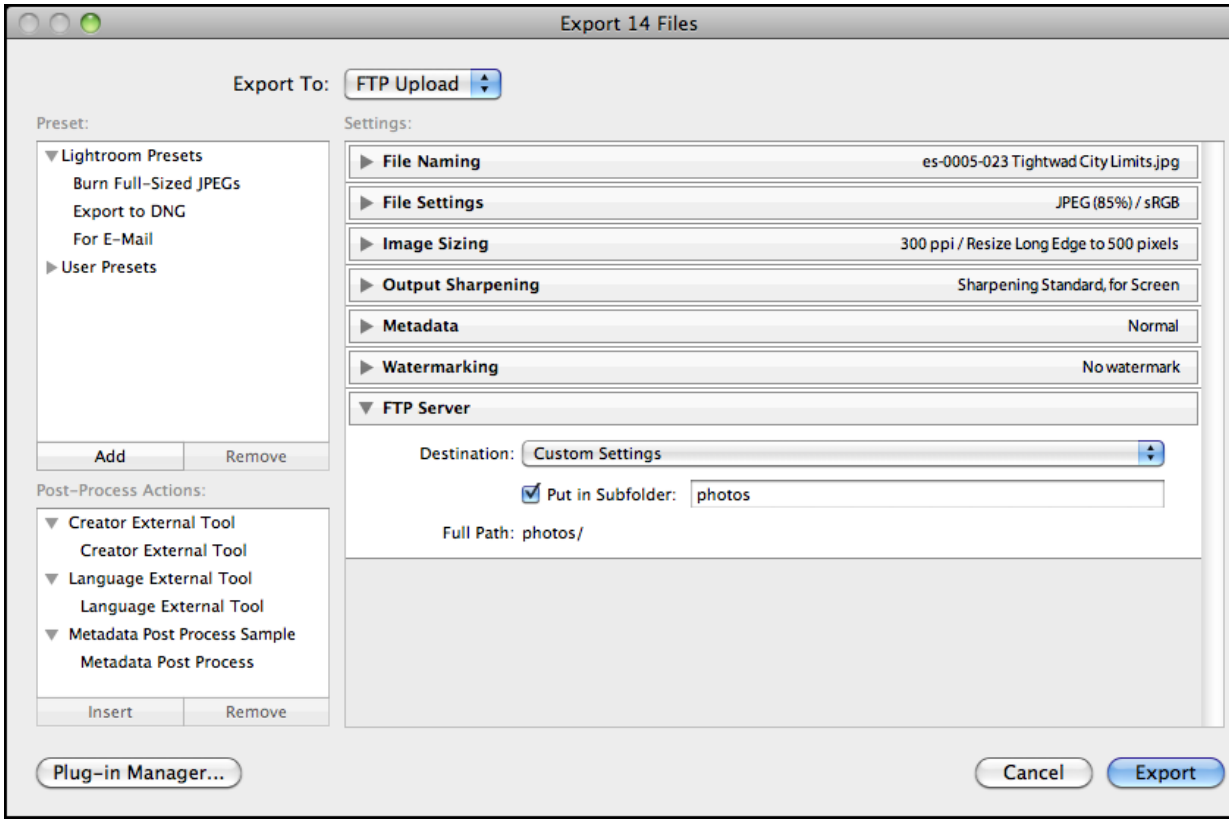
The following steps show how to use the FTP plug-in and guide you through exporting images to an FTP server.

Bring up the FTP plug-in

1. Use the Plug-in Manager to add the plug-in, found in the Lightroom SDK samples folder: *LR_SDK*/Sample Plugins/ftp_upload.lrdevplugin.
2. In the Lightroom Library module, make sure you have at least one image available for export, then choose **File > Export** to bring up the Export dialog.
3. Use the Export destination list at the top of the Export dialog to select the FTP Upload plug-in:

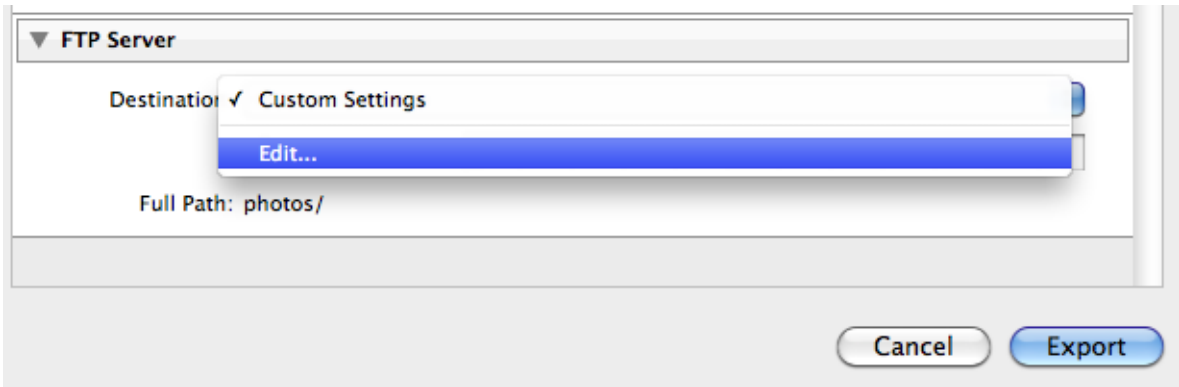


This loads the FTP plug-in and displays the additional FTP Server section it defines for the Export dialog.

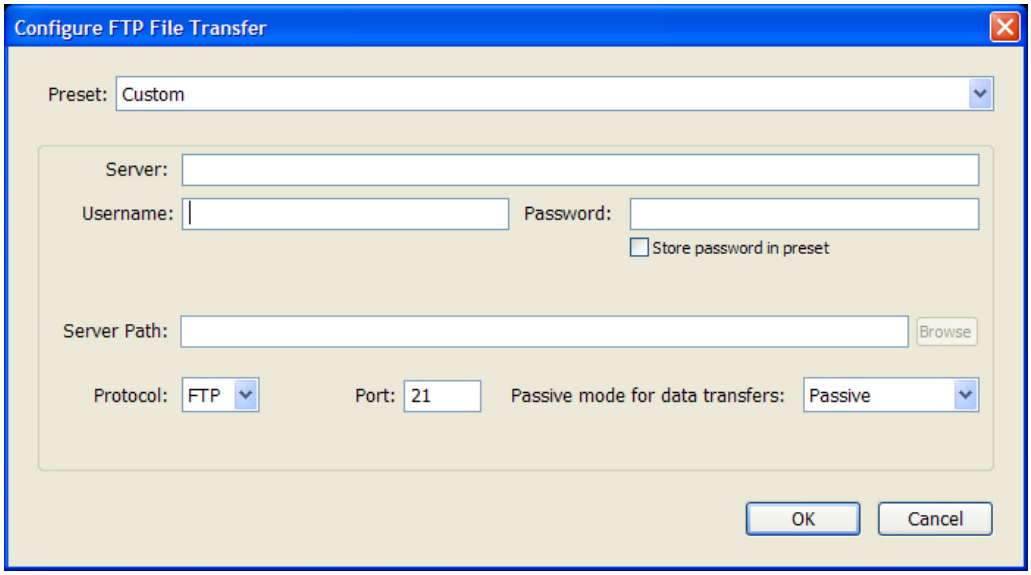


Configure the connection

1. In the FTP Server section of the Export dialog, bring up the Destination pop-up menu and choose Edit.

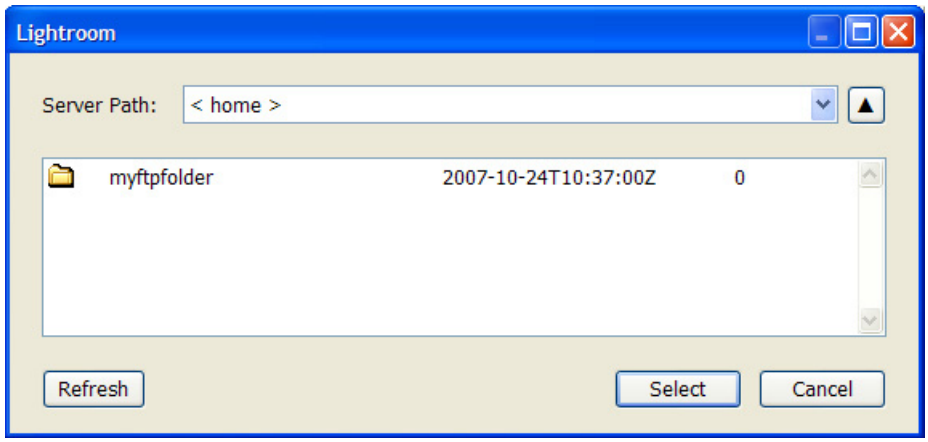


This displays the configuration dialog for the FTP server settings.



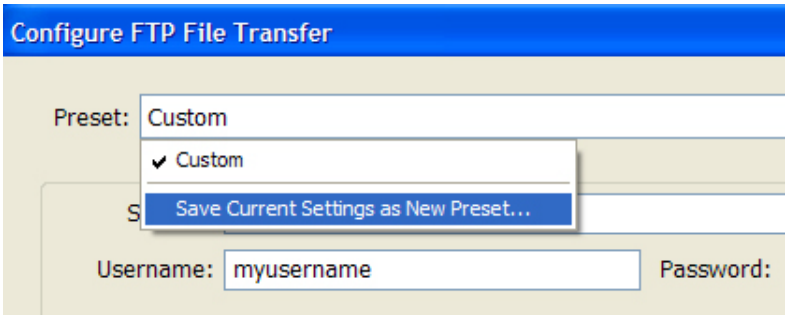
2. Fill out the configuration section:

   - **Server**: Enter the name of the FTP server you wish to connect to, for example: `myftpserver.adobe.com`. You do not need to enter the protocol.
   - **Username**: Enter the username you use to log into your FTP Server.
   - **Password**: Enter the password you use to log into your FTP Server. If you wish, check the 'Store password in preset' checkbox.
   - **Protocol**: Select the protocol from the drop down menu. The default is FTP.
   - **Port**: If your FTP servers uses a port other than port 21, enter the number.
   - **Server Path**: If you need to add the path to your home folder on the FTP server, you can enter the path, or you can click **Browse** to browse the remote file system.



   Navigate to your desired folder and click **Select**. This returns you to the FTP Configure File Transfer dialog.

3. To store this configuration in a preset, bring up the Preset popup and select **Save Current Settings as new Preset**.



In the resulting dialog, enter a name for your preset and click **Create**. Lightroom connects to your FTP server and displays a Browse dialog.

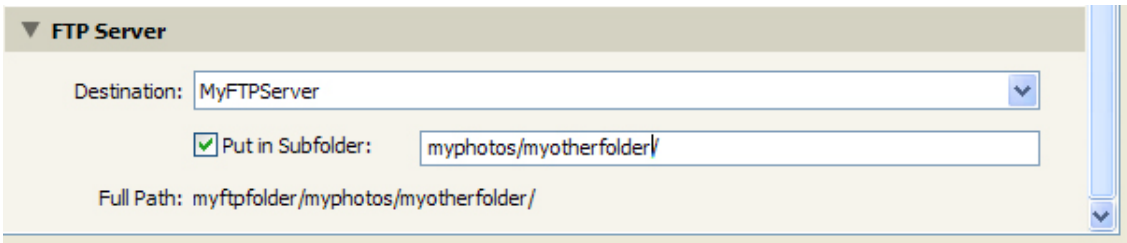
4. Click **OK**. This returns you to the Export dialog.

5. If you want to upload your images to a subfolder within your home folder, select the **Put in Subfolder** checkbox in the FTP Server section of the Export dialog.

   This enables the text field, where you can enter the folder name. The default subfolder name is 'photos'. You can enter another single folder name, or create a subfolder hierarchy be entering the path; for example `myphotos/myotherfolder/`.

   If the folder does not exist on the FTP server, it is created.

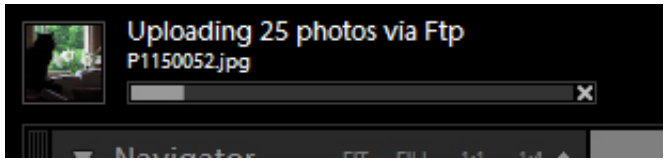
   The bottom the FTP Server section of the Export dialog displays the full path to which your images will be uploaded at.



Establish the connection

1. Click **Export** at the bottom of the Export dialog to begin the export operation.

2. A progress indicator appears in the upper-left corner of the Lightroom catalog window, which allows you to monitor the progress of the Export operation.



The Flickr plug-in

This is more than a sample; it is most of the source code to the plug-in that Lightroom uses to implement its built-in Flickr support. The source code demonstrates how to customize the Publish experience with a Publish Service Provider that uploads photos to the Flickr web service. It also provides examples of other aspects of Lightroom's extensibility and the SDK in general, such as:

- Creating plug-in-defined preset fields and user presets
- Binding settings values to UI components
- Localizing strings

Plug-in files

The Flickr plug-in folder is *LR_SDK*/Sample Plugins/flickr.lrdevplug-in

Info.lua	Information file that describes the plug-in to Lightroom.
FlickrExportServiceProvider.lua	The service definition file.
FlickrExportDialogSections.lua	Defines the initialization routes and customizing for the Export dialog.
FlickrPublishSupport.lua	Defines the publishing operations for publication to Flickr.
FlickrAPI.lua	Handles Flickr requests and responses.
FlickrUser.lua	Manages the Flickr user account and authentication.

The Flickr API

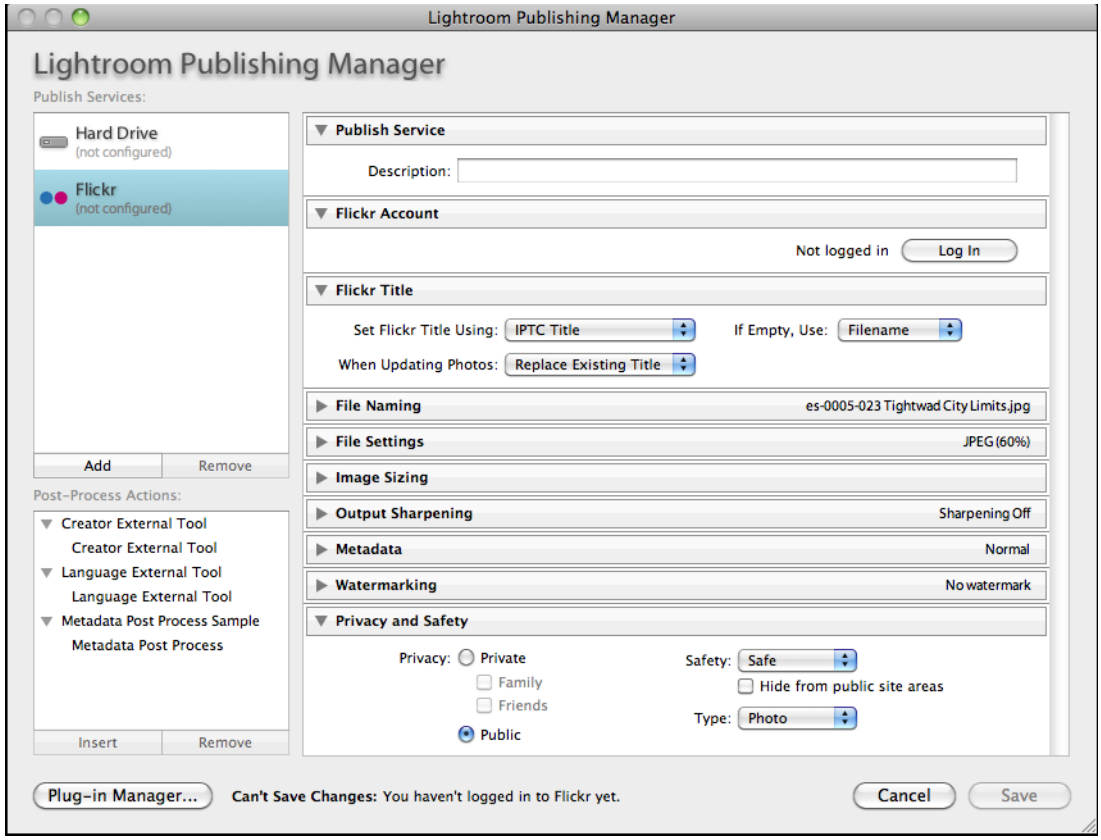
The Flickr plug-in makes use of the services API which Flickr provides. The Flickr API offers many callable methods, several of which are used in this sample. For information about the Flickr API, see http://www.flickr.com/services/api; in particular, these sections discuss services that are used here:

Flickr API Home	http://www.flickr.com/services/api
Authentication	http://www.flickr.com/services/api/auth.spec.html
Frob	http://www.flickr.com/services/api/flickr.auth.getFrob.html
Auth Tokens	http://www.flickr.com/services/api/flickr.auth.getToken.html

Flickr plug-in walkthrough

These steps guide you through authorizing Lightroom with a Flickr account and publishing images.

