

Chapter 8: Profiling Flex applications

As you interact with your application, identify performance bottlenecks and memory leaks in your applications by using the Adobe® Flex® profiler in Adobe Flash® Builder™. The profiler is only available for Flash Builder Premium.

About profiling

The Adobe Flex profiler helps you identify performance bottlenecks and memory leaks in your applications. You launch it from within Adobe Flash Builder, and as you interact with your application, the profiler records data about the state of the application, including the number of objects, the size of those objects, the number of method calls, and the time spent in those method calls.

Profiling an application can help you understand the following about your application:

Call frequency In some cases, you might discover that computationally expensive methods are called more than once when multiple calls are not necessary. By identifying the most commonly called methods, you can focus your performance-tuning time on a smaller area of the application, where it will have the most impact on performance.

Method duration The profiler can tell you how much time was spent in a particular method, or, if the method is called multiple times, what the average amount of time spent in that method was during a profiling section. If you discover that some methods cause a performance bottleneck, you can try to optimize those methods.

Call stacks By tracing the call stack of a method, you can see the entire path that the application takes as it calls successive methods. This might lead you to discover that methods are being called unnecessarily.

Number of instances (object allocation) You might discover that the same object is being created many times, when only a specific number of instances are required. In these cases, you might consider implementing a Singleton pattern if you really require only one of those objects, or applying other techniques that reduce excessive object allocation. If the number of objects is large, but necessary, you might consider optimizing the object itself to reduce its aggregate resource and memory usage.

Object size If you notice that some objects are disproportionately large, you can try to optimize those objects to reduce their memory footprint. This is especially helpful if you optimize objects that are created many times in the application.

Garbage collection When comparing profiling snapshots, you might discover that some objects that are no longer required by the application are still “loitering,” or are still stored in memory. To avoid these memory leaks, you add logic that removes any remaining references to those objects.

You should not look at profiling as only a single, discrete step in the process of developing an application. Rather, profiling should be an integral part of each step of application development. If possible, you should profile an application early and often during application development so that you can quickly identify problem areas. Profiling is an iterative process, and you gain the most benefit by profiling as often as possible.

About types of profiling

Before you use the profiler, you should decide what kind of profiling you are going to do: performance profiling or memory profiling.

Performance profiling is the process of looking for methods in your application that run slowly and can be improved. Once identified, these hot spots can be optimized to speed up execution times so that your application runs faster and responds more quickly to user interaction. You generally look for two things when doing performance profiling: a method that is called only once but takes more time to run than similar methods, or a method that may not take much time to run but is called many times. You use the performance profiling data to identify the methods that you then optimize. You might find that reducing the number of calls to a method is more effective than refactoring the code within the method.

Memory profiling is the process of examining how much memory each object or type of object is using in the application. You use the memory profiling data in several ways: to see if there are objects that are larger than necessary, to see if there are too many objects of a single type, and to identify objects that are not garbage collected (memory leaks). By using the memory profiling data, you can try to reduce the size of objects, reduce the number of objects that are created, or allow objects to be garbage collected by removing references to them.

Memory profiling can slow performance of your application because it uses much more memory than performance profiling. You should only do memory profiling when necessary.

You often do both performance profiling and memory profiling to locate the source of performance problems. You use performance profiling to identify the methods that result in excessive memory allocation and long execution times. Then, you use memory profiling to identify the memory leaks in those methods.

When you know what kind of profiling you are going to do, you can start the profiler.

Additional resources

The profiler alone does not improve the size, speed, and perceived performance of your application. After you use the profiler to identify the problem methods and classes, look at the following resources in the Flex documentation for help in improving your application:

- Optimizing Flex Applications in *Building and Deploying Adobe Flex3 Applications*
- Improving Startup Performance in *Building and Deploying Adobe Flex3 Applications*

How the Flex profiler works

The profiler is an agent that communicates with the application that is running in Flash Player. It connects to your application with a local socket connection. As a result, you might have to disable anti-virus software to use it if your antivirus software prevents socket communication.

When the profiler is running, it takes a snapshot of data at very short intervals, and records what Adobe Flash Player is doing at the time. This is called *sampling*. For example, if your application is executing a method at the time of the snapshot, the profiler records the method. If, by the next snapshot, the application is still executing that same method, the profiler continues to record the time. When the profiler takes the next snapshot, and the application has moved on to the next operation, the profiler can report the amount of time it took for the method to execute.

Sampling lets you profile without noticeably slowing down the application. The interval is called the *sampling rate*, and it occurs every 1 ms or so during the profiling period. This means that not every operation is recorded and that not every snapshot is accurate to fractions of a millisecond. But it does give you a much clearer idea of what operations take longer than others.

By parsing the data from sampling, the profiler can show every operation in your application, and the profiler records the execution time of those operations. The profiler also records memory usage and stack traces and displays the data in a series of views, or panels. Method calls are organized by execution time and number of calls, as well as number of objects created in the method.

The profiler also computes cumulative values of data for you. For example, if you are viewing method statistics, the cumulative data includes the time and memory allocated during that method, plus the time and memory allocated during all methods that were called from that method. You can drill down into subsequent method calls until you find the source of performance problems.

About the profiling APIs

The profiler uses the ActionScript APIs defined in the `flash.sampler.*` package. This package includes the `Sample`, `StackFrame`, `NewObjectSample`, and `DeleteObjectSample` classes. You can use the methods and classes in this package to write your own profiler application or to include a subset of profiling functionality in your applications.

In addition to the classes in the `flash.sampler.*` package, the profiler also uses methods of the `System` class.

About internal player actions

Typically, the profiler records data about methods of a particular class that the Flash Player was executing during the sampling snapshot. However, sometimes the profiler also records internal player actions. These actions are denoted with brackets and include `[keyboardEvent]`, `[mark]`, and `[sweep]`.

For example, if `[keyboardEvent]` is in the method list with a value of 100, you know that the player was doing some internal action related to that event at least 100 times during your interaction period.

The following table describes the internal Flash Player actions that appear in profiling data:

Action	Description
<code>[generate]</code>	The just-in-time (JIT) compiler generates AS3 machine code.
<code>[mark]</code>	Flash Player marks live objects for garbage collection.
<code>[newclass]</code>	Flash Player is defining a class. Usually, this occurs at startup but a new class can be loaded at any time.
<code>[pre-render]</code>	Flash Player prepares to render objects (including the geometry calculations and display list traversal that happens before rendering).
<code>[reap]</code>	Flash Player reclaims DRC (deferred reference counting) objects.
<code>[render]</code>	Flash Player renders objects in the display list (pixel by pixel).
<code>[sweep]</code>	Flash Player reclaims memory of unmarked objects.
<code>[verify]</code>	The JIT compiler performs ActionScript 3.0 bytecode verification.
<code>[event_typeEvent]</code>	Flash Player dispatches the specified event.

You can use this information to help you identify performance issues. For example, if you see a large number of entries for `[mark]` and `[sweep]`, you can assume that there are a large number of objects being created and then marked for garbage collection. By comparing these numbers across different performance profiles, you can see whether changes that you make have any effect.

To view data about these internal actions, you view a performance profile in the Performance Profile view or a memory profile in the Allocation Trace view. For more information, see [“Using the Performance Profile view”](#) on page 165 and [“Using the Allocation Trace view”](#) on page 162.

Using the profiler

The profiler requires Flash Player version 9.0.124 or later. You can profile applications that were compiled for Flex 2, Flex 2.0.1, and Flex 3. You can use the profiler to profile ActionScript 3.0 applications that were built with Flash Authoring, as well as desktop applications that run on Adobe® AIR®.

The profiler requires debugging information in the application that you are profiling. When you compile an application and launch the profiler, Flash Builder includes the debugging information in the application by default. You can explicitly include debugging information in an application by setting the `debug` compiler option to `true`. If you export an application by using the Export Release Build option, the application does not contain debugging information in it.

Starting, stopping, and resuming the profiler

You can profile applications that you are currently developing in Flash Builder. Flash Builder includes debugging information when it compiles and runs an application during a profiling session. You can also profile external applications that you are not currently developing in Flash Builder but whose SWF file is available with a URL or on the file system. To profile an application, the application's SWF file must include the debugging information. For more information, see [“Profiling external applications”](#) on page 155.

Start profiling a Flex application

- 1 Close all instances of your browser.
- 2 Open your application in Flash Builder.
- 3 Click the Profile *application_name* button in the main toolbar. Flash Builder informs you that you should close all instances of your browsers if you have not already done so.
- 4 Click the OK button. Flash Builder compiles the application and launches it in a separate browser window. Flash Builder also displays the Configure Profiler dialog box.
- 5 Select the options in the Configure Profiler dialog box and click Resume. To profile an application, you must select the Enable Memory Profiling option or the Enable Performance Profiling option. You can select both options when profiling an application.

The following table describes the options:

Setting	Description
Connected From	Shows you the server that you are launching the application from. If the application is running on the same computer as the profiler, this value is localhost. You cannot change this value. However, you can profile an application that is running on a separate computer. See “Profiling external applications” on page 155.
Application	Shows you which application you are about to profile. You cannot change this value.
Enable Memory Profiling	Instructs the profiler to collect memory data. Use this option to detect memory leaks or find excessive object creation. If you are doing performance profiling, you can deselect this option. By default, Enable Memory Profiling is selected.

Setting	Description
Watch Live Memory Data	Instructs the profiler to display memory data in the Live Objects view while profiling. This is not required for doing either memory or performance profiling. You can select this option only if you selected Enable Memory Profiling. By default, Watch Live Memory Data is selected.
Generate Object Allocation Stack Traces	Instructs the profiler to capture a stack trace each time a new object is created. Enabling this option slows down the profiling experience. Typically, you select this option only when necessary. This option is only available when you select Enable Memory Profiling. By default, Generate Object Allocation Stack Traces is not selected. If you do not select this option, you cannot view allocation trace information in the Object References view or in the Allocation Trace view.
Enable Performance Profiling	Instructs the profiler to collect stack trace data at the sampling intervals. Use these samples to determine where your application spends the bulk of the execution time. If you are doing memory profiling, you can deselect this option. By default, Enable Performance Profiling is selected.

You can change the default values of these options by changing the profiling preferences. For more information, see [“Setting profiler preferences”](#) on page 155.

- 6 You can now start interacting with your application and examining the profiler data.

Pause and resume profiling a Flex application




After you have started the profiler, you can pause and restart applications in the Profile view. You select an application and then select the action you want to perform on that application. The following example shows you the Profile view with multiple applications. One application is currently running while other applications have been terminated.






Stop profiling a Flex application



- 1 Select the application in the Profile view.
- 2 Click the Terminate button to end the profiling session. This does not close the browser or kill the Player process. You must do that manually.
- 3 To return to the Flex Development perspective, select Flex Development from the perspective drop-down list. You can also change perspectives by selecting Control+F8 on Windows.

About the profiler buttons







The following table describes the buttons in the profiler toolbar:

Button	Name	Description
	Resume	Resumes the profiling session. This option is enabled only when an application name is selected and is currently suspended.
	Suspend	Suspends the profiling session. This option is enabled only when an application name is selected and is currently running.
	Terminate	Terminates the profiling session. This option is enabled only when an application name is selected and it has not been terminated already.

Button	Name	Description
	Run Garbage Collector	<p>Instructs Flash Player to run garbage collection. This option is enabled only when an application name is selected and the application is currently running.</p> <p>For more information about garbage collection, see “About garbage collection” on page 171.</p>
	Take Memory Snapshot	<p>Stores the memory usage of an application so that you can examine it or compare it to other snapshots.</p> <p>This option is enabled only when an application name is selected and that application is currently running and when you select Enable Memory Profiling in the launch dialog box. The profiler adds new memory snapshots as children of the selected application in the Profile view.</p> <p>To open the new memory snapshot in the Memory Snapshot view, double-click the memory snapshot entry.</p> <p>Garbage collection occurs implicitly before memory snapshots are recorded. In other words, clicking the Take Memory Snapshot button is the equivalent of clicking the Run Garbage Collection button and then clicking the Take Memory Snapshot button.</p> <p>For more information about memory snapshots, see “Using the Memory Snapshot view” on page 159.</p>
	Find Loitering Objects	<p>Compares two memory snapshots in the Loitering Objects view.</p> <p>This option is enabled only when two memory snapshots are selected and when you selected Enable Memory Profiling in the launch dialog box.</p> <p>For more information about the Loitering Objects view, see “Using the Loitering Objects view” on page 169.</p>
	View Allocation Trace	<p>Compares the methods between two memory snapshots in the Allocation Trace view.</p> <p>This option is enabled only when two memory snapshots are selected, and when you select Enable Memory Profiling in the launch dialog box.</p> <p>For more information about the Allocation Trace view, see “Using the Allocation Trace view” on page 162.</p>
	Reset Performance Data	<p>Clears the performance profiling data.</p> <p>This option is enabled only when an application name is selected and the application is running and when you select Enable Performance Profiling in the launch dialog box.</p> <p>You typically click this button, interact with your application and then click the Capture Performance Profile button to get a performance snapshot of the application from the time you reset the data.</p> <p>For more information about the Performance Profile view, see “Using the Performance Profile view” on page 165.</p>

Button	Name	Description
	Capture Performance Profile	Stores a new performance snapshot as a child of the selected application. This option is enabled only when an application name is selected and the application is running and when you select Enable Performance Profiling in the launch dialog box. To open the Performance Profile view, double-click the performance snapshot entry. For more information about the Performance Profile view, see “Using the Performance Profile view” on page 165.
	Delete	Removes the selected snapshot’s data from memory. Clicking this button also removes the application from the profile view, if the application has been terminated. This option is enabled only when a performance or memory snapshot is selected.
n/a	Save	Saves profiling data to disk. The Save option is available from the Profiler view menu. This option is enabled only when an application name is selected.

Some views in the profiler, such as Method Statistics and Object Statistics, have navigation buttons that you use to traverse the stack or change the view. The following table describes the navigation buttons in these profiler views:

Button	Name	Description
	Back	Shows all the methods that you traversed from the first selected method to the currently displaying method.
	Forward	Shows the currently displayed method and the methods that lead to the currently selected method. This item is enabled after you move backward.
	Home	Displays the first selected method.
	Open Source File	Opens a source editor that shows the source code of the selected method.
	Filters	Lets you control which methods you want to include in the table. For more information, see “About profiler filters” on page 175.
	Show/Hide Zero Time Methods	Shows or hides methods that have a time of 0.00 in the average time column, which is a result of not showing up in any samples.

Saving and loading profiling data

After you run the profiler, you can save the data so that you can compare a snapshot from the current profiling session with a snapshot you take after you make changes to your code. This helps you determine if you identified the right problem areas and if your changes are improving the performance and memory usage of your application.

When you save profiling data, you save all the application data in that profile. This includes all performance profiles, memory snapshots, and allocation traces. Flash Builder writes this information to a group of binary files in the location that you specify.

Save profiling data

- 1 Select the application in the Profile view.

- 2 Open the drop-down list in the Profile view and select Save. The Browser for Folder dialog box appears.
- 3 Choose a location to save the profile data and click OK. You should create a new folder for each set of profiling data that you want to save. Otherwise, the new data will overwrite the old data if you choose the same folder.

Retrieve saved profiling data

- 1 Select the Saved Profiling Data view.
- 2 Click the Open button. The Browser for Folder dialog box appears.
- 3 Navigate to the folder that contains your application's profile data and click OK. Flash Builder displays the available profiling data in the Saved Profiling Data view. You cannot resume the application in this view, but you can view the memory snapshots, performance profile, or other data that you saved.

You cannot delete saved application data from within Flash Builder.

Delete profiling data

- 1 Select the snapshot from the application in the Profile view.
- 2 Click the Delete button.

Setting profiler preferences

You can set some profiler preferences so that your settings are applied to all profiling sessions. You can use these settings to define the Flash Player/browser that you use to profile the application in, as well as define the default filters and the port number that the application is available on, if the profiled application is running on a server.

Set Flash Builder preferences for multiple profiling sessions

- ❖ Open the Preferences dialog and select Flash Builder > Profiler.

Select the options under the Profiler menu to navigate to the various options. The following table describes the preferences you can set:

Menu Selection	Description
Profiler	Lets you select the default profiling method. Select the options to enable or disable memory profiling and performance profiling.
Connections	Lets you define the port number that Flash Builder listens to the profiled application on. The default port number is 9999. You cannot change the port to 7935, because that port is used by the debugger.
Exclusion Filters	Lets you define the default packages that are excluded from the profiler views. For more information on using filters, see "About profiler filters" on page 175.
Inclusion Filters	Lets you defines the default packages that are included in the profiler views. All other packages are excluded. For more information on using filters, see "About profiler filters" on page 175.
Player/Browser	Lets you define the location of the Flash Player executable and browser executable that Flash Builder uses to run your profiled application.

Profiling external applications

In addition to profiling applications that you are developing in Flash Builder, you can profile external applications. External applications can be SWF files located anywhere that is accessible. This includes applications that are located on a remote web server or an application that is on your local file system.

For the SWF file, you can specify either a URL or a file system location. If you specify a URL, Flash Builder launches the application's SWF file within the default browser. The browser must be using the debugger version of Flash Player to successfully profile the application.

If you specify a file system location for the SWF file, Flash Builder opens the application within the debugger version of the stand-alone Flash Player. In general, you should request the file by using a URL. Running applications in the stand-alone version of Flash Player can produce unexpected results, especially if your application uses remote services or network calls.

Profile an external application

- 1 Change to the Flex Profiling perspective.
- 2 Select Profile > Profile External Application. The Profile External Application dialog box appears.
- 3 Select the Launch the Selected Application option (the default) and click the New button. The Add an Application dialog box appears.

You can also manually launch the application by selecting the Launch the Application Manually Outside Flash Builder option.

- 4 Enter the location of the SWF file and click OK, or click the Browse button and locate your application on your file system.
- 5 Click the Launch button. If you specified a URL for the location of the application, Flash Builder launches the application within the default browser. If you specified a file system location for the application, Flash Builder opens the application in the stand-alone version of Flash Player.

If you specified a SWF file that was not compiled with debugging information, Flash Builder returns an error. Recompile the application with the `debug compiler` option set to `true` and launch it again.

About the profiler views

The Flex profiler is made up of several views (or panels) that present profiling data in different ways. The following table describes each of these views:

View	Description
Profile	Displays the currently connected applications, their status, and all the memory and performance snapshots that are associated with them. Initially, profiling sessions start with no recorded performance or memory snapshots.
Saved Profiling Data	Displays a list of saved snapshots, organized by application. You can load saved profiling data by double-clicking the saved snapshot in this list. For more information, see "Saving and loading profiling data" on page 154.
Live Objects	Displays information about the classes used by the current application. This view shows which classes are instantiated, how many were created, how many are in the heap, and how much memory the active objects are taking up. For more information, see "Viewing information in the Live Objects view" on page 158.

View	Description
Memory Snapshot	<p>Displays the state of the application at a single moment in time. Contrast this with the Live Objects view, which is updated continuously. The Memory Snapshot view shows how many objects were referenced and used in the application and how much memory each type of objects used at that time.</p> <p>You typically compare two memory snapshots taken at different times to determine the memory leaks that exist between the two points in time.</p> <p>You view the Memory Snapshot view by clicking the Take Memory Snapshot button and then double-clicking the memory snapshot in the Profile view.</p> <p>For more information, see “Using the Memory Snapshot view” on page 159.</p>
Loitering Objects	<p>Displays the objects that were created between two memory snapshots and still exist in memory or were not garbage collected. You can double-click a class name in the table to open the Object References view. This lets you examine the relationship between the selected objects and the other objects.</p> <p>You view the Loitering Objects view by selecting two memory snapshots and clicking the Loitering Objects button.</p> <p>For more information, see “Using the Loitering Objects view” on page 169.</p>
Allocation Trace	<p>Displays method statistics when comparing two memory snapshots.</p> <p>You view the Allocation Trace view by selecting two memory snapshots and then clicking the View Allocation Trace button.</p> <p>For more information, see “Using the Allocation Trace view” on page 162.</p>
Object References	<p>Displays objects and the objects that reference them.</p> <p>You view the Object References view by double-clicking a class name in the Memory Snapshot or Loitering Objects views.</p> <p>For more information, see “Using the Object References view” on page 160.</p>
Object Statistics	<p>Displays details about the caller and callee of the selected group of objects.</p> <p>You view the Object Statistics view by double-clicking an entry in the Allocation Trace view.</p> <p>For more information, see “Using the Object Statistics view” on page 164.</p>
Performance Profile	<p>Displays how the methods in the application performed during a given time interval. You then click a method name in the table to open the Method Statistics view, which lets you locate performance bottlenecks.</p> <p>You view the Performance Profile view by double-clicking one of the performance snapshots in the Profile view.</p> <p>For more information, see “Using the Performance Profile view” on page 165.</p>
Method Statistics	<p>Displays the performance statistics of the selected group of methods.</p> <p>You view the Method Statistics view by double-clicking a row in the Performance Profile view or selecting a method in the Performance Profile and clicking the Open Method Statistics button.</p> <p>For more information, see “Identifying method performance characteristics” on page 167.</p>
Memory Usage	<p>Graphically displays peak memory usage and current memory usage over time.</p> <p>For more information, see “Using the Memory Usage graph” on page 170.</p>

Viewing information in the Live Objects view

The Live Objects view displays information about the classes that the current application uses. This view shows which classes are instantiated, how many were created, how many are in memory, and how much memory the active objects are taking up.

The profiler updates the data in the Live Objects view continually while you profile the application. You do not have to refresh the view or keep focus on it to update the data.

To use the Live Objects view, you must enable memory profiling when you start the profiler. This is the default setting. If you close the Live Objects view and want to reopen it, open the drop-down list in the Profile view and select Watch Live Objects.

The following example shows the Live Objects view:

Class	Package (Filtered)	Cumulative Instances	Instances	Cumulative Memory
Employee	valueObjects	1006 (32.3%)	1000 (48.57%)	244000 (78.38%)
_EmployeeEntityMetadata	valueObjects	1006 (32.3%)	1000 (48.57%)	60000 (19.27%)
DataMgtCF		1 (0.03%)	1 (0.05%)	1476 (0.47%)
MethodQueueElement	UIComponent.as\$168	62 (1.99%)	0 (0.0%)	848 (0.27%)
_DataMgtCF_mx_managers_SystemManager		1 (0.03%)	1 (0.05%)	728 (0.23%)
_mx_managers_CursorManagerStyle__embed_css_Assets_swf_r		1 (0.03%)	1 (0.05%)	424 (0.14%)
_e5e0a8b749c58154626094e667bd3c593a478f9fa8720b4c6f5		1 (0.03%)	1 (0.05%)	404 (0.13%)
_5e891232bd6d81b9d6974dfdba52e58bf92fe717751ed14a87b		1 (0.03%)	1 (0.05%)	404 (0.13%)
_12bb6e99c417300f479d05e018f8dc231181d5e4fbd4451087bc		1 (0.03%)	1 (0.05%)	404 (0.13%)
_0dc790747641a559f6e86f8aa121d894dc3470a66614c311433		1 (0.03%)	1 (0.05%)	404 (0.13%)
ListCollectionViewCursor	ListCollectionView.as\$85	22 (0.71%)	2 (0.1%)	304 (0.1%)
LineSegment	Path.as\$138	12 (0.39%)	12 (0.58%)	288 (0.09%)
RPCDataServiceRequest	RPCDataServiceAdapter.as\$141	1 (0.03%)	1 (0.05%)	260 (0.08%)
Vector.<*>	__AS3__vec	5 (0.16%)	5 (0.24%)	220 (0.07%)
Vector.<int>	__AS3__vec	8 (0.26%)	4 (0.19%)	160 (0.05%)
Vector.<Number>	AS3 __vec	8 (0.26%)	4 (0.19%)	160 (0.05%)

The following table describes the columns in the Live Objects view:

Column	Description
Class	The classes that have instances in the currently running application.
Package	<p>The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class.</p> <p>If the Package field is empty, the class is in the global package or the unnamed package.</p>
Cumulative Instances	The total number of instances of each class that have been created since the application started.
Instances	The number of instances of each class that are currently in memory. This value is always smaller than or equal to the value in the Cumulative Instances column.
Cumulative Memory	The total amount of memory, in bytes, that all instances of each class used, including classes that are no longer in memory.
Memory	The total amount of memory, in bytes, that all instances of each class currently use. This value is always smaller than or equal to the value in the Cumulative Memory column.

You typically use the data in the Live Objects view to see how much memory is being used by objects. As objects are garbage collected, the number of instances and memory use decrease, but the cumulative instances and cumulative memory use increase. This view also tells you how memory is used while the application is running.

For more information on running and analyzing the results of garbage collection, see [“About garbage collection”](#) on page 171.

You limit the data in the Live Objects view by using the profiler filters. For more information, see [“About profiler filters”](#) on page 175.

Using the Memory Snapshot view

The Memory Snapshot view displays information about the application’s objects and memory usage at a particular time. Unlike the Live Objects view, the data in the Memory Snapshot view is not continually updated.

To use the Memory Snapshot view, you must enable memory profiling when you start the profiler. This is the default setting.

Create and view a memory snapshot

- 1 Start a profiling session.
- 2 Interact with your application until you reach a point in the application’s state where you want to take a memory snapshot.
- 3 Select the application in the Profile view.
- 4 Click the Take Memory Snapshot button.

The profiler creates a memory snapshot and marks it with a timestamp. The profiler also implicitly triggers garbage collection before the memory snapshot is recorded.

- 5 To view the data in the memory snapshot, double-click the memory snapshot in the Profile view.

The following example shows the Memory Snapshot view:

Class	Package (Filtered)	Instances	Memory
Employee	valueObjects	1000 (48.85%)	244000 (78.76%)
_EmployeeEntityMetadata	valueObjects	1000 (48.85%)	60000 (19.37%)
DataMgtCF		1 (0.05%)	1476 (0.48%)
_DataMgtCF_mx_managers_SystemManager		1 (0.05%)	728 (0.23%)
_mx_managers_CursorManagerStyle__embed_css_Assets_swf_mx_skins_cursor_BusyCurs		1 (0.05%)	424 (0.14%)
_e5e0a8b749c58154626094e667bd3c593a478f9fa8720b4c6f5a9dbdec0d6f39_flash_displ		1 (0.05%)	404 (0.13%)
_5e891232bd6d81b9d6974dfdba52e58bf92fe717751ed14a87b1b6a6e8ec76bc_flash_disp		1 (0.05%)	404 (0.13%)
_12bb6e99c417300f479d05e018fdc231181d5e4fbd4451087bd39336fac89240_flash_disp		1 (0.05%)	404 (0.13%)
_0dc790747641a559f6e86f8aa121d894dc3470a66614c3114331e5d3a910bc47_flash_disp		1 (0.05%)	404 (0.13%)
RPCDataServiceRequest	RPCDataServiceAdapter.as\$141	1 (0.05%)	260 (0.08%)
Vector.<*>	__AS3__vec	5 (0.24%)	220 (0.07%)
ListCollectionViewCursor	ListCollectionView.as\$85	2 (0.1%)	152 (0.05%)
EmployeeService	services.employeeservice	1 (0.05%)	144 (0.05%)
LineSegment	Path.as\$138	6 (0.29%)	144 (0.05%)
Vector.<int>	__AS3__vec	2 (0.1%)	80 (0.03%)
Vector.<Number>	__AS3__vec	2 (0.1%)	80 (0.03%)

The following table describes the columns in the Memory Snapshot view:

Column	Description
Class	The classes that had instances in memory at the time that you recorded the memory snapshot.

Column	Description
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Instances	The number of instances in memory of each class at the time that you recorded the memory snapshot.
Memory	The amount of memory, in bytes, that all instances of each class used at the time that you recorded the memory snapshot.

You typically use a memory snapshot as a starting point to determine which classes you should focus on for memory optimizations or to find memory leaks. You do this by creating multiple memory snapshots at different points in time and then comparing the differences in the Loitering Objects or Allocation Trace views.

You can save memory snapshots to compare an application's state during a different profiling session. For more information, see [“Saving and loading profiling data”](#) on page 154.

When you double-click a row in the Memory Snapshot view, the profiler displays the Object References view. This view displays the stack traces for the current class's instances. You view the stack traces for the current class's instances in the Object References view. For more information about the Object References view, see [“Using the Object References view”](#) on page 160.

You can also specify which data to display in the Memory Snapshot view by using profiler filters. For more information, see [“About profiler filters”](#) on page 175.

Using the Object References view

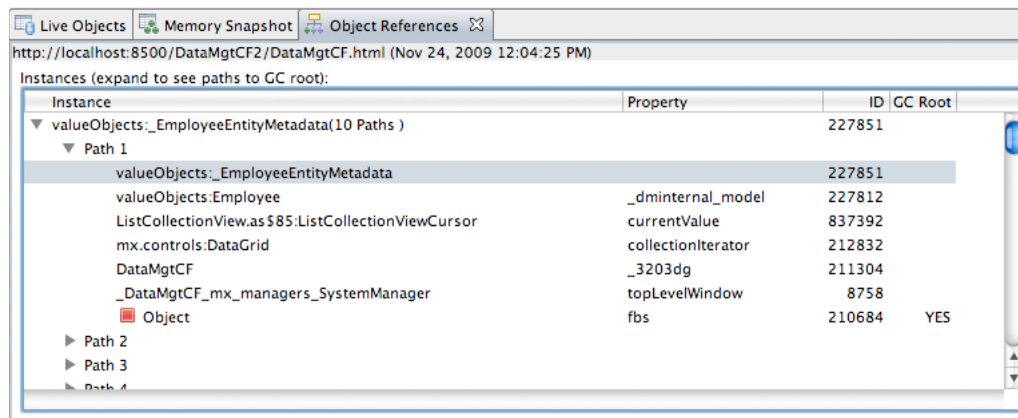
The Object References view displays stack traces for classes that were instantiated in the application.

To open the Object References view, double-click a class name in the Memory Snapshot or Loitering Objects views. The Object References view displays information about the selected class's instances.

The Object References view displays data in two tables: the Instances table and the Allocation Trace table.

The Instances table lists all objects that hold references to the current object. The number in parentheses after the class name is the total number of references to the current object. You cannot view the number of forward references for an object. If no objects hold references to the specified object, then there will be no objects listed in this table. This would not normally happen because that object should have been garbage collected if it had no references.

The following example shows the Instances table in the Object References view:



The following table describes the columns in the Instances table:

Column	Description
Instance	The class of the object that holds a reference to the specified object. Expand an instance of a class to view the paths to the object. The number of paths displayed is configurable from the Filters option in the Memory snapshot view.
Property	The property of the object that holds a reference to the specified object. For example, if you have object <code>o</code> with a property <code>i</code> , and assign that property to point to your button's label: <pre>o.i = myButton.label;</pre> That creates a reference to <code>myButton.label</code> from property <code>i</code> .
ID	The reference ID of the object that holds the reference to the selected object.
GC Root	Indicates whether an object has a back-reference to GC Root. Expand a path to an instance of an object to view whether there is a back-reference to GC Root.

The Allocation Trace table shows the stack trace for the selected instance in the Instances table. When you select an instance in the Instances table, the profiler displays the call stack for that instance in the Allocation Trace table.

The following example shows the Allocation Trace table in the Object References view:

Method	Location	Line
[-] BasicChartEvent:_BasicChartEvent_PlotSeries2_c()	BasicChartEvent.mxml	41
<anonymous>()	BasicChartEvent.mxml	
Function:http://adobe.com/AS3/2006/builtin:call()		
mx.core:ComponentDescriptor:getProperties()	ComponentDescriptor.as	239
mx.core:Container:createComponentFromDescriptor()	Container.as	3592
mx.core:Container:createComponentsFromDescriptors()	Container.as	3528
mx.containers:Panel:createComponentsFromDescriptors()	Panel.as	1510
mx.core:Container:createChildren()	Container.as	2630
mx.containers:Panel:createChildren()	Panel.as	1050
mx.core:UIComponent:initialize()	UIComponent.as	5172
mx.core:Container:initialize()	Container.as	2567
mx.core:UIComponent:http://www.adobe.com/2006/flex/mx/internal:child	UIComponent.as	5069
mx.core:Container:http://www.adobe.com/2006/flex/mx/internal:childAdd	Container.as	3340
mx.core:Container:addChildAt()	Container.as	2258
mx.core:Container:addChild()	Container.as	2188
mx.core:Container:createComponentFromDescriptor()	Container.as	3716
mx.core:Container:createComponentsFromDescriptors()	Container.as	3528
mx.core:Container:createChildren()	Container.as	2630
mx.core:UIComponent:initialize()	UIComponent.as	5172
mx.core:Container:initialize()	Container.as	2567
mx.core:Application:initialize()	Application.as	841
BasicChartEvent:initialize()	BasicChartEvent.mxml	
mx.managers:SystemManager:http://www.adobe.com/2006/flex/mx/inter	SystemManager.as	1634
mx.managers:SystemManager:initializeTopLevelWindow()	SystemManager.as	2505
mx.managers:SystemManager:docFrameHandler()	SystemManager.as	2356
[execute-queued]()		

The following table describes the columns in the Allocation Trace table:

Column	Description
Method	The top-level method in this table is the method that created the instance of the class that is listed in the Instances table. You can expand the method to show the stack trace of the method. This can help you determine where the call stack began.
Location	The file where the method is defined.
Line	The line number in the file.

You can only view data in this table when you enable allocation traces when you start the profiler.

You can open the source code of the selected class by double-clicking a class in this table.

Using the Allocation Trace view

The Allocation Trace view shows which methods were called between two memory snapshots and how much memory was consumed during those method calls. To open the Allocation Trace view, you select two memory snapshots, and then click the View Allocation Trace button. For information on recording a memory snapshot, see [“Using the Memory Snapshot view”](#) on page 159.

The result of the memory snapshot comparison is a list of methods that Flash Player executed between the two memory snapshots. For each of these methods, the profiler reports the number of objects created in that method.

You can use this information to optimize performance. After you identify methods that create an excessive number of objects, you can optimize those hot spots.

To use the Allocation Trace view, you must enable allocation traces when you start the profiler. The default is disabled.

The following example shows the Allocation Trace view:

Method	Package (Filtered)	Cumulative Instances	Self Instances	Cumulative Memory	Self Memory
[newclass]		49 (1.85%)	44 (7.86%)	65244 (19.5%)	65028 (84.44%)
global.flash.utils.describeType		478 (18.01%)	478 (85.36%)	5844 (1.75%)	5844 (7.59%)
_DataMgtCF_mx_managers.SystemManager.create		9 (0.34%)	3 (0.54%)	3868 (1.16%)	3708 (4.81%)
global.flash.utils.setInterval		21 (0.79%)	7 (1.25%)	1596 (0.48%)	1148 (1.49%)
global.flash.utils.getQualifiedClassName		7 (0.26%)	7 (1.25%)	422 (0.13%)	422 (0.55%)
DataMgtCF.button1_clickHandler		108 (4.07%)	3 (0.54%)	20989 (6.27%)	248 (0.32%)
SetIntervalTimer.SetIntervalTimer		14 (0.53%)	7 (1.25%)	448 (0.13%)	224 (0.29%)
_Super_Employee.valueObjects:_Super_Employee	valueObjects	5 (0.19%)	5 (0.89%)	204 (0.06%)	204 (0.26%)
[enterFrameEvent]		247 (9.31%)	3 (0.54%)	15631 (4.67%)	96 (0.12%)
[mouseEvent]		1032 (38.88%)	2 (0.36%)	79793 (23.85%)	64 (0.08%)
<anonymous>		7 (0.26%)	1 (0.18%)	21265 (6.36%)	28 (0.04%)
_Super_Employee.set last_name	valueObjects	76 (2.86%)	0 (0.0%)	4021 (1.2%)	0 (0.0%)
_Super_Employee.get _model	valueObjects	4 (0.15%)	0 (0.0%)	1007 (0.3%)	0 (0.0%)
Employee.valueObjects.Employee	valueObjects	5 (0.19%)	0 (0.0%)	204 (0.06%)	0 (0.0%)
global\$init.global\$init		49 (1.85%)	0 (0.0%)	65244 (19.5%)	0 (0.0%)
[textEvent]		4 (0.15%)	0 (0.0%)	144 (0.04%)	0 (0.0%)
[keyboardEvent]		4 (0.15%)	0 (0.0%)	2792 (0.83%)	0 (0.0%)

The following table describes the columns in the Allocation Trace view:

Column	Description
Method	The method that was called during the snapshot interval. This column also contains the class whose instance called this method.
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Cumulative Instances	The number of objects instantiated in this method and all methods called from this method.
Self Instances	The number of objects instantiated in this method. This does not include objects that were instantiated in subsequent method calls from this method.
Cumulative Memory	The amount of memory, in bytes, used by the objects instantiated in this method and all methods called from this method.
Self Memory	The amount of memory, in bytes, used by the objects instantiated in this method. This does not include the memory used by objects that were instantiated in subsequent method calls from this method.