1. In the Lightroom Library module, make sure the Publish Services panel is open, and shows the Flickr service.
   - If the Flickr service does not appear, go to the Plug-in Manager and enable the Flickr plug-in.
2. In the Publish Services panel, click **Set Up** next to the Flickr entry. This shows the Flickr plug-in's extensions to the Publishing Manager:

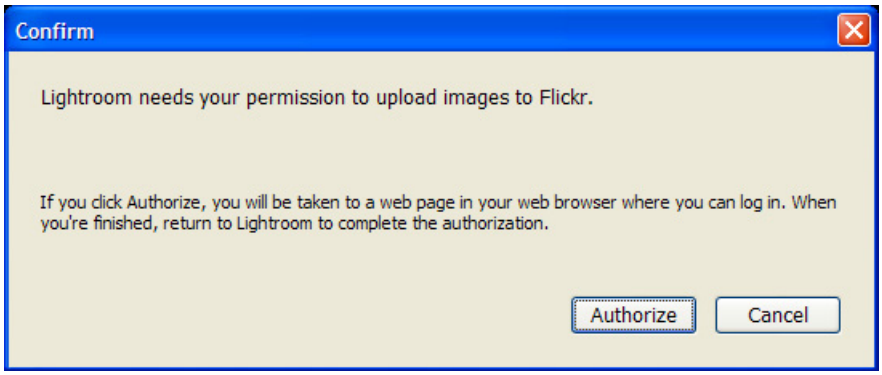


3. If it is not already selected, select the Flickr publish services at the left.

   The plug-in's Publish Service Provider defines text for the top section that shows when the service is selected, and adds new sections to the dialog. Notice that is also adds a status message at the bottom, next to the Plug-in Manager button.

4. In the Flickr Account section of the Publishing Manager, click **Log In**.

   This invokes a dialog that informs you that you must authorize Lightroom in order for the plug-in to correctly upload images. This dialog is defined by the plug-in code in `FlickrAPI.lua`.



   Click **Authorize**.

5. The plug-in brings up the default browser, which displays a web page where you must log into your Flickr account. Log in with the correct Flickr username and password for the account.

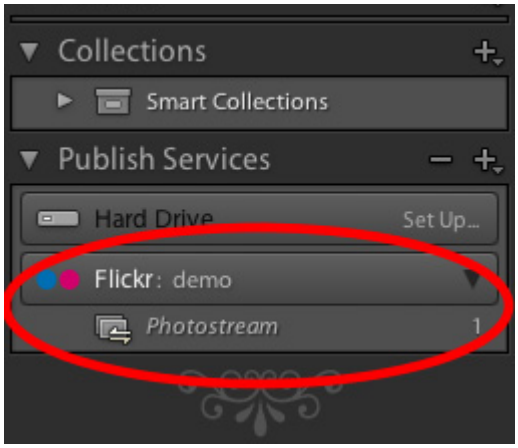
When you have finished logging into Flickr and agreed to allow Lightroom access, it shows a web page stating that the application is authorized.

6. Return to Lightroom and click **Done**. This returns you to the Publishing Manager dialog.

    - Once you have logged in, notice that the **Log In** button in the Flickr Account section has changed to **Switch User**.

7. Click **Save** to establish the new publish service.

    - Notice that the new publish service now appears under the Flickr service in the Publish Services panel.

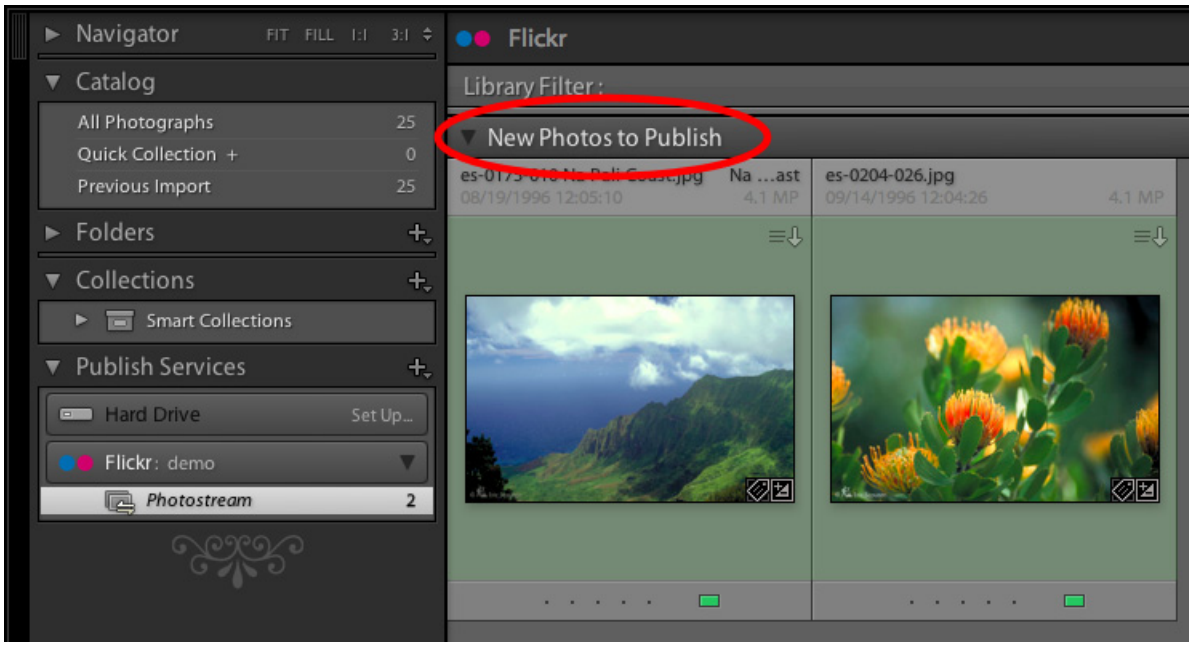


Add photos and publish the collection

Lightroom has created a default collection named "Photostream" in your new service. So far, there are no photos in it. The next thing to do is add some photos to the publish collection, so you can publish them with your new service.

1. Select some photos from your catalog and drag them to Photostream in the Publish Services panel.

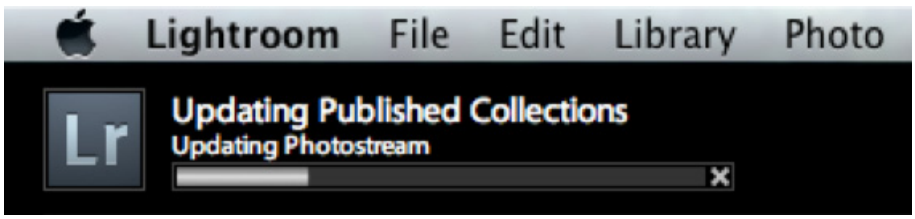
2. Click on the Photostream collection to display those photos in a section called "New Photos to Publish."



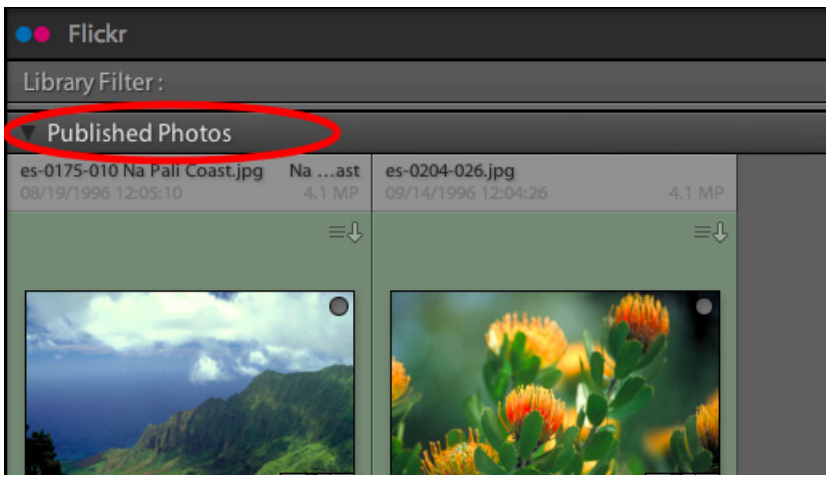
3. Notice that these photos are not published until you explicitly ask Lightroom to publish them.

   To do this, click **Publish** at the bottom left corner of the screen.

4. As it does during an export operation, Lightroom renders the photos and uploads them to the service. A progress bar in the top left corner of the screen shows the status of the publish operation.



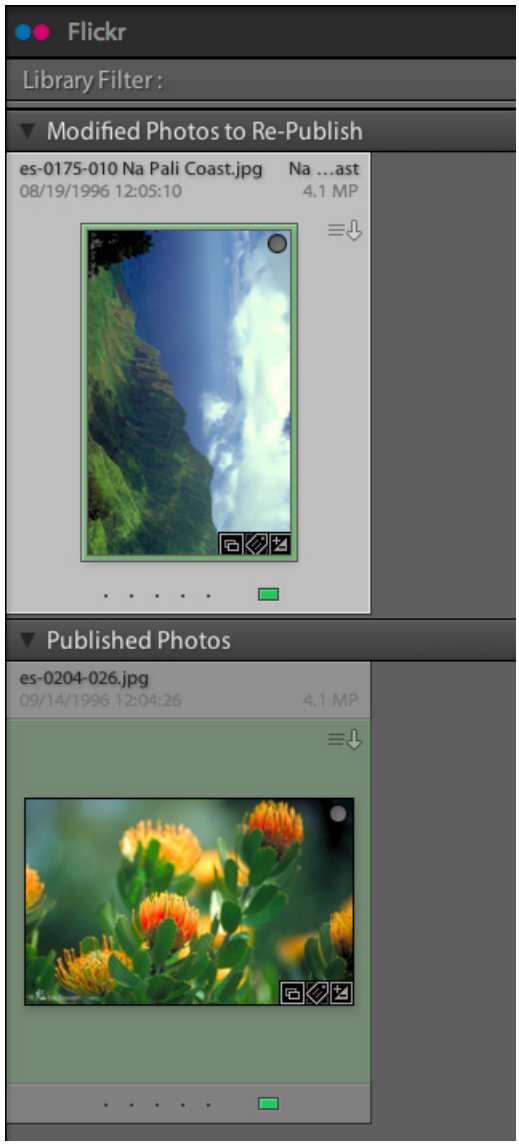
5. Once the process is complete, Lightroom show these photos in a new group called "Published Photos."



Publish modified photos

A publish service differs from an export service in that it maintains an ongoing relationship between Lightroom and the service (in this case, Flickr). If you make changes to photos that have already been published or delete them, Lightroom attempts to mirror those changes on the service. To see this:

1. Select one of your photos that has been published and make a simple change to it (for example, rotate the photo 90°). You will see that photo move to the "Modified" state.



2. Click **Publish** again.

   Lightroom re-exports the photo, replacing the existing photo on the server with the new version.

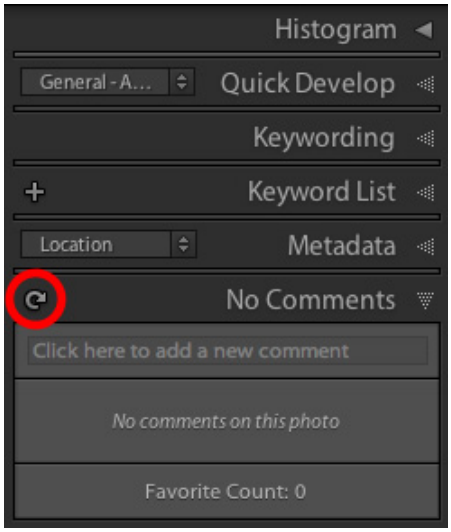
NOTE: Because of limitation in the Flickr API, Lightroom cannot replace photos on a free account. If you attempt to do so, Lightroom ask s if you want to delete the photo and republish it.

Get feedback from viewers

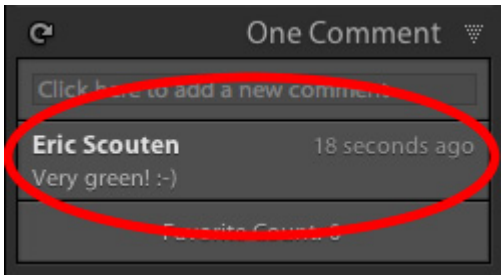
Lightroom's Publish feature allows you to retrieve selected feedback from a web service and display it in Lightroom's Comments panel. To see this:

1. Select one of your photos that has been published.
2. Right click or CTRL-click on the selected photo and choose **Show in Flickr.**

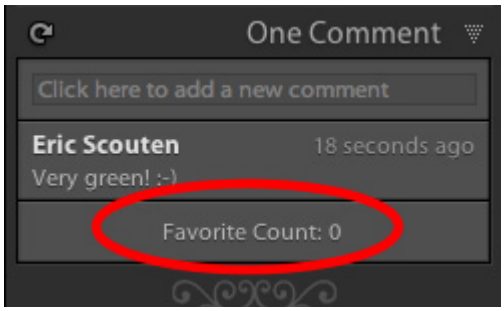
   This takes you to the Flickr web site where you can see the photo.
3. Using the Flickr site, enter a comment on your photo.
4. Return to Lightroom and click **Refresh** in the Comments panel.



After a brief delay, the comment you entered on Flickr appears in the Comment panel.



Similarly, Lightroom can gather and display *ratings* given by web site viewers; what constitutes a rating depends on the web service. In the case of Flickr, it is the number of times somebody other than you has marked your photo as a favorite.