When recording methods during sampling intervals, the profiler also records internal Flash Player actions. These actions show up in the method list in brackets and appear as `[mouseEvent]` or `[newclass]` or with similar names. For more information about internal Flash Player actions, see [“How the Flex profiler works”](#) on page 149.

To open the Object Statistics view, click a row in the Allocation Trace table. This view provides details about the objects that were created in the method that you selected. It also lets you drill down into the objects that were created in methods that were called from this method. For more information, see [“Using the Object Statistics view”](#) on page 164.

You limit the data in the Allocation Trace view by using the profiler filters. For more information, see [“About profiler filters”](#) on page 175.

Using the Object Statistics view

The Object Statistics view shows the performance statistics of the selected group of objects. This view helps you identify which methods call a disproportionate number of other methods. It also shows you how much memory the objects instantiated in those method calls consume. You use the Object Statistics view to identify potential memory leaks and other sources of performance problems in your application.

To access the Object Statistics view, you select two memory snapshots in the Profile view and view the comparison in the Allocation Trace view. Then you double-click a row to view the details in the Object Statistics view.

There are three sections in the view:

- A summary of the selected object's statistics, including the number of instances and amount of memory used.
- Self Instances table: A list of objects that were instantiated in the method that you selected in the Allocation Trace view. This does not include objects that were instantiated in subsequent method calls from this method. The number of objects in this list matches the number of objects specified in the Self Instances column in the Allocation Trace view.
- Callee Instances table: A list of objects that were instantiated in methods that were called by the method that you selected in the Allocation Trace view. This does not include objects that were instantiated in the method itself. The number of objects in this list matches the number of objects specified in the Cumulative Instances column in the Allocation Trace view.

The following example shows the method summary and the Self Instances and Callee Instances tables of the Object Statistics view:

The screenshot shows the Object Statistics view for the method [mouseEvent]. The summary statistics are:

- Method: [mouseEvent]
- Cumulative instances: 1032 (38.88%)
- Self instances: 2 (0.36%)
- Cumulative memory: 79793 (23.85%)
- Self memory: 64 (0.08%)

The Self instances table shows:

Class	Package	Self Instances	Self Memory
Event	flash.events	2 (100.0%)	64 (100.0%)

The Callee instances table shows:

Class	Package	Cumulative Instances	Cumulative Memory
Class		13 (1.26%)	34904 (43.78%)
Array		189 (18.35%)	8920 (11.19%)
DataGridItemRenderer	mx.controls.dataGridClasses	6 (0.58%)	7512 (9.42%)
String		279 (27.09%)	5433 (6.81%)
Rect	spark.primitives	8 (0.78%)	3648 (4.58%)

The following table describes the fields in the Self Instances table in the Object Statistics view:

Column	Description
Class	The classes that were instantiated only in the selected method. This does not include classes that were instantiated in subsequent calls from this method.

Column	Description
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Self Instances	The number of instances of this class that were created only in the selected method. This does not include instances that were created in subsequent calls from this method.
Self Memory	The amount of memory, in bytes, that is used by instances that were created only in the selected method. This does not include the memory used by instances that were created in subsequent calls from this method.

The following example shows the Callee Instances table of the Object Statistics view:

Callee instances:

Class	Package	Cumulative Instances	Cumulative Memory
Class		13 (1.26%)	34904 (43.78%)
Array		189 (18.35%)	8920 (11.19%)
DataGridItemRenderer	mx.controls.dataGridClasses	6 (0.58%)	7512 (9.42%)
String		279 (27.09%)	5433 (6.81%)
Rect	spark.primitives	8 (0.78%)	3648 (4.58%)
Object		153 (14.85%)	3292 (4.13%)
Point	flash.geom	130 (12.62%)	3120 (3.91%)
SpriteAsset	mx.core	4 (0.39%)	1952 (2.45%)
DataGridListData	mx.controls.dataGridClasses	48 (4.66%)	1536 (1.93%)
Function		41 (3.98%)	1312 (1.65%)
SolidColor	mx.graphics	8 (0.78%)	1056 (1.32%)
EdgeMetrics	mx.core	23 (2.23%)	920 (1.15%)
Dictionary	flash.utils	13 (1.26%)	752 (0.94%)
TextFormat	flash.text	6 (0.58%)	600 (0.75%)
ListEvent	mx.events	12 (1.17%)	576 (0.72%)
PropertyChangeEvent	mx.events	10 (0.97%)	520 (0.65%)
Matrix	flash.geom	9 (0.87%)	504 (0.63%)

The following table describes the fields in the Callee Instances table of the Object Statistics view:

Column	Description
Class	The classes that were instantiated in the selected method. This includes classes that were instantiated in subsequent calls from this method.
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Cumulative Instances	The number of instances of this class that were created in the selected method and in subsequent calls from this method.
Cumulative Memory	The amount of memory, in bytes, that is used by instances that were created in the selected method and in subsequent calls from this method.

Using the Performance Profile view

The Performance Profile view is the primary view to use when doing performance profiling. It shows statistics such as number of calls, self-time, and cumulative time for the methods that are called during a particular sampling interval. You use this data to identify performance bottlenecks.

The process of performance profiling stores a list of methods and information about those methods that were called between the time you clear the performance data and the time you capture new data. This time difference is known as the *interaction period*.

To use the Performance Profile view, you must enable performance profiling when you start the profiler. This is the default setting.

Generate a performance profile

- 1 Start a profiling session with performance profiling enabled.
- 2 Interact with your application until you reach the point where you want to start profiling.
- 3 Click the Reset Performance Data button.
- 4 Interact with your application and perform the actions to profile.
- 5 Click the Capture Performance Profile button.

The time difference between when you clicked Reset Performance Data and the time you clicked Capture Performance Profile is the interaction period. If you do not click the Reset Performance Data button at all, then the performance profile includes all data captured from the time the application first started.

- 6 Double-click the performance profile in the Profile view.

The following example shows the Performance Profile view:

Method	Package (Filtered)	Calls	Cumulative Time (ms)	Self Time (ms)	Avg. Cumulative Time (ms)	Avg. Self Time (ms)
[tincan]		1 (0.01%)	4203 (61.77%)	4203 (61.77%)	4203.0	4203.0
[mouseEvent]		1 (0.01%)	1387 (20.39%)	1384 (20.34%)	1387.0	1384.0
NetConnectionMessageResponder.resultHandler		0 (0.0%)	952 (13.99%)	0 (0.0%)	0.0	0.0
[reap]		1 (0.01%)	173 (2.54%)	173 (2.54%)	173.0	173.0
[enterFrameEvent]		1 (0.01%)	138 (2.03%)	1 (0.01%)	138.0	1.0
TypeUtility.convertResultHandler	com.adobe.serializers.utility	0 (0.0%)	129 (1.9%)	0 (0.0%)	0.0	0.0
TypeUtility.convertListToStrongType	com.adobe.serializers.utility	0 (0.0%)	128 (1.88%)	1 (0.01%)	0.0	0.0
[mark]		1 (0.01%)	108 (1.59%)	104 (1.53%)	108.0	104.0
TypeUtility.convertToStrongType	com.adobe.serializers.utility	0 (0.0%)	88 (1.29%)	7 (0.1%)	0.0	0.0
_Super_Employee.set uid	valueObjects	0 (0.0%)	56 (0.82%)	1 (0.01%)	0.0	0.0
[verify]		1 (0.01%)	46 (0.68%)	33 (0.49%)	46.0	33.0
TypeUtility.assignProperty	com.adobe.serializers.utility	0 (0.0%)	44 (0.65%)	0 (0.0%)	0.0	0.0
[generate]		1 (0.01%)	33 (0.49%)	33 (0.49%)	33.0	33.0
_Super_Employee.valueObjects._Super_Employee	valueObjects	0 (0.0%)	32 (0.47%)	5 (0.07%)	0.0	0.0

The following table describes the columns in the Performance Profile view:

Column	Description
Method	The name of the method and the class to which the method belongs. Internal actions executed by Flash Player appear as entries in brackets; for example, [mark] and [sweep]. You cannot change the behavior of these internal actions, but you can use the information about them to aid your profiling and optimization efforts. For more information on these actions, see "How the Flex profiler works" on page 149.
Package	The package that the class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Calls	The number of times the method was called during the interaction period. If one method is called a disproportionately large number of times compared to other methods, then you can look to optimizing that method so that the execution time is reduced.

Column	Description
Cumulative Time	The total amount of time, in milliseconds, that all calls to this method, and all calls to subsequent methods, took to execute during the interaction period.
Self Time	The amount of time, in milliseconds, that all calls to this method took to execute during the interaction period.
Avg. Cumulative Time	The average amount of time, in milliseconds, that all calls to this method, and calls to subsequent methods, took to execute during the interaction period.
Avg. Self Time	The average amount of time, in milliseconds, that this method took to execute during the profiling period.

If you double-click a method in the Performance Profile view, Flex displays information about that method in the Method Statistics view. This lets you drill down into the call stack of a particular method. For more information, see [“Identifying method performance characteristics”](#) on page 167.

You limit the data in the Performance Profile view by using the profiler filters. For more information, see [“About profiler filters”](#) on page 175.

You can save performance profiles for later use. For more information, see [“Saving and loading profiling data”](#) on page 154.

Identifying method performance characteristics

The Method Statistics view shows the performance characteristics of the selected method. You typically use the Method Statistics view to identify performance bottlenecks in your application. By viewing, for example, the execution times of a method, you can see which methods take a disproportionate amount of time to run. Then you can selectively optimize those methods.

For more information, see [“Using the Performance Profile view”](#) on page 165.

View method details in the Method Statistics view

- 1 Double-click a row in the Performance Profile view or select a method in the Performance Profile view.
- 2 Click the Open Method Statistics button.

There are three sections in the view:

- A summary of the selected method’s performance, including the number of calls, cumulative time, and self-time.
- Callers table: Details about the methods that called the selected method. In some situations, it is important to know if the selected method is being called excessively, how it is being used, and whether it is being used correctly.
- Calleees table: Details about the methods that were called by the selected method.

The following example shows the method summary and the Callers and Callees tables of the Method Statistics view:

Method	Package (Filtered)	Cumulative Time (ms)	Self Time (ms)

Method	Package (Filtered)	Cumulative Time (ms)	Self Time (ms)
DataMgtCF_...DataMgtCF_Button3_click		27 (1.22%)	0 (0.00%)
DataMgtCF_...DataMgtCF_Button1_click		19 (0.86%)	0 (0.00%)
DataMgtCF_...DataMgtCF_Button4_click		4 (0.18%)	0 (0.00%)

The following table describes the fields in the Callers table of the Method Statistics view:

Column	Description
Method	The methods that called the method that appears in the summary at the top of this view. If this list is empty, the target method was not called by any methods that are not filtered out.
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Cumulative Time	The amount of time, in milliseconds, that each calling method, and all subsequent methods, spent executing.
Self Time	The amount of time, in milliseconds, that each calling method spent executing. This does not include methods called by subsequent methods.

The following table describes the fields in the Callees table of the Method Statistics view:

Column	Description
Method	The methods that were called by the method that is shown in the summary at the top of this view. If this list is empty, then the target method was not called by any methods that are not filtered out.
Package	The package that each class is in. If the class is not in a package, then the value of this field is the file name that the class is in. The number following the dollar sign is a unique ID for that class. If the Package field is empty, the class is in the global package or the unnamed package.
Cumulative Time	The amount of time, in milliseconds, that each called method, and all subsequent methods, spent executing.
Self Time	The amount of time, in milliseconds, that each called method spent executing. This does not include methods called by subsequent methods.

You can navigate the call stack while you attempt to find the performance bottlenecks by clicking the methods in either the Callers or Callees tables. If you double-click a method in these tables, the profiler displays that method's summary at the top of the Method Statistics view and then shows the callers and callees for the newly selected method in the two tables.

Note: The cumulative time minus the self-time in this view will not always equal the cumulative time of the callers. That is because if you drill up to a caller, then the cumulative time will be the self-time of that caller plus all chains from which the original method was called, but not other callees.

You can also use the Back, Forward, and Home profiler buttons to navigate the call stack.

You can limit the data in the Method Statistics view by using the profiler filters. For more information, see “[About profiler filters](#)” on page 175.

Using the Loitering Objects view

The Loitering Objects view shows you the differences between two memory snapshots of the application that you are profiling. The differences that this view shows are the number of instances of objects in memory and the amount of memory that those objects use. This is useful in identifying memory leaks. The time between two memory snapshots is known as the *snapshot interval*.

To open the Loitering Objects view, select two memory snapshots and click the Loitering Objects button. For information on recording a memory snapshot, see “[Using the Memory Snapshot view](#)” on page 159.

The following example shows the Loitering Objects view:

Class	Package	Instances	Memory
Array		102 (23.67%)	4748 (4.41%)
String		72 (16.71%)	4932 (4.58%)
Object		62 (14.39%)	2208 (2.05%)
DataGridListData	mx.controls.dataGridClasses	54 (12.53%)	1728 (1.61%)
WeakMethodClosure	flash.events	26 (6.03%)	416 (0.39%)
Class		18 (4.18%)	84081 (78.15%)
Date		16 (3.71%)	512 (0.48%)
EmbeddedFont	mx.core	13 (3.02%)	312 (0.29%)
Function		9 (2.09%)	288 (0.27%)
_EmployeeEntityMetadata	valueObjects	7 (1.62%)	420 (0.39%)
Employee	valueObjects	7 (1.62%)	1288 (1.2%)
ListRowInfo	mx.controls.listClasses	6 (1.39%)	288 (0.27%)
Graphics	flash.display	5 (1.16%)	60 (0.06%)

The following table describes the columns in the Loitering Objects view:

Column	Description
Class	The classes that were created but not destroyed during the snapshot interval.

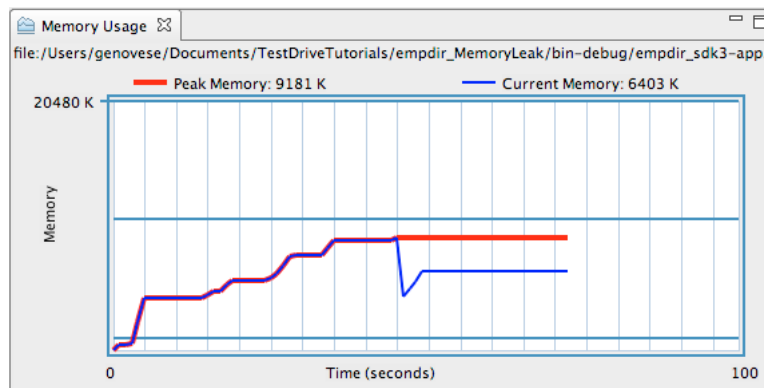
Column	Description
Package	<p>The package that each class is in. If the class is not in a package, then the value of this field is the filename that this class is in. The number following the dollar sign is a unique ID for that class.</p> <p>If the Package field is empty, the object is in the global package or the unnamed package.</p>
Instances	<p>The number of instances created during the snapshot interval. This is the difference between the number of instances of each class that existed in the first snapshot and the number of instances of each class in the second snapshot.</p> <p>For example, if there were 22,567 strings in the first snapshot, and 22,861 strings in the second snapshot, the value of this field would be 294.</p>
Memory	<p>The amount of memory allocated during the snapshot interval. This is the difference between the amount of memory that the instances of each class used at the time the first snapshot was taken and the amount of memory that the instances of each class used at the time the second snapshot was taken.</p> <p>For example, if Strings took up 2,031,053 bytes in the first snapshot and 2,029,899 bytes in the second snapshot, the value of this field would be 1154 bytes.</p>

For more information on identifying memory leaks, see [“Locating memory leaks”](#) on page 171.

Using the Memory Usage graph

The Memory Usage graph shows you the memory used by the application that you are profiling. This value is different from the memory usage of the Flash Player and its browser. That is because this value does not include the profiler agent or the browser’s memory usage. It consists only of the sum of the profiled application’s live objects. As a result, if you compare the value of memory usage in this graph against the amount of memory the browser uses as shown in, for example, the Windows Task Manager, you will get very different results.

The following image shows the graph in the Memory Usage view:



The value for Current Memory is the same as the sum of the totals in the Live Objects view’s Memory column, assuming that all filters are disabled.

The value for Peak Memory is the highest amount of memory that this application has used during the current profiling session.

The Memory Usage graph shows the application’s memory for the last 100 seconds. You cannot configure this number, and you cannot save historical data for the chart.

If you close the Memory Usage graph and want to reopen it, click the drop-down menu button in the Profile view and select Memory Usage.

About garbage collection

Garbage collection is the act of removing objects that are no longer needed from memory. Memory used by instances that no longer have any references to them should be deallocated during this process.

Flash Player performs garbage collection as necessary during an application's life cycle. Unreferencing an object does not trigger garbage collection. So, when you remove all references to an object, the garbage collector does not necessarily deallocate the memory for that object. That object becomes a candidate for garbage collection.

Clicking the Run Garbage Collector button does not guarantee that all objects that are eligible for garbage collection will be garbage collected. Garbage collection is typically triggered by the allocation of memory for new resources. When new resources require memory that is not available in the current allocation, the garbage collector runs and frees up memory that has been marked for deallocation. As a result, even if you remove all references to it, it might not be immediately garbage collected, but likely will be garbage collected when other instances are created and used. If an application is idle, you can watch its memory allocation. Even though there may be objects that are marked for collection, an idle application's memory usage typically does not change until you start interacting with it.