Metadata and filtering samples

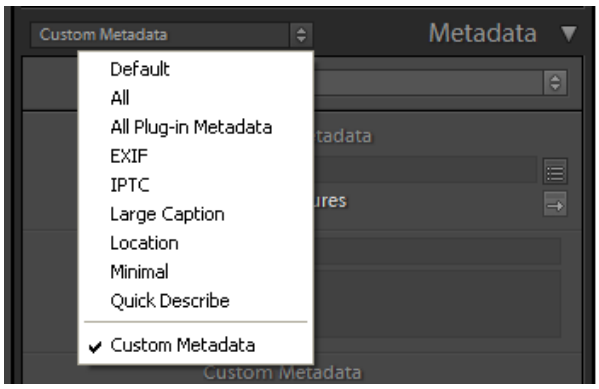
These related samples include the following:

`custommetadatasample.lrdevplugin`
 	
Files:
 	
`Info.lua`
`CustomMetadataDefinition.lua`
`CustomMetadataTagset.lua`
`AllMetadataTagset.lua`
`DisplayMetadata.lua`
`PluginInfoProvider.lua`
`PluginInit.lua`
`strings/en/TranslatedString.txt`
	Creates custom metadata fields and tagsets for use within Lightroom. Also demonstrates:
	
	• Customization of the Plug-in Manager dialog and plug-in load behavior.
	
	• Localization of display strings using a string dictionary.
	
	• Creation of a dialog that displays the values of these custom metadata fields for selected photos.

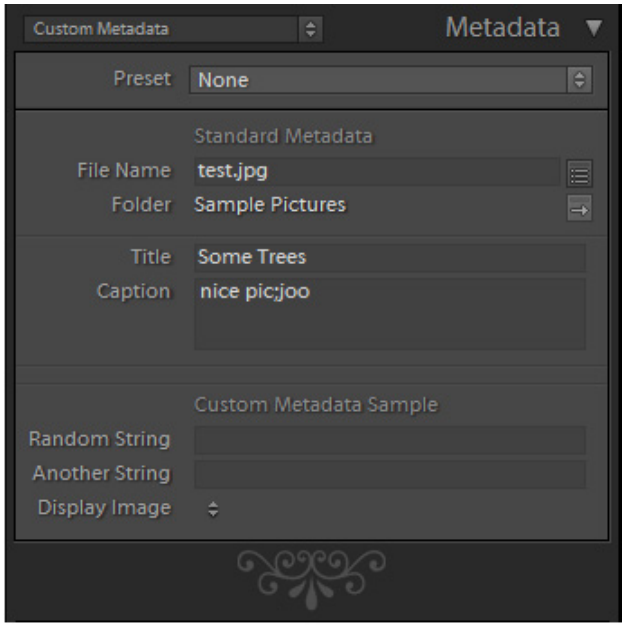
`metaexportfilter.lrdevplugin`
 	
Files:
 	
`Info.lua`
`Metadata.lua`
`MetadataExportFilterProvider`
	Demonstrates a simple Export Filter Provider by defining a post-process action and a related section in the Export dialog. This action offers the user a choice of metadata values to filter on, and removes all photos that do not match that choice from the export operation. It makes use of `shouldRenderPhoto()`, and of the metadata fields defined in the previous sample.

Custom metadata sample walkthrough

1. Use the Plug-in Manager to add the plug-in, found in the Lightroom SDK samples folder: *`LR_SDK`*`/Sample Plugins/custommetadatasample.lrdevplugin`
2. In the Metadata panel of the Library module, open the menu at the top left and choose Custom Metadata.

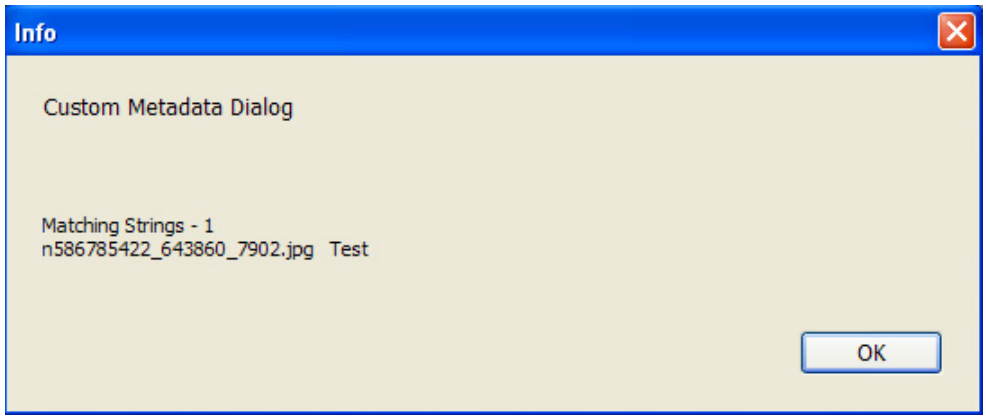


3. The custom metadata created by the plug-in appears in the metadata panel.



4. Select a photo.
5. Change the Display Image value to Yes, and the Random String value to Test.

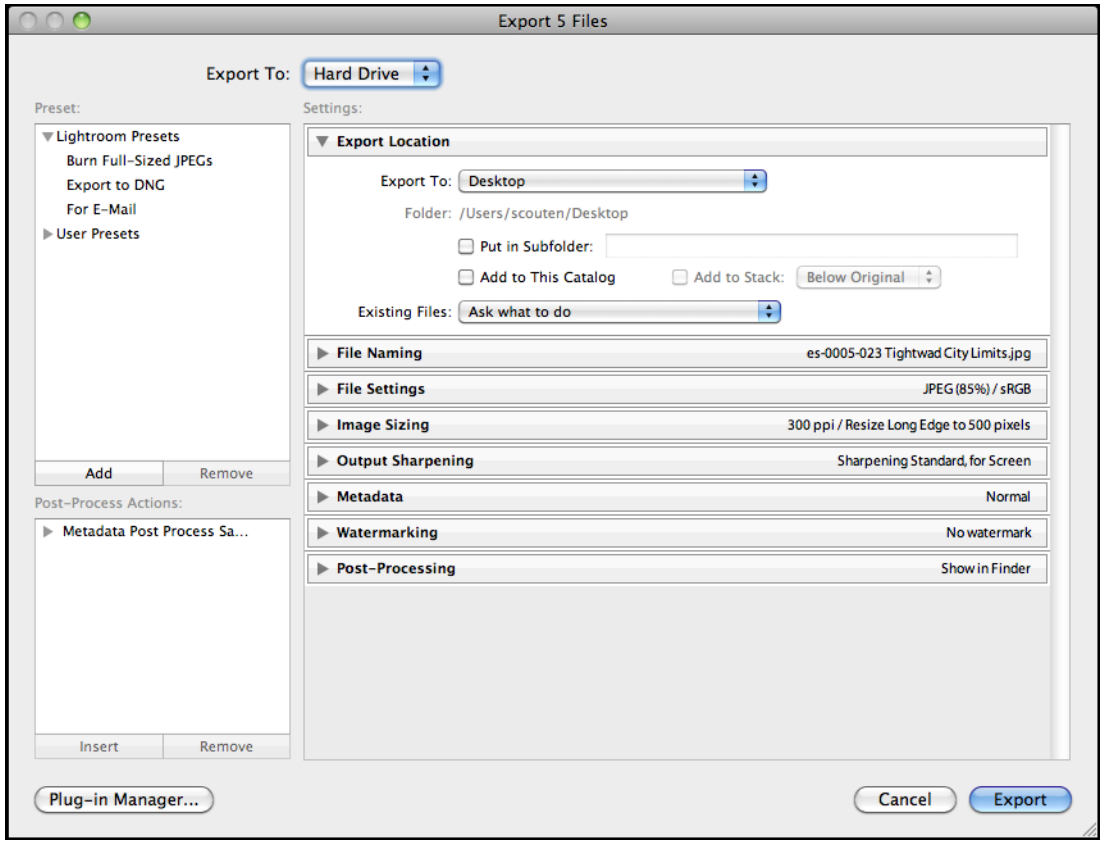
   These values are used by the following sample.
6. Select several photos in the Library module, including the one that you modified in the step 5 above.
7. From the Library menu, choose Plug-in Extras > Custom Metadata Dialog (an item added by this plug-in).
8. A dialog defined by this plug-in appears, showing the name of the photo for which you set the Display Image value to Yes:



Metadata filter sample

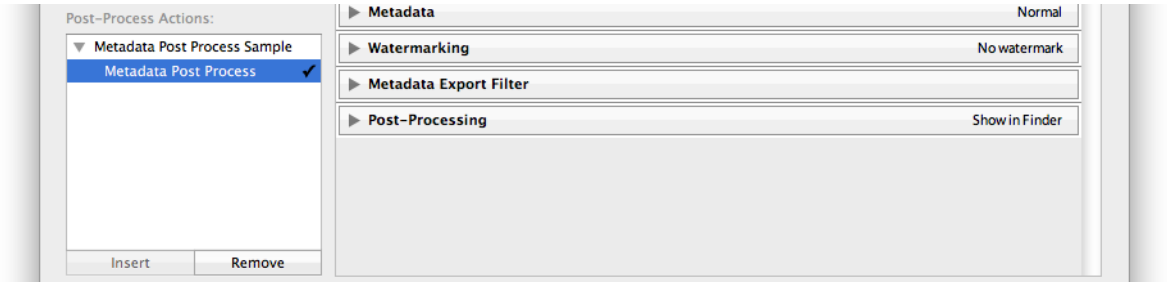
1. Use the Plug-in Manager to add the plug-in, found in the Lightroom SDK samples folder: *LR_SDK*/Sample Plugins/metaexportfilter.lrdevplugin.
2. Select some photos in the Library module, noting the Title metadata values for one or more.

3. Open the Export dialog by choosing File > Export.



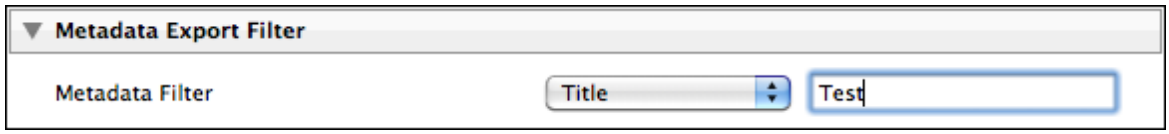
4. Notice the entry for this plug-in, **Metadata Post Process**, in the Post-Process Actions panel.

5. Open the entry by clicking the gray arrow at the left, select the action, click Insert.



6. Notice the check mark by the action, which indicates that it has been inserted into the export operation, and the related dialog section at the right, which has been defined by this plug-in.

7. In the new Metadata Export Filter dialog section, choose Title from the drop-down menu, and enter the title of one of your selected photos in the edit text field.



8. Click Export to start the export operation.

   Only the single picture whose Title value matches the one you entered is exported; all the other photos in your selection are removed from the export operation by this Export Filter Provider.

Post-processing samples

The samples `creatorfilter.lrdevplugin` and `languagefilter.lrdevplugin` provide additional examples of post processing, using Export Filter Providers. These plug-ins show the typical construction of Export Filter Providers, making use of an external application to process XMP metadata. Together, the samples demonstrate how you can combine multiple post-processing actions, allowing the user to choose one, both, or neither of the actions.

The Creator External Tool (defined in `creatorfilter.lrdevplugin`) also includes the metadata filter logic defined in the "Metadata filter sample" on page 165, which excludes files with matching metadata from the export operation. This illustrates how to combine a simple exclusion filter with the external post-processing that writes XMP metadata.

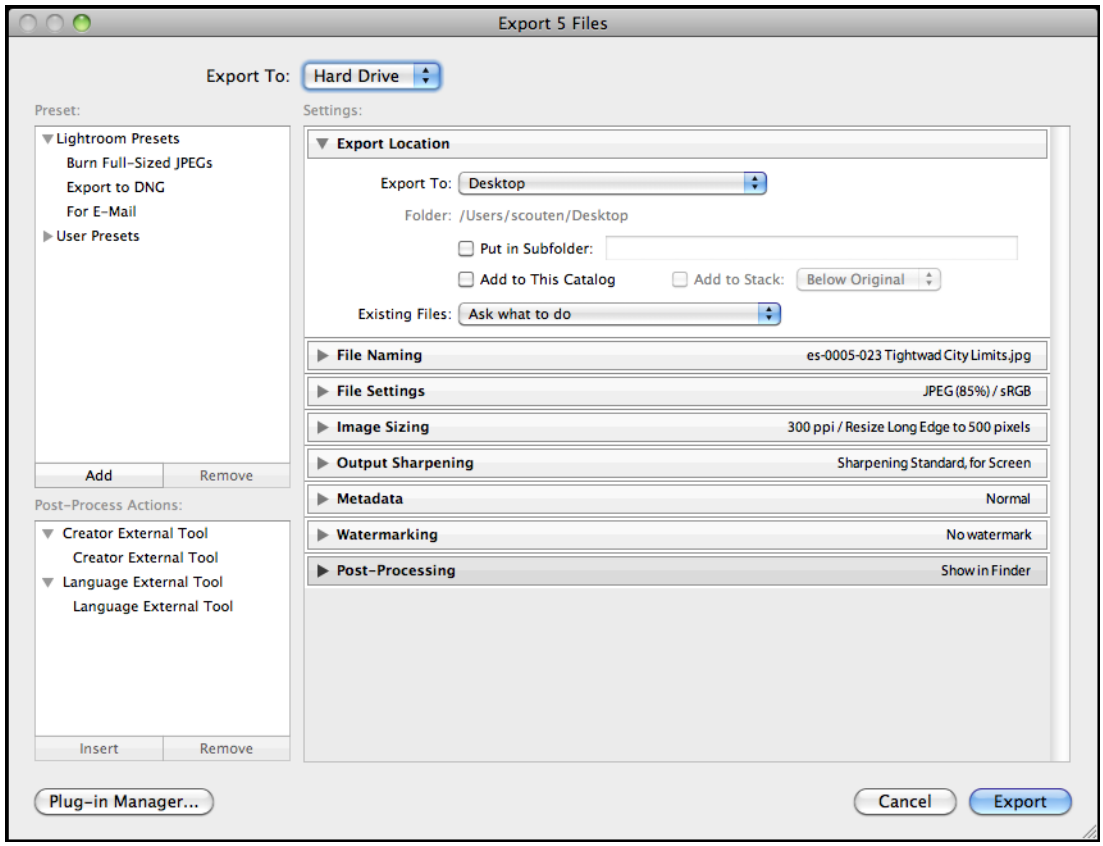
Plug-in files

`creatorfilter.lrdevplugin`	Allows the user to add or modify certain XMP metadata values to photos being exported.
`Info.lua` `CreatorExternalToolFilterProvider.lua`	The Export Filter Provider information and definition script.
`win\LightroomCreatorXMP.exe` `mac/LightroomCreatorXMP`	The platform-specific external XMP application that performs the selected action.
`languagefilter.lrdevplugin`	Allows the user to update one of the localized values for the Title property in the XMP metadata.
`Info.lua` `LanguageExternalToolFilterProvider.lua`	The Export Filter Provider information and definition script.
`win\LightroomLanguageXMP.exe` `mac/LightroomLanguageXMP`	The platform-specific external XMP application that performs the selected action.

Post-processing actions walkthrough

1. Use the Plug-in Manager to add the plug-ins, found in the Lightroom SDK samples folders: *LR_SDK*`/Sample Plugins/creatorfilter.lrdevplugin` and *LR_SDK*`/Sample Plugins/languagefilter.lrdevplugin`.