Flash Player allocates memory in blocks of many bytes, and not one byte at a time. If part of a block has been marked for garbage collection, but other parts of the block have not been marked, the block is not freed. The garbage collector attempts to combine unused portions of memory blocks into larger blocks that can be freed, but this is not guaranteed to occur in every pass of the garbage collector.

Garbage collection occurs implicitly before memory snapshots are recorded. In other words, clicking the Take Memory Snapshot button is the equivalent of clicking the Run Garbage Collector button and then clicking the Take Memory Snapshot button.

Run garbage collection while profiling your application

- ❖ Select the application in the Profile view, and click the Run Garbage Collector button.

You analyze the garbage collector's effectiveness by comparing two memory snapshots before and after garbage collection occurs.

Identifying problem areas

You can use a variety of techniques to identify problem areas in your applications by using the profiler.

Locating memory leaks

One of the most common problems you face in application development is memory leaks. Memory leaks often take the form of objects that were created within a period of time but not garbage collected.

One way to identify memory leaks is to look at the number of references to an object in the Instances table in the Object References view. You can generally ignore references from document properties and bindings, and look for unexpected or unusual references, especially objects that are not children of the object. For more information, see [“Using the Object References view”](#) on page 160.

You can also examine paths to instances of an object to determine if a path has a back-reference to the garbage collector (GC Root). An instance that was expected to be released, but has references to GC Root, indicates a memory leak. In these cases, modify your application code so references to GC Root are removed. An instance that has no references to GC Root is ready for garbage collection. Flash Player eventually frees this memory.

Another way to locate memory leaks is to compare two memory snapshots in the Loitering Objects view to determine which objects are still in memory after a particular series of events. This process is described in this section.

Common ways to clean up memory links are to use the `disconnect()`, `clearInterval()`, and `removeEventListener()` methods.

Finding a back-reference to GC Root for an instance

- 1 Create a memory snapshot.

See “[Create and view a memory snapshot](#)” on page 159.

- 2 Specify the number of back-reference paths to find.

From the Memory Snapshot view, specify the maximum number of paths to find, or select Show All Back-Reference Paths.

- 3 Double-click a class in the memory snapshot to open the Object Reference view.

- 4 Expand the listed paths to and examine if there is a back-reference to GC Root.

Find loitering objects

- 1 Create two memory snapshots.

See “[Create and view a memory snapshot](#)” on page 159.

- 2 Select the two memory snapshots to compare.

Note: If you have more than two memory snapshots, you cannot select a third one. You can compare only two memory snapshots at one time.

- 3 Click the Find Loitering Objects button.

Loitering Objects view shows you potential memory leaks. The Instances and Memory columns show the differences between the number of instances of a class and the amount of memory consumed by those instances during the interval between one snapshot and the next. If you see a class that you did not expect to be created during that time, or a class that you expected to be destroyed during that time, investigate your application to see if it is the source of a memory leak.

- 4 To determine how an object in the Find Loitering Objects view was instantiated, double-click the object in the view. The Object References view shows you the stack trace for each instance that you selected.

One approach to identifying a memory leak is to first find a discrete set of steps that you can do over and over again with your application, where memory usage continues to grow. It is important to do that set of steps at least once in your application before taking the initial memory snapshot so that any cached objects or other instances are included in that snapshot.

Then you perform that set of steps in your application a particular number of times—3, 7, or some other prime number—and take the second memory snapshot to compare with the initial snapshot. In the Find Loitering Objects view, you might find loitering objects that have a multiple of 3 or 7 instances. Those objects are probably leaked objects. You double-click the classes to see the stack traces for each of the instances.

Another approach is to repeat the sequence of steps over a long period of time and wait until the memory usage reaches a maximum. If it does not increase after that, there is no memory leak for that set of steps.

Common sources of memory leaks include lingering event listeners. You can use the `removeEventListener()` method to remove event listeners that are no longer used. For more information, see Object creation and destruction in *Building and Deploying Adobe Flex3 Applications*.

Analyzing execution times

By analyzing the execution times of methods and the amount of memory allocated during those method calls, you can determine where performance bottlenecks occur.

This is especially useful if you can identify the execution time of methods that are called many times, rather than methods that are rarely called.

Determine frequency of method calls

- 1 Start a profiling session and ensure that you enable performance profiling when configuring the profiler on the startup screen.
- 2 Select your application in the Profile view.
- 3 Interact with your application until you reach the point where you want to start analyzing the number of method calls. To see how many times a method was called from when the application started up, do not interact with the application.
- 4 Click the Reset Performance Data button. This clears all performance data so that the next performance profile includes any data from only this point forward.
- 5 Interact with your application until you reach the point where you check the number of method calls since you reset the performance data.
- 6 Click the Capture Performance Profile button.
- 7 Double-click the performance profile in the Profile view.
- 8 In the Performance Profile view, sort the data by the Method column and find your method in the list.

The value in the Calls column is the number of times that method was called during this sampling interval. This is the time between when you clicked the Reset Performance Data button and when you clicked the Capture Performance Profile button.

Examine the values in the Cumulative Time, Self Time, Avg. Cumulative Time, and Avg. Self Time columns of the Performance Profile view. These show you the execution time of the methods.

Compare the time each method takes to execute against the time that all the methods that are called by a particular method take to execute. In general, if a method's self-time or average self-time are high, or high compared to other methods, you should look more closely at how the method is implemented and try to reduce the execution time.

Similarly, if a method's self-time or average self-time are low, but the cumulative time or average cumulative time are high, look at the methods that this method calls to find the bottlenecks.

Locating excessive object allocation

One way to identify trouble areas in an application is to find out where you might be creating an excessive number of objects. Creating an instance of an object can be an expensive operation, especially if that object is in the display list. Adding an object to the display list can result in many calls to style and layout methods, which can slow down an application. In some cases, you can refactor your code to reduce the number of objects created.

After you determine whether there are objects that are being created unnecessarily, decide whether it is reasonable or worthwhile to reduce the number of instances of that class. For example, you could find out how large the objects are, because larger objects generally have the greatest potential to be optimized.

To find out which objects are being created in large numbers, you compare memory snapshots of the application at two points in time.

View the number of instances of a specific class

- 1 Start a profiling session and ensure that you enable memory profiling when configuring the profiler on the startup screen.
- 2 Interact with your application until you reach the place to take a memory snapshot.
- 3 Click the Take Memory Snapshot button. The profiler adds a new memory snapshot to the application list in the Profile view.
- 4 Open the memory snapshot by double-clicking it in the Profile view.
- 5 To view the number of instances of a particular class, and how much memory those instances use, sort by the Class column and find your class in that column. You can also sort by the other columns to quickly identify the objects that take up the most memory or the objects with the most instances. In most cases, Strings are the class with the most instances and the most memory usage.

For more information about the Memory Snapshot view, see [“Using the Memory Snapshot view”](#) on page 159.

Locate instances of excessive object allocation

- 1 Start a profiling session and ensure that you enable memory profiling when configuring the profiler on the startup screen.
- 2 Interact with your application until you reach the first place to take a memory snapshot.
- 3 Click the Take Memory Snapshot button.
The profiler saves the memory snapshot in the Profile view, and marks the snapshot with a timestamp.
- 4 Interact with your application until you reach the second place to take a memory snapshot.
- 5 Click the Take Memory Snapshot button again.
The profiler saves the second memory snapshot in the Profile view, and marks the snapshot with a timestamp.
- 6 Select the two memory snapshots to compare.

Note: *If you have more than two memory snapshots, you cannot select a third one. You can compare only two at a time.*

- 7 Click the View Allocation Trace button.

The Allocation Trace view shows which methods were called between the two snapshots and how much memory was consumed during those method calls. See [“Using the Allocation Trace view”](#) on page 162.

About profiler filters

The amount of data in a profiler view can sometimes be overwhelming and the level of detail can be too great. The internal actions of Flash Player might obscure the data that you are truly interested in, such as your own methods and classes. Also, Flash Player creates and destroys many objects without your direct interaction. Thus, you could see that thousands of strings or arrays are being used in your application.

You can set filters in the following views:

- Live Objects
- Memory Snapshot
- Performance Profile
- Method Statistics
- Allocation Trace

You can define which packages should appear in the profiler views. You do this by using the profiler filters. There are two types of filters:

Exclusion filters The exclusion filters instruct the profiler to exclude from the profiler views packages that match the patterns in its pattern list. If you use charting controls, for example, but do not want to profile them, you can add the `mx.charts.*` pattern to the exclusion filter. You can also exclude global built-in items. These include global classes such as `String` and `Array`.

Inclusion filters The inclusion filters instruct the profiler to include in the profiler views only those packages that match the patterns in its pattern list. If you have a custom package named `com.mycompany.*`, for example, you can view details about only classes in this package by adding it to the inclusion filter.

The default exclusions are `flash.*`, `spark.*`, `mx.*`, and the Flex framework classes in the global or unnamed package. These include global classes such as `String` and `Array`. This means that the default inclusions are user-defined classes in the unnamed package and user-defined classes in nonframework packages (such as `com.mycompany.MyClass`).

You can exclude user-defined classes that are in the unnamed package from the profiling data. To do this, add `*` to the exclusion list.

Maximum visible rows Maximum visible rows sets the number of rows of data that can be displayed in a view. Increase this value if the data you are looking for is not displayed in the view. Decrease this value to improve performance of the profiler. Use other filters to ensure that you are displaying the data of interest to you.

Maximum back-reference paths to find: Maximum back-reference paths sets the number of paths to a referenced object to display when examining object references. The paths are displayed according to the shortest path. By default, the ten shortest paths are displayed. Increase this value to display additional paths or select Show All Back-Reference Paths. Showing additional paths can help you locate objects that are no longer referenced. See [“Locating memory leaks”](#) on page 171.

Set default filter preferences

- ❖ Open the Preferences dialog and select Flash Builder > Profiler > Inclusion Filters or Exclusion Filters.

When displaying profiler data, the profiler applies the exclusion filters first; then it applies the inclusion filters. For example, suppose you set the exclusion filter to `mx.controls.*`, but set the inclusion filter to `mx.*.*`; the profiler does not show details about any classes in the `mx.controls` package because that package was excluded, even though their pattern matches the inclusion pattern list. Similarly, suppose you set the exclusion filter to `mx.*.*` and the inclusion filter to `mx.controls.*`; the profiler does not show details about any classes in `mx.controls.*` package because they were excluded before it was included.

When you filter out certain data points, the percentage values of columns are adjusted to reflect only the percentage of non-filtered data.

The profiler maintains filters from one profiling session to the next for the same application.

The filter settings are not inherited by subviews. For example, if you apply a filter to the data in the Memory Snapshot view, and then navigate to the Object References view by double-clicking a method, the Object References view does not apply the same filter.

Determine whether data is being filtered

- 1 Click the Filter button or look at the titles of the data tables. If there are filters applied, the Package column's heading is Package (Filtered).
- 2 (Optional) Reset the filters to the default by clicking the Restore Defaults button.

Chapter 9: Building a user interface with Flash Builder

About the structure of user interfaces in Flex

The building blocks of user interfaces in Flex are MXML containers and controls. A container is a rectangular region that you use to organize and lay out controls, other containers, and custom components. A control is a user interface component such as a Button, TextArea, or ComboBox. Custom components are components that you have defined in separate MXML or ActionScript files, and include in your application.

Note: You can also add assets created with Adobe® Flash® Professional to your application. See [“Creating and editing Flash components”](#) on page 183.

Spark components

Spark components are new for Flex 4 and are defined in the spark.* packages. Components available in previous releases of Flex, referred to as MX components, are defined in the mx.* packages.

The main differences between Spark and MX components are how you use CSS styles with the components and how you skin them. For details on using visual components in an application, see [Visual components](#).

Spark and MX containers

Flex defines two sets of containers: Spark containers and MX containers. Spark containers are defined in the spark.components package while MX containers are defined in the mx.core and mx.containers packages. Spark containers specify layout differently from MX containers.

While you can use MX containers to perform most of the same layout that you can perform using the Spark containers, Adobe recommends that you use the Spark containers when possible.

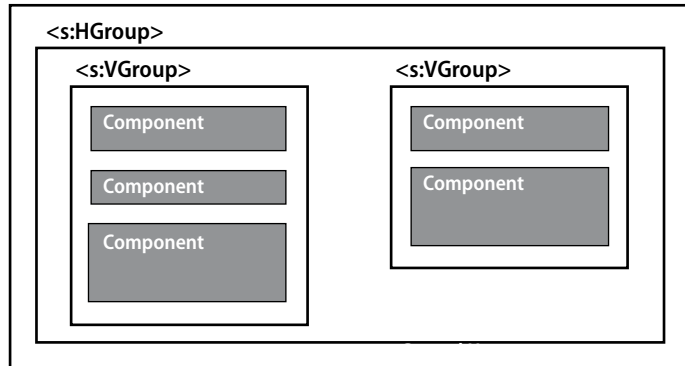
For details on using containers to layout an application, see [Introduction to containers](#).

Spark containers

Applications in Flex typically consist of an MXML application file (a file with an `<s:Application>` container as the parent tag, and one or more components defined in separate MXML files, ActionScript files, or Flash component files (SWC files). You can insert containers and controls directly in the MXML application file, or you can insert them in separate MXML files to create custom components and then insert the custom components in the application file.

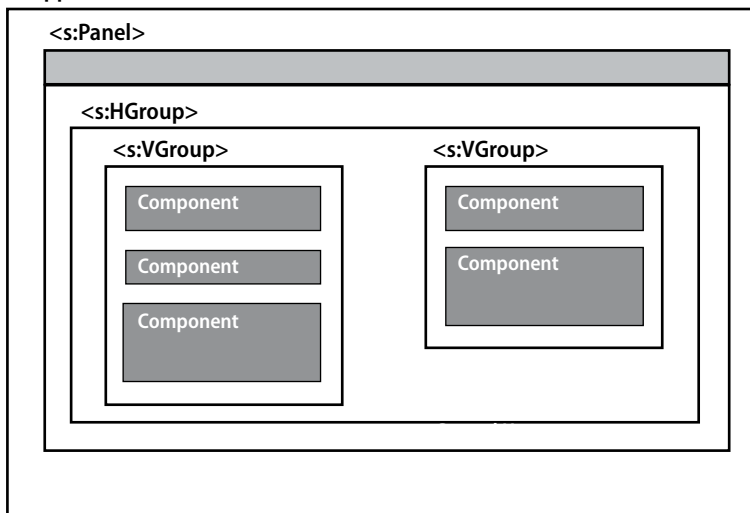
The following example shows a simple structure for an application. The containers and controls are inserted directly into the MXML application file.

`<s:Application>`



Here is a similar example that uses a Panel container as the base structure for the containers and components.

`<s:Application>`

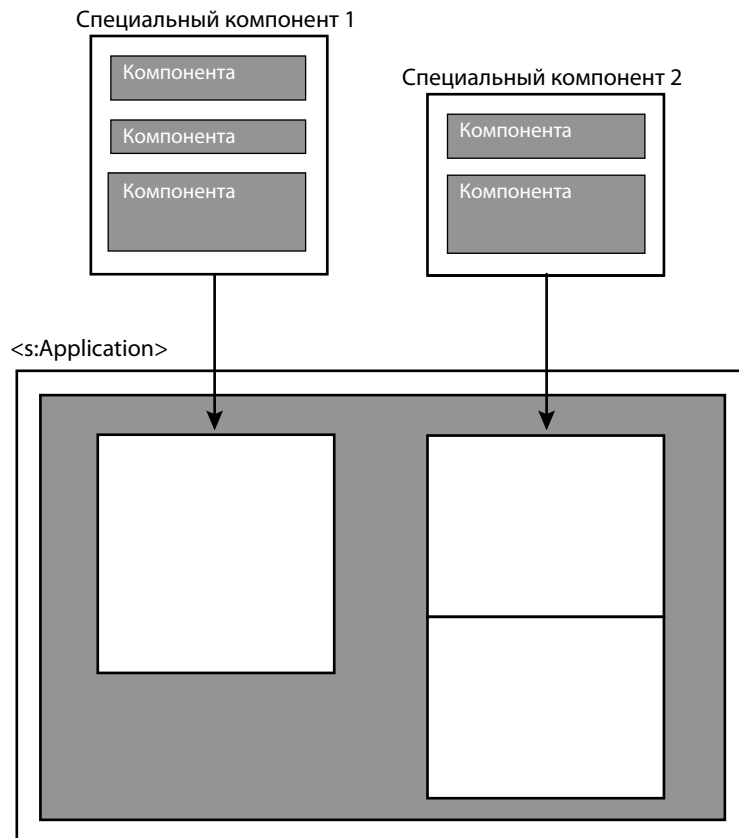


For more information about containers, components, and laying out a user interface, see *Building the user interface*, including the sections *Introduction to containers* and *Visual components*.

Component-based structure for applications

A component-based structure is useful when your user interface consists of distinct functional elements. For example, your layout could have an element that retrieves and displays a product catalog, another element that retrieves and displays details about any product that the user clicks in the catalog, and an element that lets the user add the selected product to a shopping cart. This user interface could be structured as three custom components or as a mixture of custom components and controls inserted directly into the layout.

The following example shows a component-based structure for an application. You group the controls of the user-interface elements in separate custom component files, which you then insert into the MXML application file.



More Help topics

[“Creating Custom MXML Components”](#) on page 126

Layouts for Spark containers

Most Spark containers, including the top-level `<s:Application>` container, allow you to specify a layout that defines the positioning of components added to the container. By default, these Spark containers use `BasicLayout`.

You can specify the following layouts for Spark containers:

- `BasicLayout`
Uses absolute positioning. You explicitly position all container children, or use constraints to position them. This is the default layout for all containers except `HGroup` and `VGroup`.
- `HorizontalLayout`
Lays out children in a single horizontal row.
- `VerticalLayout`
Lays out children in a single vertical column.
- `TileLayout`
Lays out children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary.

The HGroup and VGroup containers specify `HorizontalLayout` and `VerticalLayout`, respectively. You typically do not override the default layout of these containers.

More Help topics

[“MX containers”](#) on page 180

Spark container children

All Spark containers can take as children, Spark and MX visual components.

While you can use MX containers to perform most of the same layout that you can perform using the Spark containers, Adobe recommends that you use the Spark containers when possible.

MX containers

One of the main differences between Spark and MX containers is that the layout algorithm for MX containers is fixed, but for Spark containers it is selectable. Therefore, MX defines a different container for each type of layout. Spark defines a smaller set of containers, but lets you switch the layout algorithm to make them more flexible.

Spark containers can take as children, Spark and MX visual components. However, MX navigator components cannot take Spark components as children. MX navigator components include the `ViewStack`, `TabNavigator`, and `Accordion` containers. Other MX navigator components include the `TabBar`, `ButtonBar` and other similar components.

Some components implemented in MX do not have corresponding equivalents in Spark. In these cases, the MX components are included as a recommended component in the Components view and when using Content Assist.

More Help topics

[“Components view”](#) on page 181

[“About Content Assist”](#) on page 98

Adding and changing components

Use Flash Builder to add, size, position, edit, or delete components in an application. You can also add and edit custom components defined in separate MXML and ActionScript files.

Add components in MXML Design mode

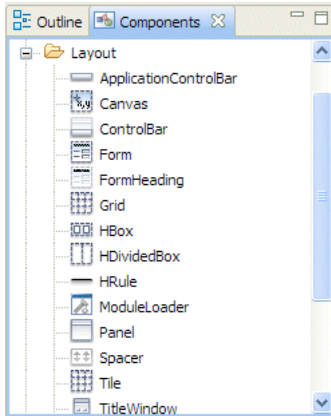
You add standard containers and controls to your user interface in MXML Design mode. You drag and drop components from the Components view to the Design area of the MXML file and position them according to the layout rule of the container. You can also add custom components that you define in separate MXML and ActionScript files and save in the current project or in the source path of the current project.

- 1 In the MXML editor's Design mode, open the MXML file in which you want to insert the component.

An MXML file must be open in Design mode to use the Components view. The MXML file can be the main application file (a file with an `Application` container) or a custom MXML component file.

- 2 In the Components view, locate the component that you want to add.

If the Components view is not open, select Window > Components.



Components view organizes components by category. The Components view also displays recommended components for a project. You can change the view settings to list all components. For more information, see [“Components view”](#) on page 181.

- 3 Drag a component from the Components view into the MXML file.

The component is positioned in the layout according to the layout rule of the parent container.

Components view

When you switch to Design mode of the MXML editor, Flash Builder displays the Components view, which you use to drag components to the design area as you lay out your application.

By default, the Components view displays recommended components for a project. The version of the Flex SDK for a project determines which components appear as recommended components.

For example, when creating a new project, Flash Builder, by default, uses the Flex 4 SDK. Recommended components for the Flex 4 SDK are Spark components. MX components that do not have an equivalent Spark component also appear as recommended components.

The Components view groups components according to category, such as Controls, Layout, Navigators, and Charts. There is also a Custom category that lists components defined in separate MXML and ActionScript files. For more information, see [“Creating Custom MXML Components”](#) on page 126.

Note: The Components view lists visible components.

Modify how components appear in Components view

Use the View menu within the Components view to modify how components appear.

- 1 To show all components, deselect Only Show Recommended Components.

When you specify to show all components, Components view groups the components according to Spark and MX components.

- 2 To show fully-qualified class name, select Show Fully Qualified Class Names.

Add components by writing code

When using Source mode of the MXML editor, code hinting assists you when adding standard Flex containers and controls to your user interface. In Flash Builder, as in Eclipse, code hinting is called Content Assist.

Content Assist in Flash Builder helps you insert recommended components. The following example shows how to use code hints to insert an `<s:VGroup>` container into an `<s:HGroup>` container. Because `HGroup` is a Spark component, Content Assist recommends a `VGroup` container, and not `<mx:VBox>`.

- 1 Open an MXML file in the MXML editor's Source mode.

The MXML file can be the main application file (a file with an `Application` container) or a custom MXML component file.

- 2 Place the insertion point in the parent container tag.

For example, to insert a `VGroup` container inside an `HGroup` parent container, place the insertion point after the opening `<s:HGroup>` tag:

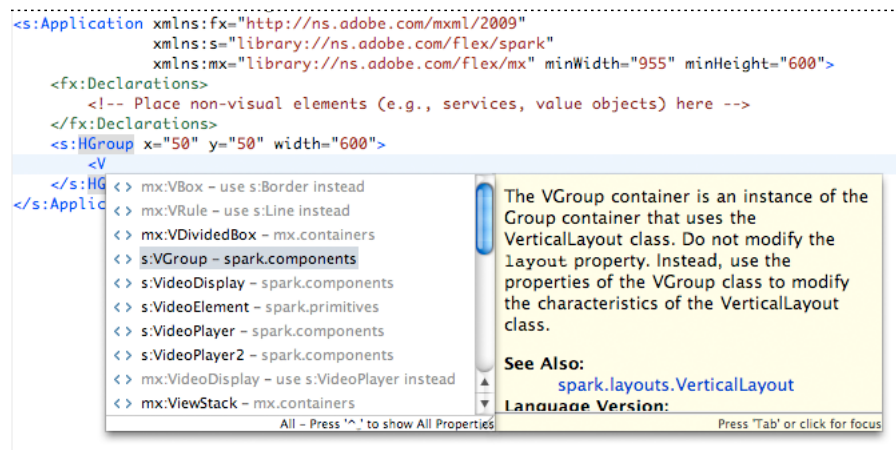
```
<s:HGroup>
    insertion point here
</s:HGroup>
```

- 3 Type the component tag.

As you type the tag, Content Assist appears suggesting possible entries. Recommended components are listed in the normal black type color. Components that are not recommended appear gray.

In this example, `VGroup` is recommended and `VBox` is not recommended.

- 4 If necessary, use the arrow keys to select your tag from the menu, then press Enter.



In addition to the standard components, Content Assist lists custom components you defined in separate MXML and ActionScript files and saved in the current project or in the source path of the current project.

Content Assist can also suggest properties, events, effects, and styles. Press `Control+Space` to cycle through the recommendations in Content Assist.

You can change the type and order of recommendations for Content Assist. From the Preferences dialog, select `Flash Builder > Editors > MXML Code > Advanced`.

More Help topics

[“About Content Assist”](#) on page 98

[“Creating Custom MXML Components”](#) on page 126

Creating and editing Flash components

Adobe Flash® Professional CS5 creates applications compatible with Adobe Flash Player 10. Adobe applications also support Flash Player 10, which means that you can import assets from Flash Professional CS5 to use in your applications. You can create controls, containers, skins, and other assets in Flash Professional CS5, and then import those assets into your application as SWC files. For information on creating components using Flash Professional CS5, see [Flex Skin Design Extensions and Flex Component Kit for Flash Professional](#).

In the Design View of the Flash Builder editor, you can insert a new Flash component by adding a placeholder for a Flash component or container. You can invoke Flash Professional CS5 from Flash Builder to create the component or container. You can also invoke Flash Professional CS5 to edit previously created Flash components.

If your application contains an SWFLoader component to launch Flash movie files, you can launch Flash Professional CS5 to create or edit the associated FLA and SWF files.

Inserting a Flash component or Flash container

- 1 In the Flash Builder editor, select Design View and make sure the Components View is visible.
- 2 From the Custom folder in the Components View, drag either a New Flash Component or a New Flash Container to the design area.

You can resize or position the component or container.

- 3 From either the context menu or the Standard View of the File Properties window, select Create in Adobe Flash.

Note: You can also double-click the component in Design View to create the item in Adobe Flash.

- 4 In the dialog that opens, specify names for the class and the SWC file, then click Create to open Adobe Flash Professional CS5.
- 5 In Flash Professional CS5, edit the component or container. Select Done when you are complete to return to Flash Builder.

Editing a Flash component or Flash container

This procedure assumes you have previously inserted a Flash component or container into Flash Builder.

- 1 In Design View of the Flash Builder editor, select the Flash component or container you want to edit.
- 2 From either the context menu or the Standard View of the Flex Properties window, select Edit in Adobe Flash.

Note: You can also double-click the component in Design View to edit the item in Adobe Flash.

- 3 In Flash Professional CS5, edit the component or container. Select Done when you are complete to return to Flash Builder.

Creating or editing a Flash movie associated with a SWFLoader component

This procedure assumes your application contains an SWFLoader component.

- 1 In Design View of the Flex editor, select the SWFLoader component.
- 2 From either the context menu for the component or the Standard View of the Flex Properties window, launch Flash Professional CS5 by doing one of the following:
 - Select Create in Adobe Flash to create a new Flash movie associated with the SWFLoader component.
 - Select Edit in Adobe Flash to edit the Flash movie associated with the SWFLoader component.
- 3 After you have finished editing the movie, select Done to return to Flash Builder.

Importing Flash CS3 Assets

You can add Flash components that were created in Adobe Flash CS3 Professional to your user interface.

Note: Before you can create Flex components in Flash CS3, you must install the Flex Component Kit for Flash CS3. For more information, see the article [Importing Flash CS3 Assets into Flex](#).

- 1 Ensure that the Flash component is saved in the library path of the current project.

The library path specifies the location of one or more SWC files that the application links to at compile time. The path is defined in the Flex compiler settings for the project. In new projects the `libs` folder is on the library path by default.

To set or learn the library path, select the project in the Package Explorer and then select Project > Properties. In the Properties dialog box, select the Flex Build Path category, and then click the Library Path tab. For more information, see [“Building projects manually”](#) on page 76.

The library path can also be defined in the `flex-config.xml` configuration file in Adobe LiveCycle® Data Services ES.

- 2 Open an MXML file and add a Flash component in one of the following ways:
 - In the MXML editor’s Design mode, expand the Custom category of the Components view and drag the Flash component into the MXML file. For documents that are already open, click the Refresh button (the green circling arrows icon) to display the component after you insert it.
 - In Source mode, enter the component tag and then use Content Assist to quickly complete the tag.

Working with components visually

Flash Builder lets you work with components visually in the MXML editor so you can see what your application looks like as you build it. The MXML editor has two modes: Source mode for writing code, and Design mode for developing applications visually.

Using the MXML editor in Design mode

In Design mode you can view, select, pan, move, resize, scroll, and magnify items in the design area.

View an MXML file

- 1 If the MXML file is not already open in the MXML editor, double-click the file in the Package Explorer to open it.
- 2 If the MXML editor displays source code, click Design at the top of the editor area.

You can quickly switch between modes by pressing Control+` (Left Quote).

Switching between Source and Design modes automatically shows or hides design-related views like the Components, Properties, and States views. To turn this behavior on and off, from the Preferences dialog, select Flex > Editors > Design Mode, then select the Automatically Show Design-related Views option.

Select and move components in the design area

- ❖ Click the Select Mode (arrow) button on the right side of the editor toolbar. Select Mode is activated by default when you open a document. Press V on the keyboard to enter Select Mode. Click and drag a component to the desired place. You can also drag to resize and click to select.



Pan and scroll in the design area

- ❖ Click the Pan Mode button on the right side of the editor toolbar. Press H to enter Pan Mode from the keyboard. To temporarily enter Pan Mode, press and hold the spacebar on the keyboard. You cannot select or move items in Pan Mode.

Zoom in the design area

There are several ways to use the zoom tool. You can select percentages from the main and pop-up menus, click the Zoom Mode button on the toolbar, or use keyboard shortcuts. The current magnification percentage is always displayed in the toolbar.

- From the main menu select Design > Zoom In or Design > Zoom Out. You can also select the Magnification submenu and choose a specific percentage.
- Click the Zoom Mode button on the toolbar or press Z from the keyboard. A plus symbol cursor will appear in the design area.
- Select a percentage from the pop-up menu next to the Select, Pan, and Zoom Mode buttons on the editor toolbar. The design area changes to the selected percentage or fits to the window.
- Right-click in the design area to select Zoom In, Zoom Out, or the Magnification submenu. The design area changes to your selection.

You can always use the following keyboard shortcuts from the design area:

- Zoom In: Ctrl+= (Command+= on Mac OS)
- Zoom Out: Ctrl+- (Command+- on Mac OS)

For more keyboard shortcuts, select Help > Key Assist.

Selecting multiple components in an MXML file

You can select more than one component in an MXML file. This can be useful if you want to set a common value for a shared property.

- Control-click (Command-click on Macintosh) each component in the layout.
- Click the page background and draw a box that overlaps the components.
- In Outline view (Window > Outline), Control-click (Command-click on Macintosh) the components in the tree control.

Deselecting multiple components

- Click the background container.
- Click an unselected component.
- Click in the gray margin around the root component.

Positioning components

You can change the position of components visually depending on the layout property of the parent container. If the parent container specifies absolute positioning, you can drag and drop components to reposition them. Otherwise, repositioned components follow the layout rules of the parent container.

By default, Spark containers use the BasicLayout property, which allows absolute positioning. For some MX containers, you can also specify a layout property of `absolute`. For more information, see MX layout containers.

Note: You can also use layout constraints to position a component in a container. Layout constraints specify how to reposition components when a container resizes. However, they can also be used to specify position in containers that have a fixed size. See [“Setting layout constraints for components”](#) on page 193.

- 1 In the MXML editor’s Design mode, select the component in the layout and drag it to a new position.

The component is positioned in the layout according to the layout rules of the parent container.

If the container has absolute positioning, you can drag and position components anywhere in the container.

If you move a VGroup container in an HGroup container, the VGroup container is positioned into the horizontal arrangement with the other child containers (if any).

- 2 In Design mode, select the component’s parent container and edit the component’s layout properties in the Properties View.

In some cases, you can change the position of child components by changing the properties of the parent container. For example, you can use the `verticalGap` and `horizontalGap` properties of a container to set the spacing between child components and the `direction` property to specify either a row or column layout.

More Help topics

[“Spark and MX containers”](#) on page 177

[“Setting layout constraints for components”](#) on page 193

Sizing components

You can change the size of a component in Design mode by dragging a resize handle, selecting menu options, or by editing its properties in the Properties View.

Note: Use a constraint-based layout to specify how a component dynamically resizes when a container resizes. For more information, see [“Setting layout constraints for components”](#) on page 193.

Size a component visually

- ❖ In the MXML editor’s Design mode, click on the component and drag a resize handle to resize the component.
 - To constrain the proportions of the component, hold down the Shift key while dragging.
 - If snapping is enabled, as you resize, snap lines appear to line up the edges of the component with nearby components. To enable or disable snapping from the main menu, select Design > Enable Snapping.

Make two or more components the same width or height

- 1 In Design mode, select two or more components.
- 2 In the Design menu, select one of the following sizing options:

Make Same Width Sets the `width` property for all selected components to that of the component you selected first.

Make Same Height Sets the `height` property for all selected components to that of the component you selected first.

If all selected components are in the same container, and the first component you select has a percent width or height specified, all items are set to that percent width or height. Otherwise, all components are set to the same pixel width or height.

Size a component by editing its properties

- 1 In Design mode, select the component.

You can Control-click (Shift-click on Mac OS) more than one component to set their sizes simultaneously.

- 2 In the Properties View (Window > Properties), set the `height` or `width` property of the selected component or components.

The Properties View provides three views for inspecting a component's properties: a standard form view, a categorized table view, and an alphabetical table view. You can switch between them by clicking the view buttons in the toolbar.

Using snapping to position components

When you drag a component visually in a container that has absolute positioning, the component may snap into place, depending on where you drop it relative to existing components. The components can line up vertically or horizontally.

By default, Spark containers use the `BasicLayout` property, which allows absolute positioning. For some MX containers, you can also specify a layout property of `absolute`. For more information, see [Using Layout Containers](#).

You can disable snapping for one component or for all components.

Enable or disable snapping

- ❖ With the MXML file open in the MXML editor's Design mode, select (or deselect) Design > Enable Snapping.

Enable or disable snapping as a preference

- 1 Open the Preferences dialog.
- 2 Select Flash Builder > Editors > Design Mode in the sidebar of the Preferences dialog box.
- 3 Select or deselect the Enable Snapping option.

Aligning components

You can visually align components relative to each other in a container that has absolute positioning. By default, Spark containers use the `BasicLayout` property, which allows absolute positioning. For some MX containers, you can also specify a layout property of `absolute`.

You can also center components in a container by using a constraint-based layout. For more information, see ["Setting layout constraints for components"](#) on page 193.

Align components in a container that has absolute positioning

- 1 In the MXML editor's Design mode, select two or more components in the container.

For more information, see ["Selecting multiple components in an MXML file"](#) on page 185.

- 2 Select one of the following alignment options from the Design menu:

Align Left Positions all selected components so that their left edges align with that of the first component you selected.

Align Vertical Centers Positions all selected components so that their vertical center lines are aligned with the vertical center line of the first component you selected.

Align Right Positions all selected components so that their right edges align with that of the first component you selected.

Align Top Positions all selected objects so that their top edges align with that of the first component you selected.

Align Horizontal Centers Positions all selected components so their horizontal center lines are aligned with the horizontal center line of the first component you selected.

Align Bottom Positions all selected components such that their bottom edges align with that of the first component you selected.

Align Baselines Positions all selected components so that their horizontal text baselines are aligned with that of the first component you selected. For components that have no text baseline (such as HGroup), the bottom edge is considered the baseline.

For objects with no layout constraints, Flash Builder adjusts the *x* property to change the vertical alignment, and adjusts the *y* property to change the horizontal alignment.

For objects with layout constraints, Flash Builder adjusts the left and right constraints to change the vertical alignment and adjusts the top and bottom constraints to change the horizontal alignment. Only existing constraints are modified; no new constraints are added.

For example, suppose component A has a left constraint and no right constraint, and component B has both a left and right constraint. If you select component A and B and then select Design > Align Vertical Centers, Flash Builder adjusts the left constraint of object A and both the left and right constraints of object B to align them. The unspecified right constraint of object A remains unspecified.

Nudging components

You can fine-tune the position of components in a container that has absolute positioning by adjusting the components one pixel or ten pixels at a time in any direction with the arrow keys.

Nudge components by one pixel

- ❖ Select one or more components in the MXML editor's Design mode and press an arrow key.

Nudge components by ten pixels

- ❖ Select one or more components in the MXML editor's Design mode and press an arrow key while holding down the Shift key.



Holding down the arrow key continues to move the component.

Setting component properties

You visually set the properties of components in the design area or in the Properties view.

Edit the text displayed by a component

- ❖ To edit text displayed by a component such as a Label or TextInput control, double-click the component and enter your edits.