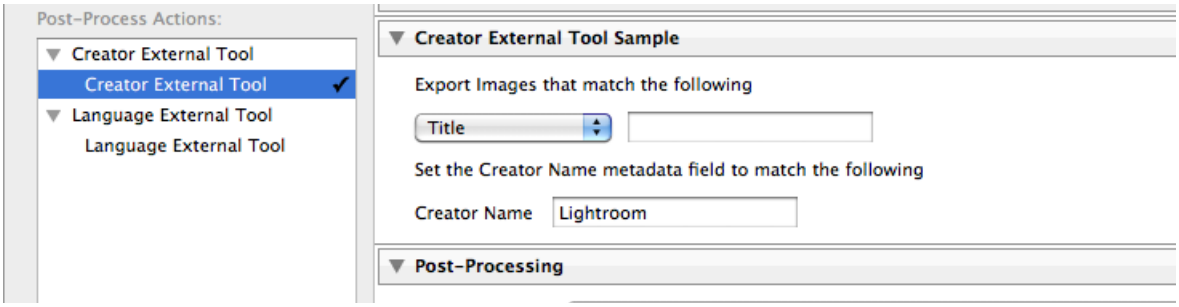
2. Select some photos in the Library module.
3. Open the Export dialog by choosing File > Export.



Notice the entry for these plug-ins, **Creator External Tool** and **Language External Tool**, in the Post-Process Actions panel. Each plug-in defines one action.

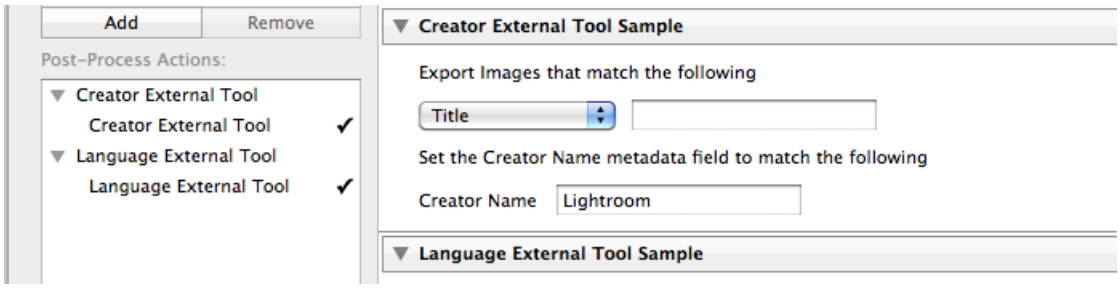
4. Open the **Creator External Tool** entry by clicking the gray arrow at the left, select the action, and click Insert (or simply double-click the action).

   The section defined for that action appears, allowing the user to enter a value for the creator-name XMP property.



Notice that this action also includes the metadata filter—that is, it allows the user to exclude photos whose plug-in-defined metadata values match the users choice. The logic for this part is the same as that defined in the "Metadata filter sample" on page 165.

5. Select and insert the action under Language External Tool; the section for that action is added, allowing the user to select a language and new value for that language's translation value of the XMP Title property.



6. Remove one or both of the actions from the processing queue, and observe the changes in the dialog. You can remove an action by double-clicking the name in the Post-Process Actions section, by selecting the action and clicking Remove, or by using the X icon in the upper right corner of the corresponding dialog section.

7. Try changing the order of the actions using the up and down arrows in the upper right corner of the corresponding dialog section.

8. With one or both of the actions inserted in the queue, make choices in the dialog sections, and click Export to begin the export operation.

9. Open the exported photos in any tool that shows XMP metadata and observe the result.

   - Any photos you filtered out based on Lightroom metadata values should not have been exported.
   - The XMP metadata should reflect the value you entered for Creator, and the translation you entered for Title.

Web engine sample

The sample plug-in `websample.lrwebengine` demonstrates the web-engine, showing how the architecture differs from that of a standard plug-in. It illustrates all of the plug-in parts by creating a simple HTML gallery.

Plug-in files

`manifest.lrweb`	The manifest maps LuaPage source files and template files to Web Gallery HTML output files using a set of commands for different kinds of pages and resource files.
`galleryInfo.lrweb`	Defines the data model and UI for the gallery.

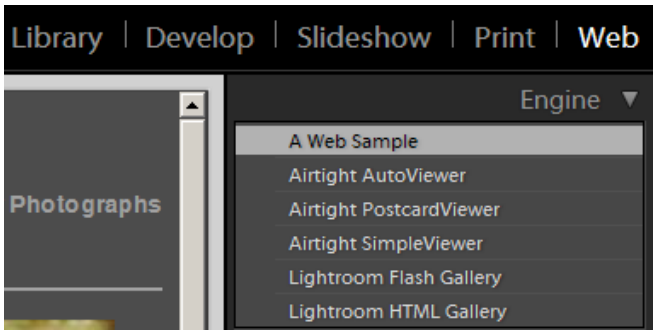
`grid.html`
`header.html`
`footer.html`
	Template LuaPages, HTML with embedded Lua and JavaScript code.

`readme.txt`	An explanation of how the sample works

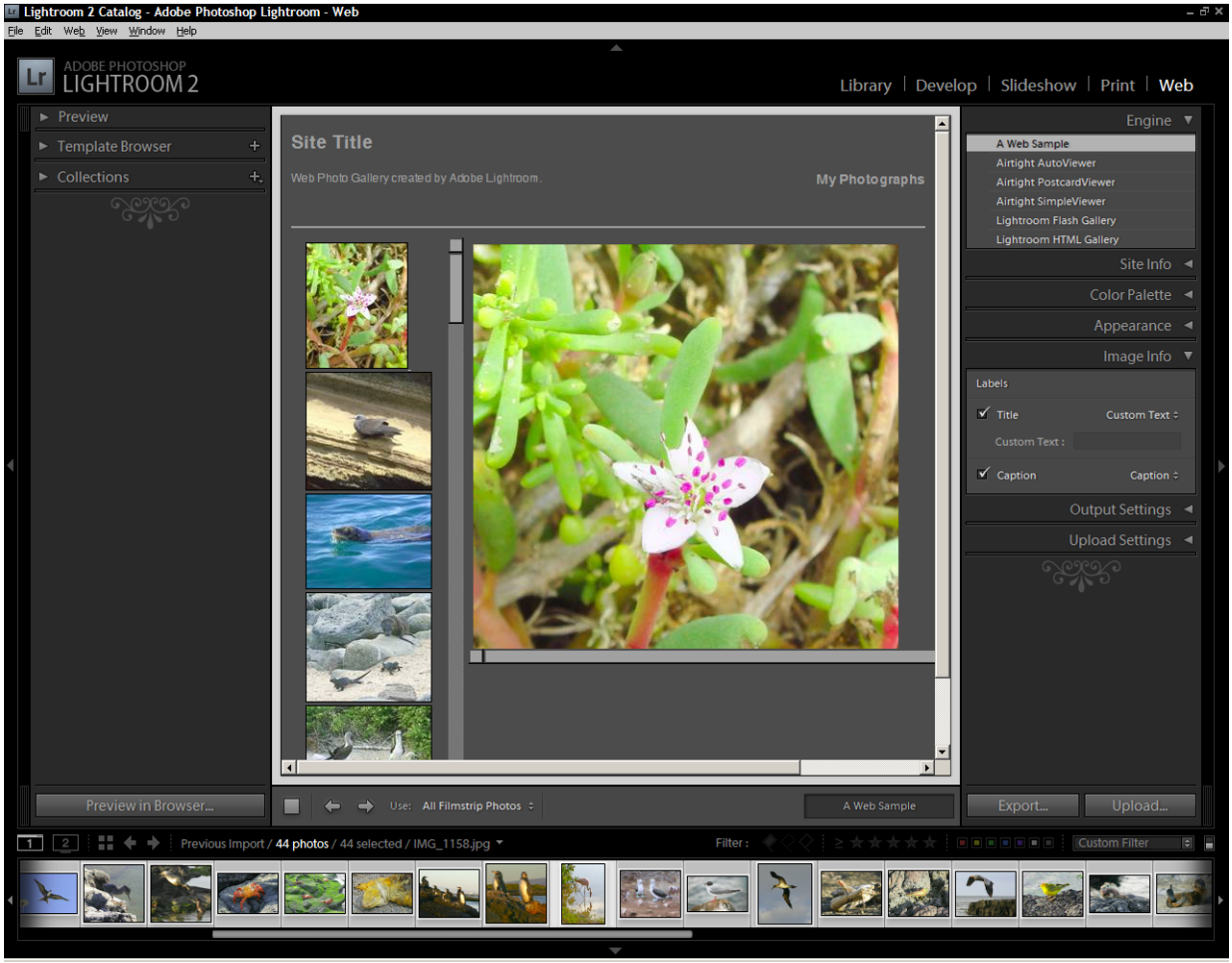
1. Install the plug-in folder, found at *`LR_SDK`*`/Sample Plugins/websample.lrwebengine`, in the Lightroom folder *`LR_Root`*`/shared/webengines`.

2. Start Lightroom, select some photos, and go to the Web module.

3. Select the web engine defined by this plug-in that appears in the Engine list:



4. The gallery preview appears, showing a filmstrip of small images on one side and a larger version of the selected image on the right.



9 Getting Started: A Tutorial Example

This chapter will help you get started with extending Lightroom's Export behavior by walking through the creation of the simple Hello World plug-in. This plug-in adds menu items to the File and Library menus, and defines dialog boxes that are displayed when the menu items are selected. The plug-in also demonstrates how to output and view trace information for debugging and development.

This chapter shows how to build plug-ins that extend the Export functionality of Lightroom. The concepts and techniques are explained in more detail in Chapter 3, "Creating Export and Publish Services."

- Additional features you can add using the same framework are demonstrated in Chapter 10, "Defining Metadata: A Walkthrough."
- Web Gallery plug-ins, which use a different framework, are demonstrated in Chapter 11, "Web Gallery Plug-ins: A Tutorial Example."

Creating an export plug-in

You can place a plug-in folder anywhere, and notify Lightroom of its location using the Plug-in Manager. A plug-in must be packaged for delivery within a single folder, with the suffix `.lrplugin`. For development, you can use the suffix `.lrdevplugin`. Thus, the Hello World plug-in will be placed in the folder `helloworld.lrdevplugin`.

Create the information file

1. Create a text file and save it as `helloworld.lrdevplugin/Info.lua`.

   You must describe your plug-in to Lightroom by creating an `Info.lua` file and placing it in your plug-in folder. This script must return a table that describes the plug-in to Lightroom.

2. Edit the script in the information file to return a table. This table must contain the version number for the SDK and a unique string to identify the plug-in.

   Add the following code to the `Info.lua` file:

   ```
   return {
       LrSdkVersion = 5.0,
       LrToolkitIdentifier = 'com.adobe.lightroom.sdk.helloworld',
   }
   ```

3. Add another entry to the returned table to create a menu item in the Lightroom File menu.

   Place the following code after the `LrToolkitIdentifier` entry:

   ```
   LrExportMenuItems = {
       title = "Hello World Dialog", -- The display text for the menu item
       file = "ExportMenuItem.lua", -- The script that runs when the item is selected
   },
   ```

   (This entry adds only one menu item, so it defines a single table, rather than a table of tables.)

4. Add another entry to the returned table to create a menu item in the Lightroom Library menu.

Place the following code after the `LrExportMenuItems` entry:

```
LrLibraryMenuItems = {
    title = "Hello World Custom Dialog", -- The display text for the menu item
    file = "LibraryMenuItem.lua", -- The script that runs when the item is selected
},
```

5. Save your changes to the file.

This defines a plug-in that adds two menu items:

- The item that we have added to the File menu, **Hello World Dialog**, appears under the Export section of that menu. It displays one of the SDK's predefined dialog boxes.
- The item that we have added to the Library menu, **Hello World Custom Dialog**, displays a customized dialog box.

Create the service scripts

Each menu item, when selected, runs the associated *service script*, which defines that command's behavior. In this case, we will define both of these commands to display dialog boxes.

The Lightroom SDK provides the facility to display both predefined and customized dialogs using the `LrDialogs` namespace. To give your script access to a namespace you must import the namespace with the `import()` function. You can then use the namespace functions to specify and invoke the dialogs.

Now we will walk through creating the service scripts for the two menu items.

1. Create the files `ExportMenuItem.lua` and `LibraryMenuItem.lua`, and save them in the plug-in folder.
2. Edit `ExportMenuItem.lua` as described below in Displaying a dialog.
3. Edit `LibraryMenuItem.lua` as described in "Displaying a custom dialog" on page 173.

Displaying a dialog

This example demonstrates a simple service script that displays one of the predefined dialogs. It shows how to import the `LrDialogs` namespace, and create a function to display the message dialog, with a script-defined message.

Use these steps to create the service script:

1. Edit the `ExportMenuItem.lua` file to import the `LrDialogs` namespace:

```
local LrDialogs = import 'LrDialogs'
```

2. Create a function named `showModalDialog()` in your own plug-in namespace:

```
MyHWExportItem = {}
function MyHWExportItem.showModalDialog()
    -- body of function
end
```

3. In the body of your function, use the `LrDialogs` namespace function `message()` to present a predefined modal message-display dialog, which displays the simple text 'Hello World.'

Place this line of code in the body of the function:

```
-- body of function
LrDialogs.message( "ExportMenuItem Selected", "Hello World!", "info" )
```

4. To call the function when the script runs, place this line at the end of the script:

```
MyHWExportItem.showModalDialog()
```

5. Save your changes to the file.

We will check the result after we have set up the second menu item.

Displaying a custom dialog

The item that we added to the Library menu creates a custom dialog, which required quite a bit more programming. These steps describe how to write the service script that defines the program data and custom interface elements, ties the data to the UI elements, and displays them in a custom dialog box.

1. Edit the `LibraryMenuItem.lua` file to import the following namespaces and classes:

```
local LrFunctionContext = import 'LrFunctionContext'
local LrBinding = import 'LrBinding'
local LrDialogs = import 'LrDialogs'
local LrView = import 'LrView'
local LrColor = import 'LrColor
```

2. Create a function named `showCustomDialog()` in your own plug-in namespace:

```
MyHWLibraryItem = {}
function MyHWLibraryItem.showCustomDialog()
    -- body of show-dialog function
end
```

Create a properties table for program data

We are going to create a properties table to keep the program data, the key values that we will bind to the UI elements to make them dynamic. This is an observable table, which requires a function context to automatically remove the notifications if anything goes wrong.

3. To get the function context, add the following code inside the `showCustomDialog()` function:

```
-- body of show-dialog function
LrFunctionContext.callWithContext( "showCustomDialog", function( context )
    -- body of called function
end)
```

Notice that the second argument is the main function, which is passed an `LrFunctionContext` object.

4. In the body of the main function, create an observable table using the `LrFunctionContext` object.

Add this to the body of the main function for `callWithContext()`:

```
-- body of called function
local props = LrBinding.makePropertyTable( context ) -- create bound table
```

5. Add a key to the observable table called `isChecked`:

```
-- body of called function
local props = LrBinding.makePropertyTable( context ) -- create bound table
props.isChecked = false -- add a property key and initial value
-- create view hierarchy
```

Create UI elements

The Lightroom SDK also provides the `LrView` class and namespace which allows you to create custom dialog elements. You need to populate the custom dialog with a view hierarchy that defines the custom-UI portion of the dialog.

We imported the `LrView` namespace with the `import()` function. Now we will use the namespace function `LrView.osFactory()` to obtain a view-factory object, then use that object to create the UI elements.

6. Add code to obtain a view-factory object:

```
-- create view hierarchy
local f = LrView.osFactory()
```

7. The variable `c` will hold the view hierarchy that defines the dialog contents. The root node is a row container, and it is bound to the observable data table that we created in step 4 above. All of the child nodes inherit this binding, so that they can easily reflect and set data values in this table.