Change text in the ID field

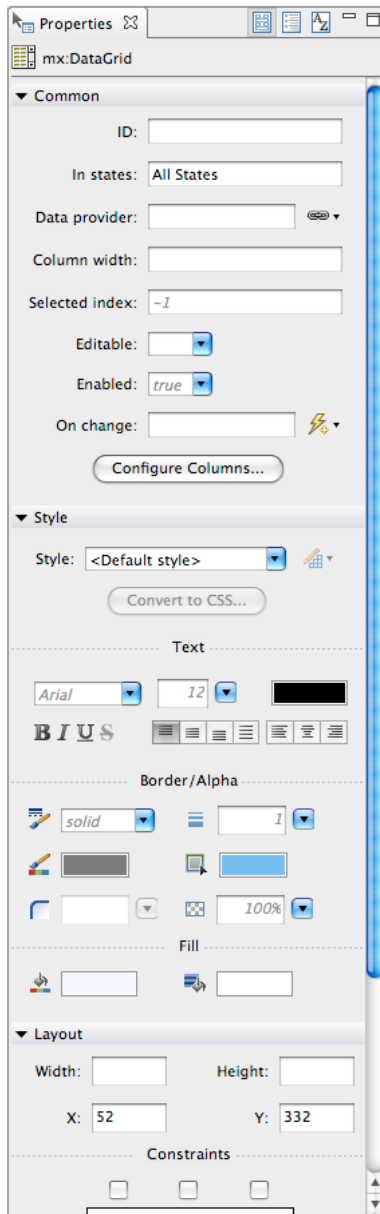
When you change text in the ID field, you are prompted to update all references in the workspace with the new ID. You can suppress this dialog box on the Design Mode preferences page:

- 1 Open the Preferences window.
- 2 select Flash Builder > Editors > Design Mode.

- 3 Select or deselect Always Update References When Changing IDs in the Properties View.

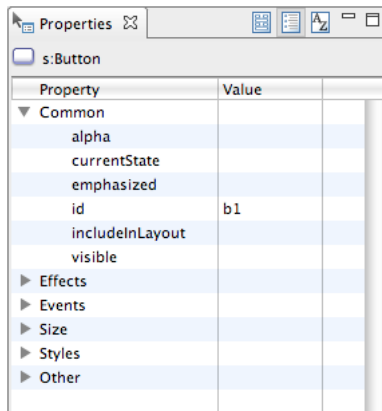
Set other properties of a component

- ❖ Select the component and set its properties in the Properties view (Window > Other Views > Flash Builder > Properties).



Properties view for a DataGrid component

To set the properties in Category view or Alphabetical view, click the view buttons in the toolbar:



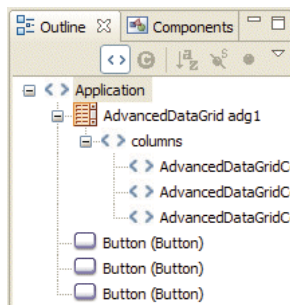
Category view for a Button component

Note: To apply your last edit, press Enter or Tab, or click outside the view.

Inspecting the structure of your MXML

Use Outline view (Window > Outline) in Design mode to inspect the structure of your design and to select one or more components. When you have multiple view states, Outline view shows you the structure of the current view state.

- 1 With the MXML file open in Design mode, select Outline view.



- 2 In Outline view, select one or more components.

The selected components in Outline view are also selected in Design mode of the editor.

In Source mode of the editor, only the first selected component in Outline view is also selected in the editor.

Hiding container borders

By default, Flash Builder shows the borders of containers in the MXML editor's Design mode. However, you can hide these borders.

- ❖ Select Design > Show Container Borders.

This command is a toggle switch. Select it again to show the borders.

Copying components to other MXML files

You can visually copy and paste components from one MXML file to another.

- 1 Make sure the two MXML files are open in the MXML editor's Design mode.

- 2 Select the component or components in one file. Then select Edit > Copy.
- 3 Switch to the other file, click inside the desired container, and select Edit > Paste.

Deleting components

You can delete components from your user interface. Select a component and do any of the following actions:

- Press the Delete key on your keyboard
- From the context menu for a component, select Delete.
- From the Flash Builder Edit menu, select Delete.

Using constraint-based layouts

Use constraints on a component to automatically adjust the component's size and position in the container when a user resizes the application window.

You typically define layout constraints in Design mode of the MXML editor. You can also edit a component's properties in Source mode to define layout constraints.

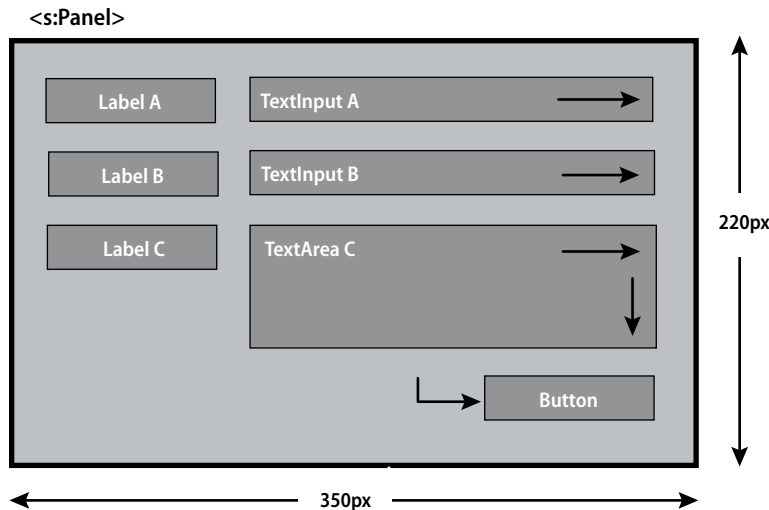
About constraint-based layouts

Flex supports constraint-based layouts. Constraint-based layouts are available in containers that support absolute positioning. For Spark containers, the default layout, `BasicLayout`, supports absolute positioning.

When using constraint-based layouts, you anchor one or more sides of a component to the edges of a container or a container's constraint region. You can also specify an offset for the component w/respect to the anchor. When a user resizes the container, the size and position of the container components are determined by the defined anchor points.

Constraint-based layout example

In the following example, all the controls are absolutely positioned in a container either by dragging them or by setting the *x* and *y* coordinates in the Properties view. The arrows in the figure indicate how constraints, specified according to the following bulleted list, make the controls behave when the user resizes the layout.



A number of layout constraints are applied to the controls to ensure that the layout adjusts correctly when the user resizes the application:

- Label A, Label B, and Label C are anchored to the left and upper edges so the labels remain in place as the user resizes the layout.
- TextInput A and TextInput B are anchored to the left and right edges so the controls stretch or compress horizontally as the user resizes the layout.
- TextArea C is anchored to the left and right edges and to the upper and lower edges so that the control stretches or compresses horizontally and vertically as the user resizes the layout.
- The Button control is anchored to the right and lower edges so that the control maintains its position relative to the lower-right corner of the container as the user resizes the layout.

The TextInput A and TextInput B controls stretch horizontally as the layout is enlarged. TextArea C control stretches horizontally and vertically. The Button control moves down and to the right.

Constraint-based layouts and absolute positioning

To create a constraint-based layout, use a container that supports absolute positioning. For Spark containers, set the layout property to `BasicLayout`. If you do not specify a layout property for a Spark container, the default property is `BasicLayout`.

For MX containers, the absolute layouts can be used only with the Application, Canvas, and Panel containers. For the Canvas container, absolute layouts are the default. For MX Application and Panel containers, set the layout property to `absolute` to implement absolute positioning.

For MX containers, the `layout="absolute"` property overrides the container's layout rule and lets you drag and position components anywhere in the container.

Using advanced constraint layouts

You can also define constraints that anchor to horizontal and vertical constraint regions. Advanced constraints are useful in the following situations:

- When controls dynamically resize to fit their content. For examples, controls that display localized strings.
- When you want multiple columns that are the same size or that need to remain a specific percentage width.

Note: You can use nested *HGroup* and *VGroup* containers to achieve results similar to advanced constraints.

For advanced constraint layouts, define *ConstraintColumn* regions and *ConstraintRow* regions. Constrain components to the edges or centers of these regions. Constraint columns are laid out in the container from left to right while constraint rows are laid out from top to bottom.

Constraint regions can be defined with fixed pixel dimensions (width or height) or as a percentage of the space in the parent container. The set of constraint columns and rows may have any combination of fixed or percentage dimensions.

Components within a parent container are constrained to the container, to constraint regions, or to any combination of container and region constraints.

Use the MXML editor in Source mode to specify advanced constraints.

Setting layout constraints for components

You can specify a constraint-based layout for components in a container that has absolute positioning.

Note: Layout constraints relative to constraint columns and rows can only be set by editing the source code.

- 1 Ensure that the open MXML file includes a container that has absolute positioning.
- 2 In the MXML editor's Design mode, drag components from the Components view into the container.
- 3 Position the component in the container either by moving it in the design area or by setting the *x* and *y* properties in the Properties View (Window > Properties).
- 4 Select a component in the design area.
- 5 In Properties View, scroll to the constraint tool in the Layout category.
- 6 Using the following table as a guide, select the constraint check boxes to achieve the effect you want when the user resizes the application:

Effect	Constraints
Maintain the component's position and size	None
Move the component horizontally	Left or Right
Move the component vertically	Top or Bottom
Move the component horizontally or vertically	Left and Top or Right + Bottom
Resize the component horizontally	Left and Right
Resize the component vertically	Top and Bottom
Resize the component both horizontally and vertically	Left and Right and Top and Bottom

Effect	Constraints
Center the component horizontally	Horizontal center
Center the component vertically	Vertical center
Center the component both horizontally and vertically	Vertical center and Horizontal center

7 Specify the distance of the constraints from the edges of the container.

For example, you can set the component to maintain its position 90 pixels from the left edge and 60 pixels from the right edge. If the user resizes the application, the component stretches or compresses to maintain these distances from the edges of the application window.

Flash Builder expresses these constraints in the MXML code as follows:

```
<s:TextInput y="160" left="90" right="60"/>
```

Note: The value of *y* is modified according to the values you specify for the left and right constraints.

Generating event handlers

Flex applications are event-driven. User interface components respond to various events, such as a user clicking a button or the initialization of an object is complete. You write event handlers in ActionScript code that define how the component responds to the event.

Note: You can also generate event handlers for non-visible items such as *RemoteObject* and *HTTPService*.

Flash Builder provides event handler assistance that generates the event handler functions for a component. Within the generated function, you write the code that defines the component behavior in response to the event.

You access event handler assistance in three ways:

- Properties View
- Context menu for an item in Design mode of the MXML editor
- Content assist for an item in Source mode of the MXML editor

About generated event handlers

When Flash Builder generates an event handler function, it places the event handler in the first Script block of the file. The function is placed at the end of the Script block. The generated event handler has protected access and accepts the appropriate subclass of *Event* as its only parameter.

Flash Builder generates a unique name for the event handler based on the component's class name or a custom name for the event handler that you specify. If you do not specify a custom name, the name is generated according to the following process:

- If an id property is defined, Flash Builder bases the name on the id property.
- If there is no id property defined for the component, Flash Builder generates a unique name, based on the component's class name.

You provide the body of the event handler. The following code block shows a generated event handler for a Button.

```

. . .
<fx:Script>
    <![CDATA[
        protected function myButton_clickHandler(event:MouseEvent):void
        {
            // TODO Auto-generated method stub
        }

    ]]>

</fx:Script>
<s:Button label="Button" id="myButton" click="myButton_clickHandler(event)"/>
. . .

```

Flash Builder designates a default event for each user interface component. For example, the default event for a Button is the click event. You can specify the event handler for the default event in the Standard View of the Properties View. To specify handlers for other events, in the Property Inspector select Category View > Events.

You can also use content assist in the Source View to generate event handlers.

Generating event handlers using the Properties View

- 1 In Design mode, select an item and then select Standard View in the Properties Inspector.

An editing field for the default event handler is visible in the Common area.

- 2 To generate an event handler for the default event:

- a (Optional) In the On *Event* text field, specify a name for the event.

For example, in the On Click text field for a Button component, specify **MyButtonClick**.

If you do not specify a name, Flash Builder generates a unique name for the event.



When specifying a name for the event handler, you have the option to specify an event parameter. If you do not specify the event parameter, Flash Builder generates the parameter with an appropriate event type.

- b Click the “lightening bolt” icon, and select Generate Event Handler.

The editor switches to Source mode, with the body of the generated event handler highlighted. Type in your implementation for the event.

- 3 To generate an event handler for any event for a selected item:

- a Select Category View and expand the Events node to view all the events for the item.

- b (Optional) Double-click the name of the event to activate the text box for the event handler name. Type the name for the event handler.

- c Click the icon in the Value field to create the event handler.

The editor switches to Source mode, with the body of the generated event handler highlighted. Type in your implementation for the event.

Generating event handlers using the context menu for an item

- 1 In Design View, open the context menu for an item.

- 2 Perform one of the following actions:

- Select the default event for the item.

For example, for a Button select Generate Click Handler.

- Select Show All Events to open the list of events in the Properties view.

Specify an event handler from the Properties view.

The editor switches to Source mode with the body of the generated event handler highlighted. Type in your implementation for the event.

Generating event handlers using content assist

- 1 In an MXML block in code view, create a component, but do not specify any events.
- 2 Enable content assist for the properties of a component by typing a space after the class name.
- 3 From the list of selected properties, select an event (for example, `doubleClick`).
- 4 Press Control+Space and select Generate Event Handler.

Flash Builder generates a unique name for the event handler and places the event handler in the Script block.

Note: If you specify a custom name for the event handler, then Flash Builder cannot generate the handler. If you want to use a custom name, first generate an event handler and then modify the name of the handler in both the event property and the generated handler.

Applying themes

Themes allow you to implement a more personalized appearance to your applications. Flash Builder provides several themes from which you can choose. You can import additional themes or create your own themes.

Themes provided by Flash Builder include a set of Spark themes and a set of Halo themes. The default theme for Flex 4 components is Spark. Halo is the default theme for Flex 3.

For more information on theme support in Flex see About themes.

Specifying a theme

Specify themes on a project basis. After specifying a theme for a project, all applications in the project share the same theme.

- 1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:
 - (Design View) Select the Appearance Panel. Then select the Current Theme.
 - (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme
- 2 Select a theme and then click OK.

Importing themes

You can use Flash Builder to import themes. The files for a theme must be enclosed in a folder. All required files for a Flex theme must be present.

The name of the theme is determined by the name element in the `metadata.xml` file contained within the theme folder. If the name element is not specified, or if `metadata.xml` is not present, then the name of the theme folder becomes the name of the theme.

For more information on the required format for Flex themes, see [“Creating themes”](#) on page 198.

Flash Builder themes can be in the following formats:

- Theme ZIP file

Extract the contents of the ZIP file before importing the theme. The extracted contents should contain all required files.

- CSS or SWC file for a theme

The CSS or SWC file must be in a folder containing all required files for a Flex theme. When you import a theme using Flash Builder, you select either the CSS or SWC file for the theme.

- MXP file

You can use Adobe Extension Manager CS4 to package files for a Flex themes in an MXP file. The theme can then imported into Flash Builder using the Extension Manager.

For more information on packaging themes in an MXP file, see [“Creating an extension file \(MXP file\) for a Flex theme”](#) on page 199.

Importing Flex themes using Flash Builder

- 1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:
 - (Design View) Select the Appearance Panel. Then select the Current Theme.
 - (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme
- 2 Select Import Theme, navigate to the folder containing the theme to import, select the CSS or SWC file, and click OK.

Importing Flex themes packaged in an MXP file

- 1 If you have not already done so, import Adobe Flash® Builder™ 4 into Adobe Extension Manager CS4:
From Adobe Extension Manager, select File > Import Product.
- 2 Open Adobe Extension Manager and select Flash Builder 4.
- 3 Select File > Install Extension, navigate to the MXP file for the theme, and click Open.

After you accept the license, Adobe Extension Manager installs the theme into Flash Builder. The theme is now available in Flash Builder from the Select Project Theme dialog.

Note: You can also double-click the MXP file to invoke Adobe Extension Manager, which then automatically installs the theme.

Downloading themes

You can download themes that can then be imported into Flash Builder.

Downloading Flex themes

- 1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:
 - (Design View) Select the Appearance Panel. Then select the Current Theme.
 - (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme
- 2 Select Find More Themes.

Flash Builder opens your default web browser to a page containing themes to download. You can also navigate to any other site containing themes for Flex that you can download.

- 3 Select a Flex theme to download.

After you download the theme, you can import the theme, as described in “[Importing themes](#)” on page 196.

Creating themes

You can create your own themes and import them into Flash Builder. A Flex theme typically contains the following files:

- SWC, SWF, CSS, PNG, JPEG, and other files that make up your theme.

The files that make up the theme can vary, but must include a SWC or CSS file.

- `preview.jpg` file

The preview image file for the theme. If your theme folder does not contain `preview.jpg`, then Flash Builder uses a default preview image for the theme.

- `metadata.xml` file

Contains information about the theme, including which versions of the SDK the theme is compatible with. If your theme folder does not contain this file, then Flash Builder creates one when importing the theme.

Typically you package a theme in ZIP file, but the ZIP file must be extracted before you can import the theme into Flash Builder. You can also package the theme files in an Adobe Extension Manager file (MXP file), and use Adobe Extension Manager to import the theme into Flash Builder.

For more information, see [About themes](#).

Metadata.xml file

The following table lists the elements that can be included in `metadata.xml`.

Element Name	Description
Name	The name of the theme that appears in Flash Builder. When importing a theme using Flash Builder, you can override the name specified in the <code>metadata.xml</code> file.
Category	Author of the theme. The category under which the theme is displayed in Flash Builder.
sdk	Specifies the Flex SDK versions for which the theme is compatible. This is a parent element for <code>minVersionInclusive</code> and <code>maxVersionExclusive</code> . If the <code>sdk</code> element is absent, then the theme is valid for all SDKs.
minVersionInclusive	Earliest Flex SDK version for which this theme is compatible. If absent, then this theme is compatible with all earlier versions of the Flex SDK.
maxVersionExclusive	Latest SDK version for which this theme is compatible. If absent, then this theme is compatible with all later versions of the Flex SDK.
mainFile	Top-level file for implementing the theme. This file can reference other files in the theme. For example, a CSS file could reference a SWC or SWF file. The -theme compiler argument references the specified file.