   Add this code:

```
local f = LrView.osFactory()

local c = f:row { -- the root node
   bind_to_object = props, -- bound to our data table
   -- add controls
```

8. Add a checkbox control as a child of the row, and bind it to the `isChecked` property we created in step 5:

```
   -- add controls
   f:checkbox {
      title = "Enable", -- label text
      value = LrView.bind( "isChecked" ) -- bind button state to data key
   },
```

9. Create an editable text field, setting the value to some arbitrary text. This field will only be enabled when the checkbox is checked:

```
   f:edit_field {
      value = “Some Text”,
      enabled = LrView.bind( "isChecked" ) -- bind state to same key
   },
}
```

10. Use `LrDialogs.presentModalDialog()` to display the custom dialog. The argument is a table with entries for the dialog `title` and the view hierarchy that defines the `contents`:

```
local result = LrDialogs.presentModalDialog(
    {
        title = "Custom Dialog",
        contents = c, -- the view hierarchy we defined
    }
)
```

11. To call the function when the script runs, add this at the bottom of the script:

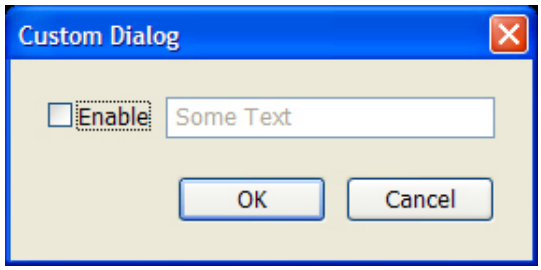
```
MyHWLibraryItem.showCustomDialog()
```

12. Save your changes to the file.

Run the plug-in

Use these steps to run the plug-in and see how the menu items bring up the two dialogs:

1. In Lightroom, choose **File > Plug-in Manager** to show the Plug-in Manager dialog.
    - If you have not yet added this plug-in to Lightroom, click **Add**, navigate to the plug-in folder you created, and click **Add Plug-in**.
    - If you added the plug-in earlier, reload it. Open the Plug-in Author Tools section, select the plug-in, and click **Reload Plug-in**.
2. Choose **File > Plug-in Extras > Hello World Dialog** to show the predefined modal dialog created by the `ExportMenuItem.lua` script.
3. Click **OK** to dismiss the dialog.
4. Choose **Library > Plug-in Extras > Hello World Custom Dialog** to show the custom modal dialog created by the `LibraryMenuItem.lua` script:



5. This example isn't very interesting yet, since no other controls are bound to data values. Click **OK** or **Cancel** to dismiss, the dialog, and we will add some more complex bindings and behavior.

Transforming data

The very simple binding we created for the checkbox allows you to set and clear a data value by selected or deselecting the checkbox button. To show a more complex relationship between the UI and the data, we will add two radio buttons and a static text field. All three are bound to the same data key, but with transformations such that when you select one radio button, it deselects the other, and updates the text to show which is selected.

Create multiple bindings to one key

Use these step to populate a custom dialog with this new set of controls and create the data transformation.

1. Edit the `LibraryMenuItem.lua` file to create a new function, `showCustomDialogWithTranform()`:

```
function MyHWLibraryItem.showCustomDialogWithTransform()
    -- body of function
end
```

2. Within this function, make the function-context call you need for the property table:

```
LrFunctionContext.callWithContext( "showCustomDialogWithTransform",
                                    function( context )
    -- body of function
end )
```

3. In this context, create the observable table, and add a property named `selectedButton`, with an initial value:

```
-- body of function
local props = LrBinding.makePropertyTable( context )
props.selectedButton = "one" -- new property with initial value
-- create view hierarchy
```

4. Now we will create a new view hierarchy for the dialog, whose controls are bound to this table. This is a slightly more complex hierarchy, where the root node is a column container, which has two rows. The rows contain the controls, two radio buttons and a text box:

```
-- create view hierarchy
local f = LrView.osFactory() -- get the view factory object

local c = f:column {
    bind_to_object = props, -- all controls bound to our table
    spacing = f:control_spacing(), -- default spacing for the child rows

    f:row { -- first row contains radio buttons
        spacing = f:control_spacing(), -- use default spacing
        f:column {
            f:radio_button {
                title = "Button one",
                checked_value = "one", -- when control value matches this,
                                       -- the button is checked
                -- add value binding in next step
            },
```

```
                f:radio_button {
                    title = "Button two",
                    checked_value = "two",
                    -- add value binding in next step
                },
            },
        },

        f:row { -- second row shows a static text box
            f:static_text {
                text_color = LrColor( 1, 0, 0 ),
                -- add title with binding later
            },
        },
    },
```

5. For both buttons, add the following to bind the current value of both to the same key:

```
                -- add value binding in next step
                value = LrView.bind( "selectedButton" ),
```

   Now this key will reflect the user's choice of buttons; selecting a button will set the key value to "one" or "two".

6. Add the `title` for the static text box. Instead of binding it directly to the key value, we will transform that value into a display string. To do this, we make the argument of the `bind()` function a table, containing the key and a transform function:

```
            -- add title with binding later
            title = LrView.bind
                {
                    key = "selectedButton",
                    transform = function( value, fromTable )
                        -- body of function
                        end,
                }
```

7. Define the transform function as follows:

```
                    -- body of function
                    if value == "one" then -- first button is selected
                        return "Button one selected"
                    else
                        return "Button two selected"
                    end
```

8. Use `LrDialogs.presentModalDialog()` to display the new custom dialog. The argument is a table with entries for the dialog `title` and the view hierarchy that defines the `contents`:

```
local result = LrDialogs.presentModalDialog
    {
        title = "Custom Dialog Transform",
        contents = c, -- the view hierarchy we defined
    }
```

9. To call the function when the script runs, replace the call to `showCustomDialog()` at the bottom of the script with a call to the new function:

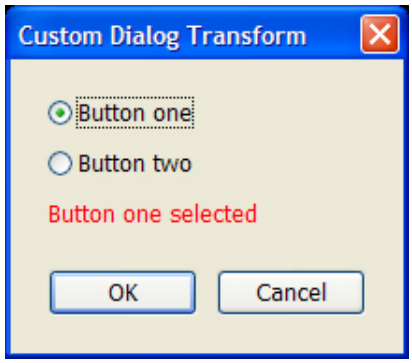
```
MyHWLibraryItem.showCustomDialogWithTransform()
```

10. Save your changes to the file.

Run the plug-in

Use these steps to run the plug-in and test the dialog:

1. Reload the plug-in, as described in step 1 on page 175.
2. Choose **Library > Plug-in Extras > Hello World Custom Dialog** to show the custom modal dialog created by the `LibraryMenuItem.lua` script:



3. Select the different buttons and notice how the text changes dynamically, reflecting your selection.
4. Dismiss the dialog with **OK** or **Cancel**.

Binding to multiple keys

This example redefines the custom dialog again, this time to demonstrate how you can bind your UI elements to more than one key, and to keys in more than one property table.

- This dialog will show how to update a numeric data value using a slider. It will update two data values, in two different tables, with two sliders.
- We will then bind a text field to the two keys, transforming the numeric values to text.
- Because the keys are in different tables, we will need to override the default table for the control by providing the table specification with the key specification.

This example also demonstrates a slightly more complex containment hierarchy, with some layout and appearance features.

Create multiple bindings to one key

Use these step to create the two tables and populate a dialog with this new set of controls.

1. Edit the `LibraryMenuItem.lua` file to create a new function, `showCustomDialogWithMultipleBind()`:

```
function MyHWLibraryItem.showCustomDialogWithMultipleBind()
    -- body of show-dialog function
end
```

2. In the body of this function, add code to create the function-context call you need for the property table:

```
-- body of show-dialog function
LrFunctionContext.callWithContext( "showCustomDialogWithMultipleBind",
                                   function( context )
    -- body of called function
end )
```

3. In this context, create two observable tables:

```
    -- body of called function
    local tableOne = LrBinding.makePropertyTable( context )
    local tableTwo = LrBinding.makePropertyTable( context )
```

4. Create a data key for each of the sliders, one in each table, with an initial numeric value:

```
    tableOne.sliderOne = 0
    tableTwo.sliderTwo = 50
```

5. At the top level, create the view hierarchy for the dialog. In this one, the root node is a column container with one row, and the controls in the row are grouped together using a group box container:

```
local f = LrView.osFactory() -- obtain the view factory object

local c = f:column {
    bind_to_object = tableOne, -- bind tableOne
    spacing = f:control_spacing(),
    f:row {
        f:group_box {
            title = "Slider One",
            font = "<system>",
            f:slider {
                value = LrView.bind( "sliderOne" ),
                min = 0,
                max = 100,
                width = LrView.share( "slider_width" )
            },

            f:edit_field {
                place_horizontal = 0.5,
                value = LrView.bind( "sliderOne" ),
                width_in_digits = 7
            },
        },

        f:group_box {
            title = "Slider Two",
            font = "<system>",
            f:slider {
                bind_to_object = tableTwo,
                value = LrView.bind( "sliderTwo" ),
                min = 0,
                max = 100,
                width = LrView.share( "slider_width" )
            },

            f:edit_field {
                place_horizontal = 0.5,
                bind_to_object = tableTwo,
```

```
            value = LrView.bind( "sliderTwo" ),
            width_in_digits = 7
        }
    },
},

f:group_box {
    fill_horizontal = 1,
    title = "Both Values",
    font = "<system>",

    f:edit_field{
        place_horizontal = 0.5,
        value = LrView.bind {
            -- Supply a table with table keys.
            keys = {
                   {
                       -- Only the key name is needed as sliderOne
                       -- in tableOne and that is already bound.
                       key = "sliderOne"
                   },
                   {
                       -- Supply the key and the table to which it belongs.
                       key = "sliderTwo",
                       bind_to_object = tableTwo
                   }
               },
            -- This operation creates the value for this edit_field.
            -- The bound values are accessed with the arg 'values'.
            operation = function( _, values, _ )
               return values.sliderTwo + values.sliderOne
            end
        },
        width_in_digits = 7
    },
}
}
```

6. For the `value` binding in the second slider, we will specify a different bound table, which overrides the default bound table for that control:

```
f:slider {
       bind_to_object = tableTwo,
       value = LrView.bind( "sliderTwo" ),
```

7. Do the same for the edit box in this group:

```
f:edit_field {
       bind_to_object = tableTwo,
       value = LrView.bind( "sliderTwo" ),
```

The two sliders are now bound to different keys in different tables; the user can change the numeric values using the sliders, and you can see the result in the associated text field for each one.

You will now add code to bind a third text box to a value derived from these two values.

8. To bind a value to multiple keys in different tables, you need to supply both the key name and the table in the binding, since the control can have only one default bound table.

Add this code to bind the value of the third edit box:

```
f:edit_field {
    -- add multi-key value binding later
    value = LrView.bind {
        keys = { -- specify the two bound keys
            {
                key = "sliderOne" -- in default table
            },
            {
                key = "sliderTwo",
                bind_to_object = tableTwo - specify a different table
            }
        },
        -- add operation
    }),
```

9. You must also supply the function that operates on the multiple key values to supply a single result for the binding. In this case, we will simply add the two numeric values, and return the result:

```
    -- add operation
    operation = function( binding, values, fromTable )
        return values.sliderTwo + values.sliderOne
    end
},
```

   Notice how you use the *values* argument passed to this function to access the value of each bound key. Whenever one of the key values changes, this function is automatically invoked; the return value becomes the result of the binding, and thus the `value` of the edit box.

10. Use `LrDialogs.presentModalDialog()` to display the new custom dialog, and call it when the script is run:

```
local result = LrDialogs.presentModalDialog {
       title = "Custom Dialog Multiple Bind",
       contents = c, -- the view hierarchy we defined
   }

MyHWLibraryItem.showCustomDialogWithMultipleBind()
```

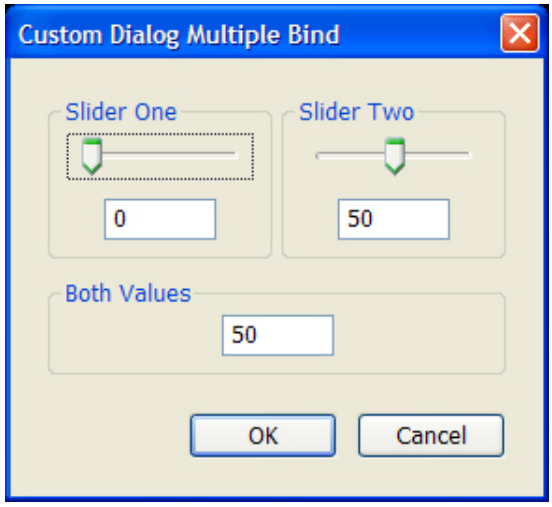
11. Save your changes to the file.

Run the plug-in

Use these steps to run the plug-in and text the dialog:

1. Reload the plug-in, as described in step 1 on page 175.

2. Choose **Library > Plug-in Extras > Hello World Custom Dialog** to show the custom modal dialog created by the `LibraryMenuItem.lua` script:



3. Move the sliders and notice how the text below them changes, reflecting the current value for each numeric property, and how the sum of the two values is displayed in the "Both Values" box.

4. Dismiss the dialog with **OK** or **Cancel**.

Adding a data observer

The most flexible way of connecting to your data is to create an *observer* for the property table; this is an independent object that is notified of changes in the table, and can take any action in response to the change, including setting UI values in any way you want.

This example demonstrates how to set up an observer that is notified when a data value changes, and how to define a function that responds to that notification by setting UI values.

Set up the dialog and table

1. Edit the `LibraryMenuItem.lua` file to create a new function, `showCustomDialogWithObserver()`:

```
function MyHWLibraryItem.showCustomDialogWithObserver()
    -- body of function
end
```

2. Within this function, make the function-context call you need for the property table:

```
LrFunctionContext.callWithContext( "showCustomDialogWithObserver",
                                    function( context )
    -- body of function
end )
```

3. In this context, create the observable table, and add a property named `myObservedString`, with an initial value:

```
    -- body of function
    local props = LrBinding.makePropertyTable( context )
    props.myObservedString = "This is a string" -- new prop with initial value
```

4. Obtain a view factory and use it to create a static text field, which initially displays the static value of the property. (We will put it into the view hierarchy later.)

```
local f = LrView.osFactory() -- obtain the view factory object

local showValue_st = f:static_text { -- create the label control
          title = props.myObservedString,-- set text to the static property value
          text_color = LrColor( 1, 0, 0 ) -- set color to a new color object
       }
```

   The `title`, which is the displayed text, is assigned to be the current value of the property we defined in the data table, `props.myObservedString`. This is not a dynamic binding, just an assignment to the current value. So far, if the property value changes, it will not change the text in the control.

5. Create an edit box (which we will also add to the view hierarchy later). Notice that this box updates its `value` with every keystroke:

```
local updateField = f:edit_field { -- create an edit box
    value = "Enter some text", -- initial text, not bound to data
    immediate = true -- update value with every keystroke
}
```

Create an observer for a data property

To make the text dynamic, we are going to add an observer for the `props.myObservedString` key. The observer is notified whenever the observed property changes; we will tell it to update the text in `showValue_st`.

6. When the observer receives a notification it invokes a function. Create the function that will be used by the observer:

```
local myCalledFunction = function()
    showValue_st.title = updateField.value -- reflect the value entered in edit box
    showValue_st.text_color = LrColor( 1, 0, 0 ) -- make the text red
end
```

   This makes the `showValue_st` text dynamic, by resetting its `title` value when the observed property changes. It also turns the text red to show that it has fired.

7. Now add the observer to the observable table. This associates the function with a specific property in the table:

```
props:addObserver( "myObservedString", myCalledFunction )
```

   This observer is notified, and calls the response function, whenever the value of the key `myObservedString` is modified.

Create the dialog contents

Now you will create the view hierarchy that defines the custom-UI portion of the dialog. This one uses a `column` as the top-level container, which contains two rows, which in turn contain the visible controls. In this case, the controls include the `showValue` text box and the `updateField` edit box that we already defined, along with additional labels and a push button.

8. Create the view hierarchy:

```
local c = f:column { --The top-level container, arranges all the rows vertically
          f:row { -- a group of labels arranged horizontally
              fill_horizontal = 1, -- the row fills its parent's width
              f:static_text { -- add a right-aligned label
                  alignment = "right",
                  width = LrView.share "label_width", -- all get the same width
                  title = "Bound value: "
              },
              showValue_st, -- the text box we already defined
          }, -- end f:row

          f:row { -- another group, a labeled edit box and button
              f:static_text {
                  alignment = "right",
                  width = LrView.share "label_width", -- shared with other label
                  title = "New value: "
              },
              updateField, -- the edit box we already defined
              -- add push button
          }, -- end row
      } -- end column
```

9. We will add one more element, a push button. This demonstrates another way to define the behavior of your UI, by specifying a direct action to be taken in response to clicking the button. In this case, the

button action resets the observed property value to the value entered by the user in the edit box. It also resets the color of the static text to black, so that we will be able to tell whether the observer function fired.

Add this code:

```
-- add push button
f:push_button {
    title = "Update",
    action = function() -- when clicked, reset values in other controls
        showValue_st.text_color = LrColor( 0, 0, 0 ) -- make text black
        props.myObservedString = updateField.value -- reset data value
                                                   -- from current entered value
    end
},
```

10. Use `LrDialogs.presentModalDialog()` to display the new custom dialog, and call it when the script is run:

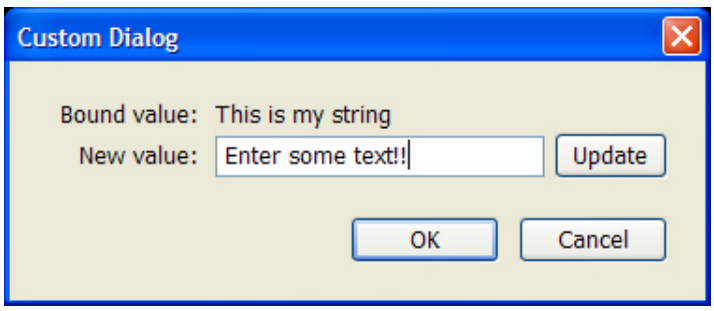
```
local result = LrDialogs.presentModalDialog {
       title = "Custom Dialog",
       contents = c, -- the view hierarchy we defined
   }

MyHWLibraryItem.showCustomDialogWithObserver()
```

11. Save your changes to the file.

Run the plug-in

Use these steps to run the plug-in and see the observer and the button in action:

1. Reload the plug-in, as described in step 1 on page 175.
2. Choose **Library > Plug-in Extras > Hello World Dialog** to show the predefined modal dialog created by the `ExportMenuItem.lua` script.
3. Click **OK** to dismiss the dialog.
4. Choose **Library > Plug-in Extras > Hello World Custom Dialog** to show the custom modal dialog created by the `LibraryMenuItem.lua` script:



5. Enter some text into the “New value” field.
6. Click **Update**. Notice the “Bound value” text changes to whatever text you entered, and the text turns red.

7. Click **Update** again, without changing the text in the "New value" field. Notice how the text turns black. This is because the observer is only notified when the bound value changes.

8. Dismiss the dialog with **OK** or **Cancel**.

Debugging your plug-in

The Lightroom SDK does not supply a development environment for you to debug your plug-ins. You can write your plug-ins using any text editor, and write debugging output using the `LrLogger` namespace.

The SDK does not provide a facility to view the debugging output directly; you can write out a log file to disk, or use a third-party application, such as one of these tools:

- WinDbg — available for download from http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx
- Microsoft Developer Studio
- Console — built-in application on Mac OS, look in /Applications/Utilities
- Xcode

Specifying a log

Use these steps to add trace information to the Hello World plug-in:

1. Edit the `LibraryMenuItem.lua` file to import the `LrLogger` namespace:

```
local LrLogger = import 'LrLogger'
```

2. After the `import` statements, create a new logger instance named `libraryLogger` and enable the `print` or `logfile` action:

```
local myLogger = LrLogger( 'libraryLogger' )
myLogger:enable( "print" ) -- or "logfile"
```

   - Choose `print` if using a console log viewing tool; see "Viewing trace information in a platform console" on page 187.
   - Choose `logfile` if using a text file for debugging; see "Viewing trace information using log files" on page 187

3. Create a function named `MyHWLibraryItem.outputToLog()` that accepts a single string argument. In the body of the function, send the accepted argument to the `LrLogger:trace()` function:

```
function MyHWLibraryItem.outputToLog( message )

    myLogger:trace( message )

end
```

4. Add trace information to the `myCalledFunction` function. Add the following code:

```
MyHWLibraryItem.outputToLog( "props.myObservedString has been updated." )
```

5. Within the action function for the Update button, add the following trace information:

```
MyHWLibraryItem.outputToLog( "Update button clicked." )
```

6. Save your changes, and reload your plug-in if necessary.

Viewing trace information using log files

These steps describe how to view debugging trace information using a text editor:

1. Start Lightroom.
2. Make sure your plug-in is configured to write debugging information to a log file, as described above:

```
local LrLogger = import 'LrLogger'
local myLogger = LrLogger( 'libraryLogger' ) -- the log file name
myLogger:enable( "logfile" )
function MyHWLibraryItem.outputToLog( message )
    myLogger:trace( message )
end
```

3. Once your plug-in has generated output, look for a the output file with the name you specified and the `.txt` extension ("`libraryLogger.txt`" in this example).
    - In Windows, the file is located in your My Documents folder.
    - In Mac OS, the file is located in the Documents folder inside your home directory.

More advanced text editors will automatically notice and update their display when the file has changed; you may want to use such a text editor.

In Mac OS, you may find it simpler to open a Terminal window and use the `tail` command to watch the file by typing a command such as:

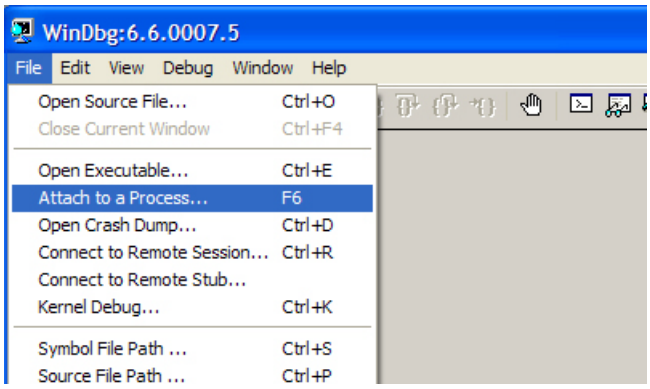
```
tail -f ~/Documents/libraryLogger.txt
```

Viewing trace information in a platform console

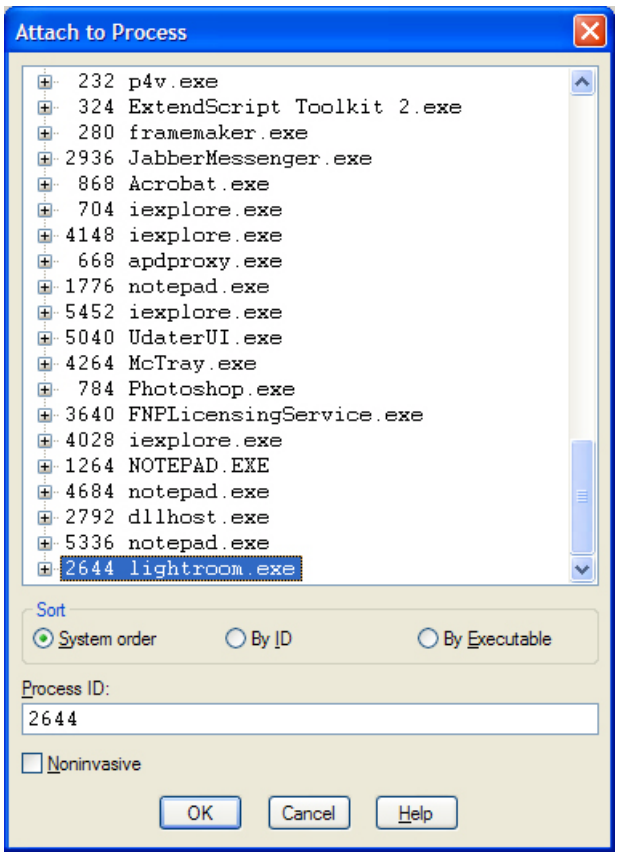
If you choose to print debugging information by setting `LrLogger:enable( "print" )`, you must use a platform-specific debugging tool to view the debugging trace information.

Debugging in Windows WinDbg

1. Start Lightroom.
2. Start WinDbg.
3. In WinDbg, choose **File > Attach to a Process**.



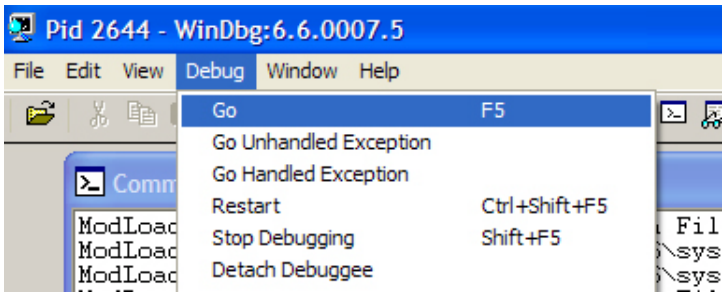
4. In the Attach to Process dialog, scroll through the processes and look for `lightroom.exe`.



5. Select `lightroom.exe` and click **OK**.

   A console window appears in WinDbg, and the Lightroom application is blocked.

6. In WinDbg, choose **Debug > Go**.



7. In Lightroom, run the Hello World plug-in (see "Run the plug-in" on page 185).

8. In WinDbg, view the console to see the trace information being written as you use the plug-in.

```
L: LrSdkMenus      TRACE running SDK menu item ID      ag.sdk.LrLibraryMenuItems.2
L: libraryLogger   TRACE Update button clicked.
L: libraryLogger   TRACE props.myObservedString has been updated.
L: libraryLogger   TRACE Update button clicked.
L: libraryLogger   TRACE Update button clicked.
L: libraryLogger   TRACE Update button clicked.
L: libraryLogger   TRACE props.myObservedString has been updated.
L: libraryLogger   TRACE Update button clicked.
```

Debugging in Mac OS Console

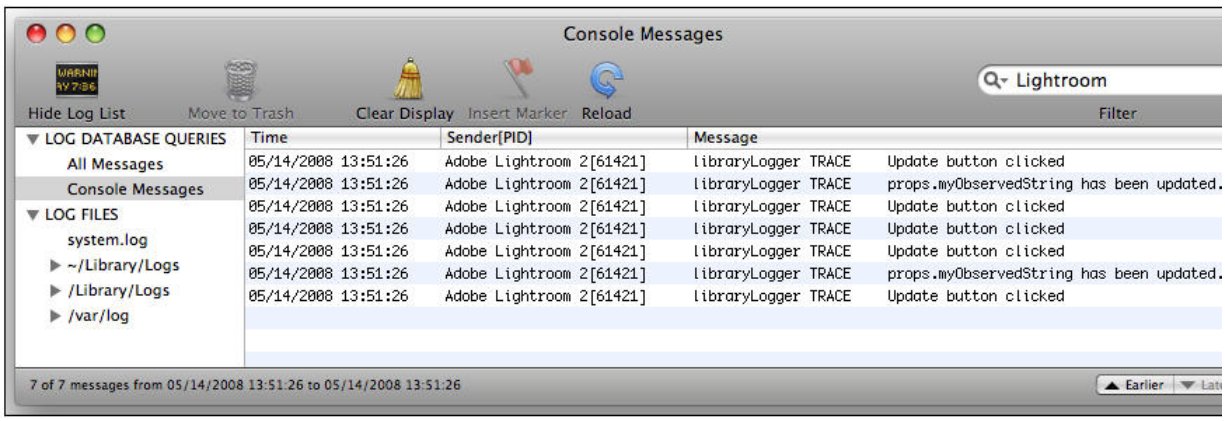
1. Start Lightroom

2. Start Console. The default location is **Applications > Utilities > Console**.

3. In Lightroom, run the Hello World plug-in (see "Run the plug-in" on page 185).

4. View the console to see the trace information being written as you use the plug-in.

5. You can type the word "Lightroom" into the Filter box in the upper right corner of the Console window, to suppress log messages from other applications.



10 Defining Metadata: A Walkthrough

This chapter shows how a plug-in can define metadata fields that Lightroom can display along with standard metadata for photos, and which you can use as a private data model for plug-in processing. It also illustrates how the private data can be used to customize the Plug-in Manager.

These concepts and techniques are introduced and explained in more detail in Chapter 3, "Creating Export and Publish Services."

Adding custom metadata

This simple example plug-in demonstrates how to define your own metadata fields to be associated with photos. Your fields can be displayed in the Lightroom Metadata panel, where users can set or modify their values. You can also define private data for your plug-in, which is not displayed to users.

First, we will create the framework for the plug-in, which is similar to that for an export plug-in.

1. Create a new folder in your chosen directory called `myMetadata.lrplugin`.
2. In this folder, create three new files

```
Info.lua
MyMetadataDefinitionFile.lua
MyMetadataTagset.lua
```

3. Edit the `Info.lua` file to add the following code:

```
return {
    LrSdkVersion = 5.0,

    LrToolkitIdentifier = 'sample.metadata.mymetadatasample',
    LrPluginName = LOC "$$$/MyMetadataSample/PluginName=My Metadata Sample",

    LrMetadataProvider = 'MyMetadataDefinitionFile.lua',
    LrMetadataTagsetFactory = 'MyMetadataTagset.lua',
}
```

Define metadata fields

Now we are going to create the custom metadata fields in the definition file.

4. Open the file `MyMetadataDefinitionFile.lua` and add the following code as the initial framework:

```
return {
    metadataFieldsForPhotos = {
    },

    schemaVersion = 1,
}
```

The schema-version value provides version control; it can be incremented to notify users of changes to the plug-in.

5. The `metadataFieldsForPhotos` table is where we define our new custom metadata fields.

   Add this first entry to the table:

```
metadataFieldsForPhotos = {
    {
        id = 'siteId',
    },
},
```

   This is the simplest type of field. It does not have any of the properties that make it visible in the Metadata panel, or modifiable by users. It is an internal field that a plug-in can use as private data. Other plug-ins can also access such a field, but they cannot write to it.

6. Now we will add a field that will be public:

```
metadataFieldsForPhotos = {
    {
        id = 'siteId',
    },
    {
        id = 'myString',
        -- add properties
    },
},
```

7. To make this new field available to edit within the Lightroom Metadata panel, we need to add a title and data type:

```
metadataFieldsForPhotos = {
    {
        id = 'siteId',
    },
    {
        id = 'myString',
        title = LOC "$$$/MyMetadataSample/Fields/MyString=My String",
        dataType = 'string',
    },
},
```

   The `title` property provides a localizable display string to be shown in the Metadata panel. Simply specifying this property makes the field visible.

   The `dataType` property tells the Metadata panel how to display the property, so as to make it editable. Because this is a simple string value, it will be shown in an editable text field. This property is optional, and "string" is the default type, so the result is the same if you leave it out.

8. Add one more property:

```
metadataFieldsForPhotos = {
    {
        id = 'siteId',
    },
    {
        id = 'myString',
        title = LOC "$$$/MyMetadataSample/Fields/MyString=My String",
        dataType = 'string',
        searchable = true
    },
},
```

Setting `searchable` to true allows you to search for images using this custom metadata field.

9. Add another entry to the table to define a Boolean field. To do this, we will use the enumerated-value data type.

```
{
    id = 'myboolean',
    title = LOC "$$$/MyMetadataSample/Fields/Display=My Boolean",
    dataType = 'enum',
    values = {
        -- add valid-value entries
    },
},
```

10. Now we will limit the possible values to the strings “true” and “false”.

```
{
    id = 'myboolean',
    title = LOC "$$$/MyMetadataSample/Fields/Display=My Boolean",
    dataType = 'enum',
    values = {
        {
            value = 'true',
            title = LOC "$$$/MyMetadataSample/Fields/Display/True=True",
        },
        {
            value = 'false',
            title = LOC "$$$/MyMetadataSample/Fields/Display/False=False",
        },
    },
},
```

Because we have declared the value type as `enum`, the Metadata panel displays this field with a pop-up menu of valid values. Each value has a localizable display string, which appear in the menu. When the user chooses the menu item, the field is assigned the corresponding string value.

11. Save this file.

Define a tagset

The drop-down menu at the top left of the Metadata panel allows users to filter what is shown in the panel, by selecting a *metadata tagset* to be displayed. There are predefined tagsets, and you can also create your own. See “Adding custom metadata tagsets” on page 76.

Now that we have defined a set of metadata fields, we will create a tagset for them, so that they can be selected for display, and displayed together in a labeled section of the Metadata panel. Our tagset will also include some predefined sets.

1. Open the file `MyMetadataTagset.lua`.

2. Edit the file to add this initial code structure:

```
return {

    title = LOC "$$$/MyMetadataSample/Tagset/Title=My Metadata",
    id = 'MyMetadataTagset',
```

```
    items = {
        -- add item entries
    },
}
```

- The `title` value is the localizable display string that will show up as the menu item for this tagset.
- The `items` table provides the specific metadata fields to be included in our tagset. We will add some representative fields. For the complete list of possible field specifiers, see "Defining metadata fields" on page 72.

3. Add the following entries to the `items` table, to include a labeled section named Standard Metadata, which displays the predefined `filename` and `folder` metadata fields, part of the built-in metadata for Lightroom:

```
items = {
    -- add item entries
    { 'com.adobe.label',
        label = LOC "$$$/Metadata/OrigLabel=Standard Metadata" },
    'com.adobe.filename',
    'com.adobe.folder',

    'com.adobe.separator',
},
```

   This labeled section is followed by a separator.

4. Add the entries for the custom metadata defined in this plug-in, in another labeled section:

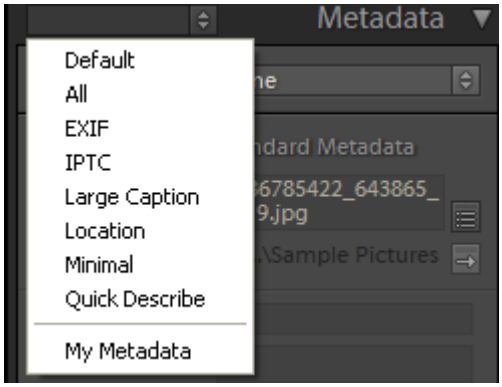
```
items = {
    { 'com.adobe.label',
        label = LOC "$$$/Metadata/OrigLabel=Standard Metadata" },
    'com.adobe.filename',
    'com.adobe.folder',
    'com.adobe.separator',

    { 'com.adobe.label', label = LOC "$$$/Metadata/CusLabel=My Metadata" },
    'sample.metadata.mymetadatasample.*',
},
```

   The asterisk wild-card character in the field-name part of the path matches all fields defined by this plug-in. The asterisk can appear only at the end of the field name.

5. Save this file.

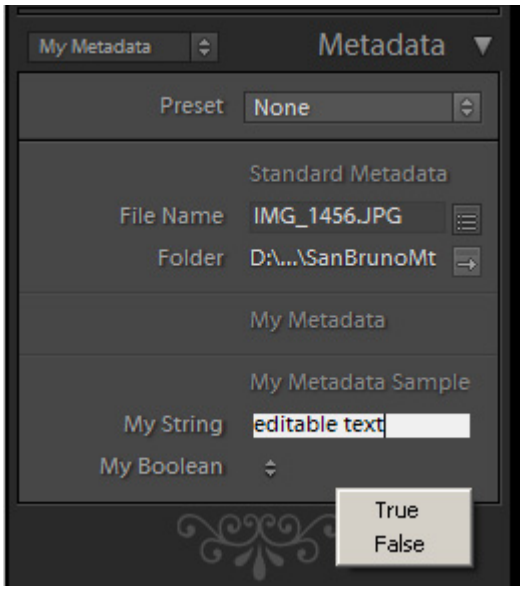
Using the plug-in

1. Open Lightroom and go to **File > Plug-in Manager**.
2. In the Plug-in Manager dialog, click **Add**.
3. Navigate to your new plug-in folder. Check that your plug-in is loaded and running, as shown by a green traffic-light icon, and the text "Installed and running". (If it is not, check the Plug-in Author Tools section of the Plug-in Manager for a diagnostic message.)
4. Select your plug-in and click **Choose Selected**.
5. In the Library module, show the Metadata panel and click the name at the top to see the drop-down menu. Your new tagset, with the label "My Metadata," should appear at the bottom of the list.



6. Select the My Metadata tagset.

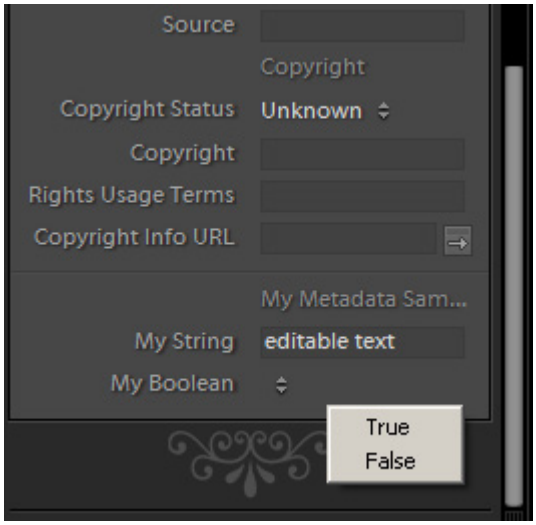
   The Metadata panel should display the `filename` and `folder` fields in a section labeled Standard Metadata, and your custom `myString` and `myBoolean` fields in a section labeled My Metadata Sample, with separators between the sections. The fields are shown with their display labels, and an edit or selection control.



7. Try editing the custom fields. The `myString` field, labeled My String, has an editable text field for setting the value, and the `myBoolean` field, labeled My Boolean, has a pop-up menu that shows the allowed values.

8. Select the All tagset.

   Notice that your custom metadata now appears at the bottom of the panel, after all of the standard metadata.



Customizing the Plug-in Manager

You plug-in can define a customized section for the Plug-in Manager dialog, which is displayed on the right when the user selects your plug-in in the list on the left. This section can make use of private data values that you make globally available within the plug-in by defining them in an initialization script.

Here is an example of adding such a section, using the metadata values we have already defined.

1. In the `Info.lua` file, add the entry that identifies the Plug-in Info Provider definition script:

```
return {
    LrSdkVersion = 5.0,

    LrToolkitIdentifier = 'sample.metadata.mymetadatasample',
    LrPluginName = LOC "$$$/MyMetadataSample/PluginName=My Metadata Sample",

    LrMetadataProvider = 'MyMetadataDefinitionFile.lua',
    LrMetadataTagsetFactory = 'MyMetadataTagset.lua',

    LrPluginInfoProvider = 'PluginInfoProvider.lua',
}
```

2. Add another line that identifies a URL where the user can go for further information about this plug-in:

```
    LrPluginInfoUrl = "http://www.mycompany.com",
```

   This URL will be displayed in the standard Status section of the Plug-in Manager dialog.

3. Create two new files in the plug-in folder named `PluginInfoProvider.lua` and `PluginManager.lua`.

4. Edit the file `PluginInfoProvider.lua` to add the basic framework:

```
require 'PluginManager'

return {
    sectionsForTopOfDialog = PluginManager.sectionsForTopOfDialog,
}
```

5. The section definition will use variables defined in an initialization script. In the `Info.lua` file, add the `LrInitPlugin` entry that identifies the plug-in initialization script:

```
return {
    LRSDKVersion = 5.0,

    LrToolkitIdentifier = 'sample.metadata.mymetadatasample',
    LrPluginName = LOC "$$$/MyMetadataSample/PluginName=My Metadata Sample",

    LrInitPlugin = 'PluginInit.lua',

    LrMetadataProvider = 'MyMetadataDefinitionFile.lua',
    LrMetadataTagsetFactory = 'MyMetadataTagset.lua',

    LrPluginInfoProvider = 'PluginInfoProvider.lua',
}
```

6. Create the file `PluginInit.lua` in the plug-in folder, and edit it to add these variables:

```
_G.currentDisplayImage = "no"
_G.pluginID = "com.adobe.lightroom.sdk.metadata.custommetadatasample"
_G.URL = "http://www.mycompany.com"
```

   The `_G` prefix here indicates that these variables are globally available within the plug-in.

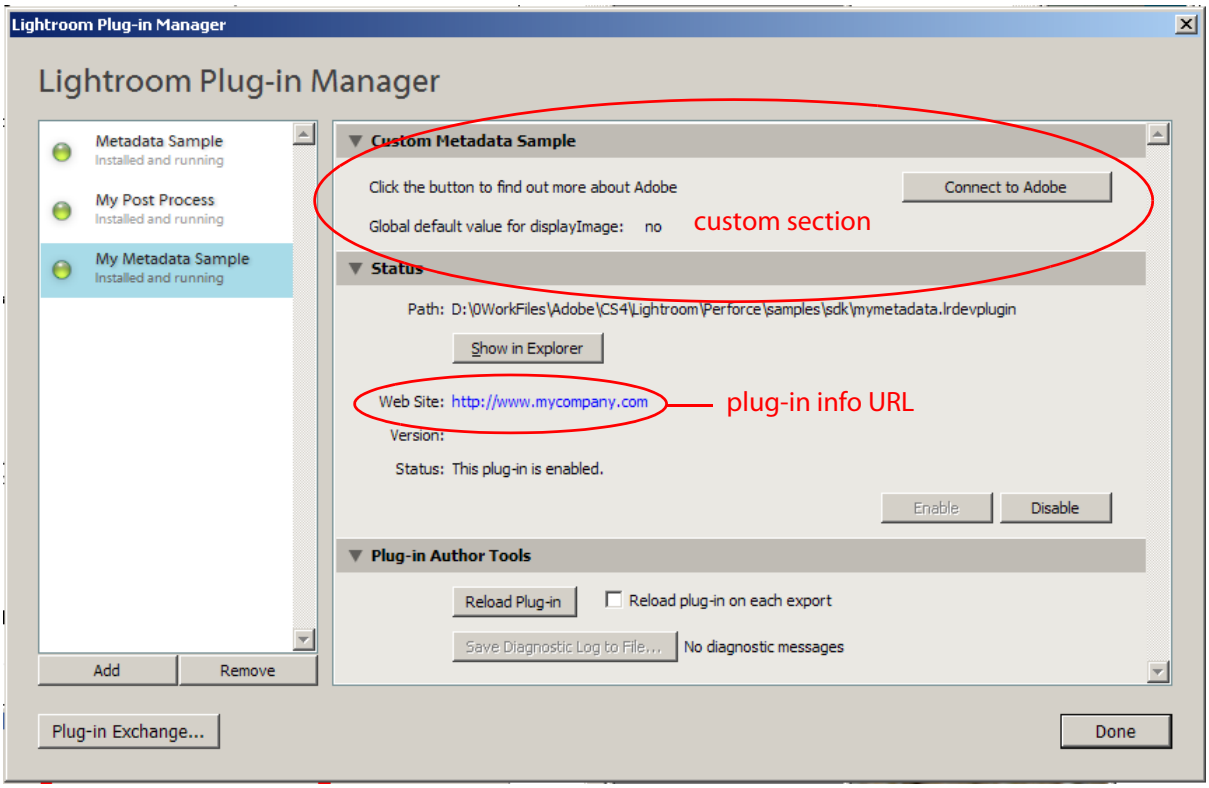
7. Edit the file `PluginManager.lua` to define the function that creates the UI content of the new section. Notice the use of the variables we defined in the initialization script:

```
local LrView = import "LrView"
local LrHttp = import "LrHttp"
local bind = import "LrBinding"
local app = import 'LrApplication'

PluginManager = {}

function PluginManager.sectionsForTopOfDialog( f, p )
    return {
           -- section for the top of the dialog
           {
               title = "Custom Metadata Sample",
               f:row {
                   spacing = f:control_spacing(),
                   f:static_text {
                       title = 'Click the button to find out more about Adobe',
                       alignment = 'left',
                       fill_horizontal = 1,
                   },
                   f:push_button {
                       width = 150,
                       title = 'Connect to Adobe',
                       enabled = true,
                       action = function()
                           LrHttp.openUrlInBrowser(_G.URL)
                       end,
                   },
               },
               f:row {
                   f:static_text {
                       title = 'Global default value for displayImage: ',
                       alignment = 'left',
                   },
                   f:static_text {
```

```
                    title = _G.currentDisplayImage,
                    fill_horizontal = 1,
                },
            },
        },
    }
end
```

8. Reload and run the plug-in again, as described in "Using the plug-in" on page 193.

   When you select the plug-in, the new section appears above the standard Lightroom sections:



11 Web Gallery Plug-ins: A Tutorial Example

This chapter provides a walkthrough example of how to build a Web Gallery plug-in, which uses a slightly different architecture from standard export and metadata plug-ins.

This sample code produces a simple HTML gallery that shows a grid of thumbnail images, which respond to a click by showing a larger version of the clicked image.

These concepts are introduced and explained in detail in Chapter 6, "Writing a Web-engine Plug-in."

Creating a Web Gallery plug-in

To begin creating the plug-in, we will create initial versions of the required files, then add code to them as we go on.

1. Create a single folder, `mySamplePlugin.lrwebengine`, to hold the plug-in files, in the following folder according to your operating system:

   - In Mac OS:

     ```
     userhome/Library/Application Support/Adobe/Lightroom/
        Web Galleries/mySamplePlugin.lrwebengine
     ```

   - In Windows:

     ```
     LightroomRoot\shared\webengines\mySamplePlugin.lrwebengine
     ```

Add descriptive files

1. In the `myWebPlugin` folder, create the *information* file that describes the plug-in, naming the file `galleryInfo.lrweb`.

   Add this initial Lua code to the file:

   ```
   return {
      LRSDKVersion = 5.0,
      LrSdkMinimumVersion = 2.0, -- minimum SDK version required by this plug-in

      title = "My Sample Plug-in",
      id = "com.adobe.wpg.templates.mysample",
      galleryType = "lua",
      maximumGallerySize = 50000,
   }
   ```

2. In the `myWebPlugin` folder, create the *manifest* file that defines the contents of the plug-in, naming it `manifest.lrweb`. Include the first command, which specifies a template for a gallery page:

1. AddGridPages {

   ```
      template = "grid.html",
      rows = 4,
      columns = 4,
   }
   ```

Add HTML template files

1. The manifest references an HTML file to be used as a template, `grid.html`. Create this file in the `myWebPlugin` folder, putting in these initial references:

```
<%@ include file="header.html" %>
<%@ include file="footer.html" %>
```

2. The two referenced HTML files contain common code for all HTML pages that will be created from this template. Create these two HTML files in the `myWebPlugin` folder.

   - The content of `header.html` initializes the HTML code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="generator" content="Adobe Photoshop Lightroom" />
    <title>My Sample Plug-in</title>
    <link rel="stylesheet" type="text/css" media="screen"
       title="Custom Settings" href="$others/custom.css" >
</head>

<body>
```

   - The content of `footer.html` simply closes off the HTML code:

```
</body>
</html>
```

Add subfolders

The plug-in folder now contains most of the top-level files for the plug-in:

```
myWebPlugin/
    galleryInfo.lrweb
    manifest.lrweb

    grid.html
    header.html
    footer.html
```

The `grid.html` page is the template for the thumbnail filmstrip; later we will add a template HTML file for the large version of the selected thumbnail image.

In addition to these top-level files, the plug-in will require resources of various kinds; default images, style sheets, JavaScript support code, and string dictionaries for localization.

1. Before going on to flesh out the content of the basic files, add some subfolders to hold resource files:

```
myWebPlugin/
    resources/
        css/
        js/
    strings/
        en/
```

Defining a data model and functionality

The information file defines the data model that your LuaPage code will use to store the information it needs to generate appropriate HTML. In this case, there will be two image sizes on the gallery pages, a small and a large version of each photo, so the model needs to define that basic parameters for each image size.

1. Add a `model` entry to the table returned by the `galleryInfo.lrweb` file:

```
return {
    LRSDKVersion = 5.0,
    LrSdkMinimumVersion = 2.0, -- minimum SDK version required by this plug-in

    title = "My Sample Plug-in",
    id = "com.adobe.wpg.templates.mysample",
    galleryType = "lua",
    maximumGallerySize = 50000,

    model = {
        ["nonDynamic.imageBase"] = "content",

        ["photoSizes.thumb.height"] = 150,
        ["photoSizes.thumb.width"] = 150,
        ["photoSizes.thumb.metadataExportMode"] = "copyright",

        ["appearance.thumb.cssID"] = ".thumb",
    },
}
```

   This begins to define the parameters for the smallest photo size, naming it "thumb". The variables define the images size, allowing us to adjust the number of rows and columns the grid will need to display them.

2. The `model.appearance` parameter associates the "thumb" photos with a style-sheet variable. To make this work, we have to add the style sheet to the project.

   In the `manifest.lrweb` file, add this code at the top:

```
AddCustomCSS {
    filename = 'content/custom.css',
}
```

3. Now we will add some code to the HTML template that makes use of these variables to display thumbnail images in the workspace.

   In the `grid.html` file, add this code before the header statement, defining local variables:

```
<%
--[[ Define some variables to make locating other resources easier. ]]
    local mySize = "thumb"
    local others = "content"
    local theRoot = "."
%>
```

Add a grid using built-in tags

The part of the HTML template that defines the grid itself, and the contents of each cell, makes use of the predefined grid tags contained in the Lightroom built-in tagset. First, we need to import this tagset into the "`lr`" namespace.

4. In the `manifest.lrweb` file, add this code at the top:

```
importTags( "lr", "com.adobe.lightroom.default" )

AddCustomCSS {
    filename = 'content/custom.css',
}
```

5. Now we can define the contents of image grid and its contents.

   Add this code to `grid.html` between the header and footer include statements:

```
<lr:ThumbnailGrid>
    <lr:GridPhotoCell>
        <img src="$others/bin/images/thumb/<%= image.exportFilename %>.jpg"
            id="<%= image.imageID %>" class="thumb" />
     </lr:GridPhotoCell>
</lr:ThumbnailGrid>
```

   The logic here retrieves the thumbnail version of an image from the `content` folder (`$others`), and makes it the content of a grid cell. The image name is variable, so each cell shows a different image.

This data model and template define a page that displays thumbnail versions of your images in the Web Gallery workspace. Next, we will need to allow for more than one page of photos, and add functionality so that clicking on a thumbnail shows the larger version of the image.

Add pagination using built-in tags

We will use more of the predefined tags to add page-navigation buttons, allowing the example to accommodate more than 16 images.

6. In the `grid.html` page, add this code after the closing `ThumbnailGrid` tag:

```
...
</lr:ThumbnailGrid>

<% if numGridPages > 1 then %>
 <div class="pagination">
    <ul>
        <lr:Pagination>
            <lr:CurrentPage>
                <li>$page</li>
            </lr:CurrentPage>
            <lr:OtherPages>
                <li><a href="$link">$page</a></li>
            </lr:OtherPages>
            <lr:PreviousEnabled>
                <li><a href="$link">Previous</a></li>
            </lr:PreviousEnabled>
            <lr:PreviousDisabled>
                <li>Previous</li>
            </lr:PreviousDisabled>
            <lr:NextEnabled>
```

```
            <li><a href="$link">Next</a></li>
          </lr:NextEnabled>
          <lr:NextDisabled>
            <li>Next</li>
          </lr:NextDisabled>
        </lr:Pagination>
      </ul>
    </div>
  <% end %>
```

The pages make use of the predefined navigation buttons (using the `$link` variable), associating them with the text "Previous" and "Next". This code makes sure that the "Previous" button is disabled on the first page, and the "Next" button is disabled on the last page.

Each page also displays its own page number (using the `$page` variable), and allows direct navigation to other pages (using the `$link` and `$page` variables)

Add another photo size

When you click on a thumbnail image in the filmstrip, we want to display a large version of that image in another frame of the page. In order to make this happen, we need to do several things:

- Define a "large" photo size in the data model.
- Create an HTML template for the large-image display frame.
- Add a link around each thumbnail image that responds to a click by finding and displaying the corresponding large image.

Add the "large" photo size

1. The data model needs to define the new photo size and its supporting parameters.

   In the `galleryInfo.lrweb` file, add these items to the `model` table:

   ```
   model = {
       ...
       ["photoSizes.thumb.metadataExportMode"] = "copyright",

       ["photoSizes.large.width"] = 450,
       ["photoSizes.large.height"] = 450,

       ["appearance.thumb.cssID"] = ".thumb",
   },
   ```

2. The project needs to include a template for the frame that displays the large image.

   In the `manifest.lrweb` file, add this command:

   ```
   AddPhotoPages {
       template = 'large.html',
       variant = '_large',
       destination = "content",
   }
   ```

   This will create an individual HTML page for each large image, which we can link to from the grid photo cell definition in `grid.html`. The name of each page has the text `"_large"` appended to it; for example, `img0731_large.html`.

Add link functionality

3. In the `grid.html` template, add a link in each photo cell of the grid that retrieves the large-version page corresponding to the thumbnail in that cell:

```
<lr:GridPhotoCell>
  <a href="$others/<%= image.exportFilename %>_large.html">
    <img src="$others/bin/images/thumb/<%= image.exportFilename %>.jpg"
       id="<%= image.imageID %>" class="thumb" />
  </a>
</lr:GridPhotoCell>
```

Notice that the reference to the filename includes the appended text, `"_large"`.

Define the large-image frame template

4. Create the new HTML template page, `large.html`, in the top-level plug-in folder. The new page is similar to the `grid.html` page, except that it declares the use of large images, rather than thumbnails, and there is an `image` function that retrieves the single image to be shown.

Use the common header and footer code, and define local variables:

```
<%
--[[ Define some variables to make locating other resources easier.]]
    local image = getImage( index )
    local theRoot = ".."
    local others = "."
    local mySize = "large"
%>

<% --[[ Include the page header]] %>
<%@ include file="header.html" %>

<% --[[ ...add pagination logic...]] %>

<% --[[ ...add image-display link...]] %>

<% --[[ Include the page footer]] %>
<%@ include file="footer.html" %>
```

5. Add this pagination code after the include-header section. This version includes an Index option which takes the site back to the grid page:

```
<div>
    <ul>
      <lr:Pagination>
        <lr:PreviousEnabled>
          <li><a href="$link">Previous</a></li>
        </lr:PreviousEnabled>
        <lr:PreviousDisabled>
          <li>Previous</li>
        </lr:PreviousDisabled>

        <li><a href="$gridPageLink">Index</a></li>

        <lr:NextEnabled>
          <li><a href="$link">Next</a></li>
        </lr:NextEnabled>
        <lr:NextDisabled>
          <li>Next</li>
```

```
        </lr:NextDisabled>
      </lr:Pagination>
    </ul>
</div>
```

6. After the pagination code, add the link that actually retrieves the large image to be shown:

```
<a href="$gridPageLink">
  <img src="bin/images/large/<%= image.exportFilename %>.jpg" />
</a>
```

Customizing the Web Gallery UI

The next step is to customize the control panel of the Web Gallery page so that users can adjust values in the model data that we have defined, and allow changes that the user makes to be shown immediately in the Web Gallery browser using Live Update.

To demonstrate these techniques, we will add an entry to the Site Info section of the panel that will allow users to modify the main title of the page, shown in the main frame. We will allow the user to edit the title string using a text-edit control in the Site Info section of the main control panel, or using in-place edit in the preview panel.

Add a binding to a control

1. Add a variable to the data model to hold the title text. In the galleryInfo.lrweb file, add these lines to the `model` entry:

```
["metadata.siteTitle.value"] = "MySample",
["appearance.siteTitle.cssID"] = "#siteTitle",
```

   The second value associates the title with a style-sheet ID.

2. Add a `views` entry to the table returned by the `galleryInfo.lrweb` file:

```
views = function( controller, f )
    local LrView = import "LrView"
    local bind = LrView.bind
    local multibind = f.multibind
    return {
        labels = f:panel_content {
            bind_to_object = controller,
            f:subdivided_sections {
                f:labeled_text_input {
                    title = "MySample",
                    value = bind "metadata.siteTitle.value",
                },
            },
        },
    }
end,
```

   The Site Info section of the control panel on the Web Gallery page corresponds to the `labels` return value of the `views` function. We are creating a **`labeled_text_input`** control in this section, and binding its value to the data-model value that holds the site-title text.

Add the title to the HTML template

The title will be shown on all pages, so it should be part of the boilerplate HTML used at the front of all pages, as defined by the `header.html` template file.

3. Edit the `header.html` template file, adding the following heading immediately following the `<body>` tag:

   ```
   <h1 onclick="clickTarget( this, 'metadata.siteTitle.value' );"
          id="metadata.siteTitle.value">$model.metadata.siteTitle.value</h1>
   ```

   Notice that the title text is found in the model variable, the same one bound to the text-input control in the control panel. An event handler here allows edit-in-place in the browser window of the Web Gallery page, in addition to the editing capability provided by UI control.

4. To make edit-in-place work, we need a JavaScript script to handle Live Update. Add the next statement immediately following, identifying a script to be executed

   ```
   <script type="text/javascript" src="$theRoot/resources/live_update.js"></script>
   ```

5. We need to make this script part of the plug-in. To do this, we need to both provide the script, and tell the plug-in it's there.

   - In the manifest.lrweb page, add this command:

     ```
     AddResources{
         source="resources",
         destination="resources",
     }
     ```

   - Create a copy of the file `live_update.js`, which is part of the Lightroom SDK, and place it in the `resources` subfolder of the plug-in. This is a sample implementation of the update functions and callbacks needed for Live Update.

Testing the plug-in

1. Save all of your changes and restart Lightroom.
2. Select some photos.
3. Go to the Web Gallery page and select the new gallery type.
4. Place the cursor over the "MySample" text that appears as the default title; you should be able to edit it.
5. Look in the Site Info section of the control panel, and try editing the title text from there.

Adding a customized tagset

A tagset for a web gallery is an external file containing macro-like definitions that can be loaded by your web pages; see "Web SDK tagsets" on page 138. We will create a tagset for this gallery that allows us to build up a complex string. Our tags will build up a complex string by combining random members of a list of predefined elements with some set text and with the content of the tag when it is included in a page.

Define the tags

1. Create a file in the plug-in folder named `myExampleTags.lrweb`, and edit it to define this list of sayings:

```
local sayings = {
    "A dish fit for the gods - Julius Caesar, Shakespeare",
    "Oh, that way madness lies - King Lear, Shakespeare",
    "A multitude of sins - James 5:20",
    "A knight in shining armour - The Ancient Ballad of Prince Baldwin",
    "Blood is thicker than water - Guy Mannering; or the astrologer, Sir Walter
Scott"
}
```

2. Add a local counter variable to keep track of which member is chosen:

```
local randomSayingCount = 0
```

3. Define a function that selects one of the sayings. Make it a global variable that can be referenced from LuaPage templates:

```
globals = {
    randomSaying = function ()
        randomSayingCount = math.mod( randomSayingCount + 1, #sayings )
        return sayings[ randomSayingCount ]
    end,
}
```

4. Add the tag definitions:

```
tags = {
    saying = {
        startTag = "write( 'Here is a saying: ' ) write( randomSaying() )",
        endTag = "write( [[And that's all.]] ) ",
    },

    aQuote = {
        startTag = 'write( [[<blockquote style="  margin: 0 0 0 30px;
            padding: 10px 0 0 20px; font-size: 88%; line-height: 1.5em;
            color: #666;">]] )',
        endTag = 'write( [[</blockquote>]] )',
    }
}
```

This defines two dynamic tags with the names `saying` and `aQuote`. The tags can be referenced from a LuaPage template using the prefix with which the tagset is imported, and the tag name in an opening and closing tag:

```
<prefix:tagname>...</prefix:tagname>
```

The inner tag uses the global function we defined to construct some strings containing both static and dynamic text. These strings are output before and after the text content of the tag. The outer tag provides some style information for the text.

Add the tagset to the gallery

5. Edit the `manifest.lrweb` file to include the tags defined in the new tagset definition file. Add this line:

```
importTags( "xmpl", "myExampleTags.lrweb" )
```

This associates the prefix "xmpl" with the imported tagset, the tags can be referenced as:

```
<xmpl:aQuote>
    <xmpl:saying>...</xmpl:saying>
</xmpl:aQuote>
```

6. Finally, we need to use the tags in one of the template pages. Edit the file `large.html` to add this code just before the footer:

```
<xmpl:aQuote>You know what they say:<br>
    <xmpl:saying> <br />....how interesting!<br /></xmpl:saying>
</xmpl:aQuote>
```

7. Save the plug-in and reload it, as described in "Testing the plug-in" on page 205.

   At the bottom of the browser, you should now see the constructed text at the bottom, which changes each time the page is displayed:

You know what they say:
Here is a saying: Oh, that way madness lies - King Lear, Shakespeare
....how interesting!
And that's all.