The following example shows a typical `metadata.xml` file for a theme created by Company ABC.

```

<theme>
  <name>WindowsLookAlike</name>
  <category>ABC</category>
  <sdk>
    <minVersionInclusive>2.0.1</minVersionInclusive>
    <maxVersionExclusive>4.0.0</maxVersionExclusive>
  </sdk>
  <mainFile>WindowsLookAlike.css</mainFile>
</theme>

```

According to the `metadata.xml` file, the theme is compatible with the Flex 2.0.1 SDK. It is also compatible with SDKs up to, but not including, Flex 4.0.0. When this theme is selected, `WindowsLookAlike.css` is the file that will be added to the `-themes` compiler argument.

Creating an extension file (MXP file) for a Flex theme

You can use Adobe Extension Manager CS4 to create an extension file (MXP file) for a Flex theme. The MXP file can be imported into Flash Builder using Adobe Extension Manager CS4.

Place all your theme files in a staging folder, and then create an extension installation file (MXI file), which is used by Extension Manager to create the MXP file. For information on the format of an MXI file, refer to the [Extension File Format](#) document.

When creating the MXI file, you specify destination paths for each of the theme's files. The destination paths are in this format:

```
$flexbuilder/<Theme Name>
```

- `$flexbuilder` is defined in the Flash Builder configuration file, `XManConfig.xml`. Extension Manager expands `$flexbuilder` according to this definition. `XManConfig.xml` is at the following location on your file system:

```
/<Install Dir>/Flash Builder 4/configuration/XManConfig.xml
```
- `<Theme Name>` is the name of the folder that will contain the Flex theme.

Creating an MXP Extension file for a Flex theme

- 1 Place all the files for the theme, including the MXI file, in a staging folder.
- 2 From the Extension Manager, select File > Package Extension.
- 3 Navigate to the extension installation file and select it.
- 4 Navigate to a location for the package file, and name it using the extension `.mxp`.

You can then test the extension file by installing it using the Extension Manager.

Adding additional themes

You can specify more than one theme file to be applied to an application. If there are no overlapping styles, both themes are applied completely. There are other considerations when adding additional themes, such as the ordering of the theme files.

To add additional themes, use the command line compiler, `mxmlec` with the `theme` compiler option to specify the path to the theme files.

Using themes provides details on specifying compiler arguments and the ordering of theme files.

Applying styles

Styles affect the appearance of an application by specifying values for visual parameters for application components. You can set styles that apply to all components in an application, to individual components, or to a set of components referenced by a style selector.

For example, you can specify styles such as:

- Text
Font family, size, weight, color, font (bold, italic, underline) and other text display settings
- Border
Thickness, color, rollover color, border-style (solid, inset, outset, none), corner radius, and others
- Color
Fill color and alpha

Note: *The styles available for a component varies, according to the component.*

You set style properties inline on an MXML tag or separately using CSS code. The CSS code can be inside `<fx:Style>` tags in an application or in a separate CSS file.

When you apply inline styles to components, you can convert component styles into a CSS rule in an external stylesheet. You can use the CSS editor to edit CSS files.

You can also convert skins applied to a component into styles.

Use Design mode of the MXML editor to apply styles to an application or specific application components. You can also use Design mode to convert applied styles or skins to CSS stylesheets.

Styles compared to skins

Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of bitmap images, SWF files, or class files that contain drawing methods that define vector images. Skins can define the appearance of a component in various states. For example, you can specify the appearance of a Button component in the up, down, over, and disabled states.

Using Flash Builder, you can convert a component's skin to CSS styles.

Apply styles to an application

Use the Appearance view to define styles that apply to an entire application. Flash Builder saves the style defined in Appearance view as a global CSS style selector.

- 1 In Design mode of the MXML editor, open an MXML application file that contains several components.
- 2 In Appearance view, specify global styles for the application.

If Appearance view is not visible, select Window > Other Views > Flash Builder > Appearance.

After you apply the styles, Flash Builder creates a global CSS style selector for the specified style.

If this is the first style referenced in the application, Flash Builder creates a new CSS file and references it in the MXML application file.

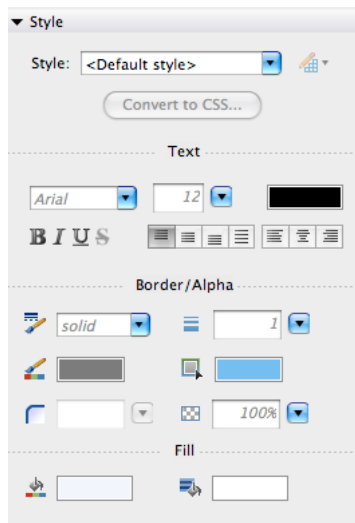
If your application already references a CSS file or a `<fx:Style>` block, Flash Builder updates the CSS with the global style selector.

Apply inline styles to a component

Use the Properties view to define inline styles for selected components.

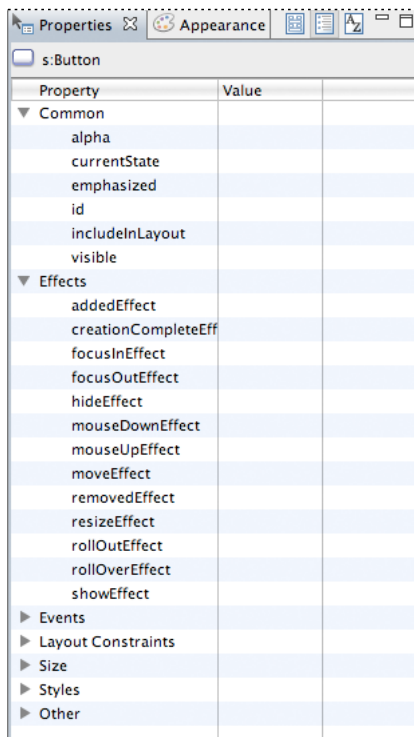
- 1 In Design mode of the MXML editor, open an MXML application file that contains several components.
- 2 Select a component and specify style property values in the Style area of Properties view.

The Style area changes, depending on the component you select.



Properties view showing styles for a DataGrid

- 3 In Properties view, select Category view to list all the styles that can be applied to the selected component.



Note: Multi-word style names in Flex can be written either like an ActionScript identifier (for example, `fontFamily`) or like similar HTML styles (for example, `font-family`).

- 4 After specifying a style, switch to Source mode to view the generated inline code that applies the style.

Apply an external or embedded style to an application

You can embed CSS styles in an MXML application file or reference an external CSS file. The following example shows CSS code to apply styles to all Spark Button components in an application. It also creates a selector, `.myStyle`, that can be applied to any component:

```
@namespace s "library://ns.adobe.com/flex/spark";
@namespace mx "library://ns.adobe.com/flex/mx";

s|Button { fontSize: 16pt; color: Red } /* type selector */
.myStyle { color: Red } /* class selector */
```

For styles applied to components, such as `s|Button`, the selector for components must specify a namespace. In this example, `s|Button` defines a style that is automatically applied to all Spark Button components.

Use the CSS period notation to create a selector that can be applied to any component. In this example, `.myStyle` does not have to declare a namespace and can be applied to any component.

Flex has specific requirements for creating and applying styles. For details, see [Using styles in Flex](#).

Apply styles to an application

- 1 In Design mode of the MXML editor, create an MXML application file that contains several Spark Button components and a CheckBox component.
- 2 (Embedded styles) In Source mode, add the following code to your application:

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    @namespace mx "library://ns.adobe.com/flex/haio";

    s|Button { fontSize: 16pt; color: Red } /* type selector */
    .myStyle { fontSize: 16pt; color: Blue } /* class selector */
</fx:Style>
```

- 3 (External stylesheet) Create a stylesheet that implements the `s|Button` and `.myStyle` selectors in Step 1. In Source mode, reference the file from the MXML application file:

```
<fx:Style source="styles.css"/>
```

- 4 Switch to Design mode. Notice that the Spark Buttons now have the style applied.
- 5 Select the Check Box component and in Properties view, select Styles > `myStyle`.
Now the CheckBox component has `.myStyle` style applied.

Convert to CSS

You can convert inline styles and component skins to CSS styles. During the conversion, you can specify whether to make the styles global or to apply to a specific component. You can also specify a CSS style selector for the generated styles.

- 1 In Design mode of the MXML editor, select a component in the design area. The component should use inline styles or specify a `skinClass` property.

See “[Generating and editing skins for Spark components](#)” on page 205 for information on specifying a `skinClass` for a component.

- 2 In Properties view, click Convert to CSS.

- 3 If you have multiple projects open in your workspace, select/deselect the resource files you want to save in the Save Resources dialog. Then click OK.

- 4 In the New Style Rule dialog box, select the .css file or click New to create a new file.

- 5 Specify the Selector type. Choose from the following:

- All Components

The style is global and applies to all components in the application.

- All Components With Style Name

Components specify this style selector using the `styleName` attribute. If you choose this option, then specify a name for the Selector.

- Specific Component

The style applies only to the selected component.

- Specific component With Style Name

The style applies only to the selected component, and references the style by the name of the type selector. If you choose this option, then specify a name for the type selector.

- 6 Click OK.

Flash Builder generates or updates the specified CSS file. Flash Builder also modifies the source code in the application to reference the type selector in the CSS file.

Flash Builder removes references to the inline style or the `skinClass` property for the component.

Edit style rule

When external CSS styles are already applied to a component, you can quickly jump from the component to edit these styles.

- 1 Select a component.

- 2 In the Properties View, click the Edit Style Rule button next to the Style pop-up menu, then select the style you want to edit.

The CSS file opens in the CSS editor’s Design mode. You use the Properties View to make further changes. You can also modify your CSS in Source mode.

Creating CSS files

Use the New CSS File wizard to create CSS files for a project. The New CSS File wizard creates a blank file that you can use to define your CSS styles.

By default, Flash Builder adds the default namespaces for Spark and MX styles.

To create a blank CSS file:

- 1 From the Flash Builder menu, select File > New CSS File.

- 2 Specify a source folder.

The source folder can be in the current project or another project.

- 3 Specify a package for the file

Select from the packages available in the project. If you want to place the file in a new package, then first create the package. Select File > New Package.

- 4 Specify a Name for the file.

- 5 Click Finish.

Flash Builder uses templates that define the contents of newly created files. You can customize the templates that Flash Builder uses. See [“Customizing File Templates”](#) on page 120.

Using the CSS editor

Flash Builder provides a CSS editor which you can use to create and edit style sheets for your application. The CSS editor is available only in Source mode.

When you create a new style sheet in Flash Builder, Flash Builder provides the following declarations for the Spark and MX namespaces:

```
@namespace s "library://ns.adobe.com/flex/spark";  
@namespace mx "library://ns.adobe.com/flex/mx";
```

Some Spark and MX components share the same local name. For example, there is a Spark Button component (in the spark.components.* package) and a MX Button component (in the mx.controls.* package). To distinguish between different components that share the same name, you specify namespaces in your CSS that apply to types.

If you do not use type selectors in your style sheets, then you are not required to declare namespaces. For more information, including examples, see [About namespaces in CSS](#).

Note: Styles are handled differently for applications that use the Flex 3 SDK. If you are working in a Flex project that specifies the Flex 3 SDK, the CSS editor reverts to behavior implemented with Flex Builder 3. Refer to the [Flex Builder 3 documentation](#) for information on using the CSS editor for applications that use the Flex 3 SDK.

Modifying user interfaces using skins

Skin classes modify the appearance of controls in a user interface. The way you create, edit, and import skins differs for Spark components and MX components.

About Spark skins

Spark skins control all visual elements of a component, including layout. Spark skins can contain multiple elements, such as graphic elements, text, images, and transitions. Spark skins support states. You can use a skin to define the appearance of a component for each of the component's states. Skins typically specify minimum sizing requirements for the component. For details on how Spark skins are implemented in Flex, see [About Spark skinning](#).

You can use Flash Builder to generate and edit skins for a Spark component. When Flash Builder generates a skin, it creates a skin class in MXML. You can modify the appearance defined by the skin in the MXML editor. Some changes can be made in Design mode of the editor, while others require you to edit the MXML file in Source mode. See [“Generating and editing skins for Spark components”](#) on page 205.

About skins for MX components

Skins for MX components can be either a bitmap graphic or a vector graphic. A bitmap graphic, called a graphical skin, is made up of individual pixels that together form an image. A vector graphic, called a programmatic skin, consists of a set of line definitions that specify a line's starting and end point, thickness, color, and other information required by Adobe® Flash® Player to draw the line. For details on how skins are implemented for MX components in Flex, see [About MX component skinning](#).

You can use Flash Builder to import skin artwork for MX components. See [“Importing skin artwork for MX components”](#) on page 209.

The `mx.skins.spark` package defines Spark skins for MX components.

Generating and editing skins for Spark components

You can use Flash Builder to generate and edit skins for Spark components. When you generate a skin, Flash Builder uses a skin from the theme for a project. The default theme for a project is Spark. You can change the theme for a project from the Appearance view. See [“Applying themes”](#) on page 196.

When generating a skin for a component, Flash Builder creates an MXML file that implements the Skin class for the component. You can specify whether to generate the Skin class as a copy of an existing skin or generate a blank Skin class file.

Use a combination of the MXML editor in Design and Source mode to edit the skin. In Design mode, use the Outline view to select elements of the skin to edit. Use States view to navigate between the states of a component. Some parts of a skin cannot be edited in Design mode. Use Source mode to edit parts of skin that are not available in Design mode.

Some components contain subcomponents. For example, an HSlider component contains Button components that define the thumb and track of the HSlider. Subcomponents can only be skinned in Source mode.

Component states, skin parts, and host components

Skins define the appearance of a component for each state of the component. For example, a Spark Button has four states, up, over, down, and disabled. When you generate a skin for a Spark Button, you can specify the appearance for each of these states.

Each component has parts that can be styled. For example, for a Button component, you can style the Button fill color, text attributes for the Button label, and the Rect components that make up the Button's border.

When using Flash Builder to create skins for a component, you specify a host component upon which the generated skin is based. By default, the host component is the base class of the component you are skinning. However, you can select a different host component.

Note: Specifying a host component for a skin class is required when generating skin classes using Flash Builder. However, if creating skin classes directly in source code, a host component is not required.

Skinning contract between a skin and its host component

The skinning contract between a skin class and a component class defines the rules that each member must follow so that they can communicate with one another.

The skin class must declare skin states and define the appearance of skin parts. Skin classes also usually specify the host component, and sometimes bind to data defined on the host component.

The component class declares which skin class it uses. It must also identify skin states and skin parts with metadata. If the skin class binds to data on the host component, the host component must define that data.

The following table shows these rules of the skinning contract:

	Skin Class	Host Component	Required?
Host component	<code><fx:Metadata> [HostComponent("spark.components.Button")] </fx:Metadata></code>	n/a	No
Skin states	<code><s:states> <s:State name="up"/> </s:states></code>	<code>[SkinStates("up")]; public class Button { ... }</code>	Yes
Skin parts	<code><s:Button id="upButton"/></code>	<code>[SkinPart(required="false")] public var upButton Button;</code>	Yes
Data	<code>text="{hostComponent.title}"</code>	<code>[Bindable] public var title:String;</code>	No

Skin declaration

In Flash Builder, a Skin declaration is the file that implements the skin for a component. Flex defines a skin declaration for each visual component. When you generate a new skin for a component, Flash Builder generates the Skin declaration.

You can view the Skin declaration for selected components:

- 1 In Design mode of the MXML editor, select a Spark component in the design area.
- 2 From the context menu for the component, select Open Skin Declaration.

The class implementing the skin opens in Source mode of the editor.

If the class is one that you created, you can edit the file.

You can also do the following to open a skin declaration file:

- With the component selected, in the Style section of the Properties view click the icon near the Skin field.
- In Source mode, with a Spark component selected, from the Flash Builder menu, select Navigate > Open Skin Declaration.

Generate and edit a skin for a Spark Button (Design mode)

This example generates a Skin class for a Spark Button, showing you how to use a combination of Flash Builder views to edit the skin. It assumes you are working in a Flex project using the default Spark theme.

- 1 Create an application file. In Design mode of the editor, add a Spark Button to the application.

- 2 From the context menu for the Button, select Create Skin.

The New MXML Skin dialog opens.

You can also do the following to open the New MXML Skin dialog.

- In the Style section of the Properties view, select the icon near the Skin field.
- From the Flash builder menu, select New > MXML Skin
- In Design mode of the editor, select Design > Create Skin

- 3 Specify the following in the New MXML Skin Dialog:

- Source Folder and Package for the generated Skin declaration.
- Name

The name for the Skin class you are creating.

- Host Component

To change the default component, click Browse and select a host component.

- (Recommended) Select Create As Copy Of and do not remove ActionScript styling code

If you are new to skinning, use a copy to get started creating a Skin class. Modify the ActionScript styling code.

- (Advanced Users) Do either of the following if you are familiar with creating Skin classes:

Remove ActionScript styling code or do not create a copy of an existing class.

If you do not create a copy of an existing class, Flash Builder generates a blank Skin class file with some comments to guide you.

The remaining steps of this procedure assume that you followed the Recommended option for generating a Skin class.

- 4 Click Finish.

Flash Builder generates a Skin class file and opens it in Design mode of the MXML editor.

The Button component is selected.

The up state of the Button is selected in the States View.

- 5 For each state of the Button, modify the Text styles, Content Background styles, and Color styles.

Use the editing tools in Style section of Properties view to make your changes.

- 6 Open Outline view:

Notice that the top-level node, SparkSkin, is selected.

- 7 In Outline view, select Rect shadow to define styles for the Button's shadow.

Notice that the Style section tools are not available.

- 8 Switch to Source mode of the editor.

Flash Builder highlights the Rect component that defines the Button's shadow. Make any changes for the Button's shadow.

- 9 Save the Skin class file and your application file.

In Design mode of the MXML editor you can view the skin for the button, assuming that you followed the recommended option in Step 3. If the styles do not show, Select the Refresh icon for the design area.

Notice that the application file added a reference to the Skin class that you created.

10 Run the application to see how the skin changes for Up, Over, and Down states of the Button.

Creating and editing skins for Spark components (Source mode)

You can open the New MXML Skin dialog directly in Source mode of the editor. For example, do the following to create a skinClass for a Spark Button component.

- 1 In Source mode of the editor, place your cursor inside a `<s:Button>` tag and type the following:

```
<s:Button skinClass="
```

After you type the first quote for the skinClass name, a context menu appears.

- 2 With Create Skin highlighted in the code hints, the Enter key opens the New MXML Skin dialog.

This dialog is the same dialog that opens in Design Mode.

See the instructions in [“Generate and edit a skin for a Spark Button \(Design mode\)”](#) on page 206 for creating the skinClass.

- 3 Click Finish.

Flash Builder generates a new skinClass based on your selections in the New MXML Skin dialog. The editor switches to the source for the newly generated class.

- 4 Edit the skinClass.

Save your class file and application file.

Note: You can convert the generated skin class to CSS to view the styles that are applied. See [“Converting a skin to a CSS style”](#) on page 208.

Converting a skin to a CSS style

Using Flash Builder, you can convert a skin for a component into a CSS style. The advantage of converting the skin to style is to use the style as a type selector for all components of that class. Otherwise, set the skinClass property for each component.

The following procedure shows how to convert a skin for a Spark Button into a CSS style.

- 1 Generate and edit a skin for a Button component.
- 2 In Design mode of the editor, select the button. In the Styles section of the Properties view, click Convert to CSS.
- 3 In the New Style Rule dialog, select or create a CSS file for the style.

If you do not have a CSS file in the project that you want to use, click New to create a file.

- 4 Specify the Selector Type. Choose from the following:

- All Components

The style applies to all components in the application.

- All Components With Style Name

Components specify this style selector by name. If you choose this option, then specify a name for the type selector.

- Specific Component

The style applies only to the selected component.

- Specific component With Style Name

The style applies only to the selected component, and references the style by the name of the type selector. If you choose this option, then specify a name for the type selector.

- 5 After specifying a Selector Type, click OK.

Flash Builder generates or updates the specified CSS file. Flash Builder also modifies the source code in the application to reference the type selector in the CSS file.

Flash Builder removes references to the `skinClass` property for the component.

Importing skin artwork for MX components

You use the Import Skin Artwork wizard to import both vector graphics artwork and bitmap artwork from the CS4 versions of Flash Professional, Fireworks, Illustrator, and Photoshop. (For bitmap artwork, any .PNG, .JPG, or .GIF can be used). The artwork can then be used as skins for Flex components.

Note: Adobe provides a set of skinning templates to make it easy to create skins for the built-in Flex components. Use the templates with Flash, Fireworks, Illustrator, or Photoshop to create the artwork. You can also use Flash to create fully functional custom Flex components. For more information, see the articles in [Importing Flash Professional Assets into Flex](#).

- 1 Select File > Import > Skin Artwork.

In the plugin version, select File > Import > Artwork.

- 2 In the Import Skin Artwork dialog box:

- Choose a folder of bitmaps or a SWC or SWF file to import skins from, or click Browse to locate one. Supported file types include the following:
 - AS3 SWF and AS3 SWC files created in Adobe Flash Professional CS5
 - Vector graphic files created in Adobe Illustrator® and exported as SWF files for Flash Player 8
 - Bitmap graphic files in PNG, GIF, and JPG formats
- Choose a folder to import the skins to. The folder must be a source folder for a Flex project (or you can specify a subfolder in the source folder). The default selection is the folder for the Flex project currently open.
- In the Copy Artwork To Subfolder field, the default folder name is based on the folder or assets being imported. Click Browse to choose a different location.
- In the Create Skin Style Rules In field, specify a name for a CSS file that will contain the style rules. The default name is based on the name of the artwork folder or Flash file being imported.
- Click the Delete All Existing Rules In File check box if you want the specified CSS file to be overwritten upon importing (as opposed to importing skins and keeping other existing definitions in the CSS file). The box is unchecked by default, and if the CSS file does not exist it is disabled.
- In the Apply Styles To Application field, the default is the selected file in the Flex Navigator or active editor view, or the main application file for the project.
- Click Next.

- 3 In the next Import Skin Artwork dialog box, select the skins you want to import and specify which CSS style type and skin part property will be used. You can check items one at a time or click Check All or Uncheck All.

- If items do not have a valid style or skin part property name, they will not be checked by default. The following examples show the naming convention used in Flash Builder:
 - Button_upSkin

- Button_glow_downSkin (maps to downSkin property of Button.glow style rule)
- TabBar-tab_upSkin (upSkin property maps to tabStyleName property of TabBar style rule)
- MyCustomComponent_borderSkin

For custom components, the item will be checked if the component has been defined somewhere within the project you are importing to.

- If necessary choose a style and skin part for the pop-up menus in each column.
- Click Finish.

A CSS file is created and displayed in the Source view. The CSS file will be attached to the application specified in the wizard. If you import a SWC file, it is automatically added to the library path for the project.

Generating custom item renderers

Spark list-based controls, such as List and ComboBox, support custom item renderers. You can also use Spark item renderers with some MX controls, such as the MX DataGrid and MX Tree controls.

Use custom item renderers to control the display of a data item in a DataGroup, SkinnableDataContainer, or in a subclass of those containers. The appearance defined by an item renderer can include the font, background color, border, and any other visual aspects of the data item. An item renderer also defines the appearance of a data item when the user interacts with it. For example, the item renderer can display the data item one way when the user moves the mouse over the data item. It displays the differently when the user selects the data item by clicking on it.

Using Flash Builder, you can generate and edit item renderers. When Flash Builder generates item renderers, it uses one of the following templates:

- Spark components

Use this template for Spark list-based controls, such as List and ComboBox.

- MX Advanced DataGrid
- MX DataGrid
- MX Tree

You can open the New MXML Item Renderer wizard from both Design mode and Source mode of the MXML editor. In the New MXML Item Renderer wizard, you specify a name and template for the item renderer. Flash Builder generates an MXML file that implements the item renderer.

Components in the application reference the generated item renderer using the `itemRenderer` property.

For details on creating and using item renderers, see Custom Spark Item Renderers.

Generate and edit an item renderer for an MX Tree component (Design mode)

This example generates an item renderer for an MX Tree component, showing you how to use a combination of Flash Builder views to edit the item renderer. It assumes that you are working in a Flex project using the default Spark theme.

- 1 Create an application file. In Design mode of the editor, add an MX Tree component to the application.

Populate your Tree with data that can be displayed when you run the application.

- 2 From the context menu for the Tree, select Create Item Renderer.

The New MXML Item Renderer dialog opens.

You can also do the following to open the New MXML Item Renderer dialog.

- In the Common section of the Properties view, select the icon near the Item Renderer Field field.
- From the Flash builder menu, select New > MXML Item Renderer.

3 Specify the following in the New MXML Item Renderer Dialog:

- Source Folder and Package for the generated item renderer declaration.
- Name

The name for the item renderer class you are creating.

- Template

Select the template to use when generating the item renderer.

4 Click Finish.

Flash Builder generates an ItemRenderer class file and opens it in Design mode of the MXML editor.

The ItemRenderer component is selected.

The normal state of the Tree is selected in States view.

5 For each state of the Tree, modify the appearance in the generated ItemRenderer class.

a Open Outline view:

Notice that the top-level node, MXTreeItemRenderer, is selected.

In the Style section of Properties view, modify the appearance of tree items.

b In Outline view, select other components of the MXTreeItemRenderer to modify the appearance of those components.

Notice that the Style section tools are not always available.

If the Style section tools are not available, then use Source mode of the editor to define the appearance. When you switch to Source mode, the source for the selected component in Outline view is highlighted.

6 Run the application to see how the ItemRenderer changes the appearance of the Tree.

Creating and editing item renderers (Source mode)

You can open the New MXML Item Renderer dialog directly in Source mode of the editor. For example, do the following to create an item renderer for a Spark List component.

1 In Source mode of the editor, place your cursor inside a `<s:List>` tag and type the following:

```
<s:List itemRender="
```

After you type the first quote for the item renderer class name, a context menu appears.

2 Double-click Create Item Renderer to open the New MXML Item Renderer dialog.

This dialog is the same dialog that opens in Design Mode.

See the instructions in [“Generate and edit an item renderer for an MX Tree component \(Design mode\)”](#) on page 210 for creating the item renderer.

3 Click Finish.

Flash Builder generates a new item renderer based on your selections in the New MXML Item Renderer dialog. The editor switches to the source for the newly generated class.

4 Edit the item renderer class.

Save your class file and application file.

ItemRenderer declaration

In Flash Builder, an ItemRenderer declaration is the file that implements the custom ItemRenderer for a component.

You can view the custom ItemRenderer declaration for selected components:

- 1 In Design mode of the MXML editor, select a component that you have implemented a custom item renderer for.
- 2 From the context menu for the component, select Open Item Renderer Declaration.

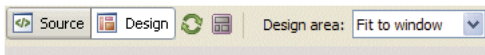
The class implementing the item renderer opens in Source mode of the editor. You can also do the following to open the item renderer declaration:

- Select the component in Design mode. In the Common section of the Properties view, select the icon near the Item Renderer field.
- In Source mode, with the component selected, from the Flash Builder menu, select Navigate > Open Skin Declaration.

Refreshing Design mode to render properly

If necessary, you can refresh the MXML and CSS editors' Design mode to render your layout properly. The rendering of your layout can become out of date in certain situations. This can happen, for example, if you modify a visual element in a dependent Flash component (SWC). Styles and skins may also not be rendered properly because Flash Builder needs to rebuild the file.

- ❖ Click the Refresh button in the editor toolbar.



Adding View States and Transitions

You can use Adobe® Flash® Builder™ to create applications that change their appearance depending on tasks performed by the user. For example, the base state of the application could be the home page and include a logo, sidebar, and welcome content. When the user clicks a button in the sidebar, the application dynamically changes its appearance (its *state*), replacing the main content area with a purchase order form but leaving the logo and sidebar in place.

In Flex, you can add this kind of interaction with view states and transitions. A *view state* is one of several views that you define for an application or a custom component. A *transition* is one or more effects grouped together to play when a view state changes. The purpose of a transition is to smooth the visual change from one state to the next.

About view states and transitions

A *view state* is one of several layouts that you define for a single MXML application or component. You create an application or component that switches from one view state to another, depending on the user's actions. You can use view states to build a user interface that the user can customize or that progressively reveals more information as the user completes specific tasks.

Each application or component defined in an MXML file always has at least one state, the *base state*, which is represented by the default layout of the file. You can use a base state as a repository for content such as navigation bars or logos shared by all the views in an application or component to maintain a consistent look and feel.

You create a view state by modifying the layout of an existing state or by creating a completely new layout. Modifications to an existing state can include editing, moving, adding, or removing components. The new layout is what users see when they switch states.

For a full conceptual overview of view states, including examples, see [View states](#).

Generally, you do not add pages to a Flex application as you do in an HTML-based application. You create a single MXML application file and then add different layouts that can be switched when the application runs. While you can use view states for these layouts, you can also use the ViewStack navigator container with other navigator containers.

When you change the view states in your application, the appearance of the user interface also changes. By default, the components appear to jump from one view state to the next. You can eliminate this abruptness by using transitions.

A *transition* is one or more visual effects that play sequentially or simultaneously when a change in view state occurs. For example, suppose you want to resize a component to make room for a new component when the application changes from one state to another. You can define a transition that gradually minimizes the first component while a new component slowly appears on the screen.

Support for Flex 3 view states

Flash Builder provides support for view states as implemented in Flex 3. If you create a project that uses the Flex 3 SDK, the MXML editor in both Design and Source mode reverts to the Flex Builder 3 implementation. For information on editing states for the Flex 3 SDK, refer to the [Flex Builder 3 documentation](#).

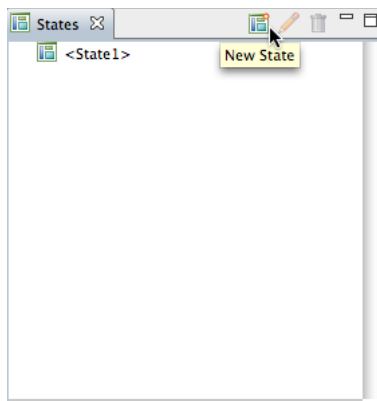
Creating a view state

By default, an application has a single view state, which you typically use as the base state. Use the Flash Builder States View to add additional states and to edit the layout and components for each state.

- 1 Using the layout tools in Flash Builder, design the layout of the base state of your application or component.

For more information, see [“Building a user interface with Flash Builder”](#) on page 177.

- 2 In the States view (Window > Other Views > Flash Builder > States), click the New State button in the toolbar.



The New State dialog box appears.

- 3 Enter a name for the new state.

- 4 Specify whether to create a state that is a duplicate of an existing state or to create a new, blank state. Click OK.
- 5 Use the layout tools in Flash Builder to modify the appearance of the state.
You can edit, move, add, or delete components. As you make changes, the changes defining the new state become part of the MXML code.
- 6 Define an event handler that lets the user switch to the new state.
For more information, see [“Switching view states at run time”](#) on page 214.

Setting a non-base state as the starting view state

By default, an application displays the base state when it starts. However, you can set another view state to be the state that displays when the application starts.

- 1 In States View (Window > States), double-click the view state that you want to use as the starting state.
- 2 In the Edit State Properties dialog box that appears, select the Set As Start State option and click OK.

Setting the view state of a component

If your application has multiple states, you can set the view state of a single component.

- 1 In Design View of the MXML editor, select a component in your layout.
- 2 In Properties View, use the In States field to select the states in which the component is visible.

Switching view states at run time

When your application is running, users need to switch from one view state to another. You can define event handlers for user controls so that users can switch states at run time.

The simplest method is to assign the `currentState` property to the click event of a control such as a button or a link. The `currentState` property takes the name of the view state you want to display when the click event occurs. In the code, you specify the `currentState` property as follows:

```
click="currentState='viewstatename' "
```

If the view state is defined for a specific component, you must also specify the component name, as follows:

```
click="currentState='componentID.viewstatename' "
```

For more information, see [Using View States](#).

- 1 Ensure that the initial state has a clickable control, such as a Button control.
In Design mode of the MXML editor, select the control and enter the following value in the On Click field in the Properties view:

```
currentState='viewstatename'
```


viewstatename is the name for the state.
- 2 If you want to switch to the base state, enter:

```
currentState= ''
```


`''` is an empty string, represented by two single quotes.
- 3 To test that the states switch correctly in the application when the button is clicked, click the Run button in the Flash Builder toolbar.

You can define a transition so that the change between view states is smoother visually. For more information, see [“Creating a transition”](#) on page 217.

Creating view state groups

Flex provides support for view state groups. The `stateGroups` attribute of the `<States>` tag lets you group one or more states together. For example, if multiple components appear in the same set of view states, you can create a view state group that contains all these view states. Then, when you set the `currentState` property to any view state in the group, the components appears. For more information, with examples, see [Defining view state groups](#).

Design mode of the MXML editor does not support editing state groups. Use Source mode to create and edit state groups. Source mode provides code hinting and a Show State pop-up menu to assist you in creating and editing state groups.

If you create view state group, be careful using Design View. If you delete a state using Design View, you can inadvertently leave a reference to a deleted component in a state group.

Deleting a view state

You can delete view states from an application using Design View of the MXML editor. However, if you have created a state group then use Source View to delete a state. This avoids inadvertently leaving a reference to a component in a deleted state.

- 1 In the Design View of the MXML editor, select the view state that you want to delete from the States View (Window > States).
- 2 Click the Delete State button on the States View toolbar.

Working with multiple states in an application

If you have an application that contains more than one state, Design mode of the MXML editor allows you to switch the view for each state, displaying only the components that defined for a specific state. For each component, you can specify the state in which it is visible.

Edit the component of a specific state

- 1 In Design View of the source editor, use the States View to add one or more additional states to an application.
- 2 Use the State drop-down menu to switch the view to the selected state.
- 3 Add, move, delete, or modify the components in the state.


Changes you make to a specific state do not appear in other states unless you specify that the component appears in more than one state.

Specify that a component appears in multiple states

- 1 In Design View of the source editor, use the States View to add one or more additional states to an application.
- 2 For any component in a state, select the component.
- 3 In the Properties View, select which states the component appears.

You can specify that the component appear in all states, or select one or more states in which the component appears.

If you specify a specific state for a component, the component does not display in the editor when editing another state.

 *Be careful when editing applications that contain multiple states. Components might seem to “disappear” when you switch the editor to a state that doesn’t contain a component visible in another state.*

Creating and editing view states in source code

Source mode of the MXML editor contains several features to help you edit source code for view states.

When an application declares view states, the MXML editor provides a Show State pop-up menu. When you select a specific view state in the Show State menu, components that do not appear in that state are de-emphasized in the editor.

The `includeIn` and `excludeFrom` properties for MXML components specify the view state or state group in which a component appears. Code hinting in the MXML editor assists you in selecting a view state or state group for these properties.

You can also use dot notation with component attributes to specify a view state in which the attribute applies. For example, if you want a Button component to appear in two view states, but also have the label change according to the view state, use the dot operator with the `label` property. Code hinting in the MXML editor assists you in selecting the view state. For example:

```
<s:Button label.State1="Button in State 1" label.State2="Same Button in State 2">
```

Example working with view states in source code

- 1 Create an application that contains more than one view state.

In Source mode of the MXML editor, add the following code after the `<s:Application>` tag.

```
<s:states>
    <s:State name="State1" />
    <s:State name="State2" />
    <s:State name="State3" />
</s:states>
```

Notice that the MXML editor adds a Show State pop-up menu after you define states in the application.

- 2 In Source mode, add the following Button components:

```
<s:Button includeIn="State1" label="Show State 2"
    click="currentState='State2'" />
<s:Button includeIn="State2" label="Show State 3"
    click="currentState='State3'" />
<s:Button includeIn="State3" label="Show State 1"
    click="currentState='State1'" />

<s:Button
    label.State1="All States: State 1 Label"
    label.State2="All States: State 2 Label"
    label.State3="All States: State 3 Label"
    x="0" y="30"/>
```

By default, the editor displays code for all states.

Note: *The click event handlers for the first three buttons cycle through the view states.*

- 3 Still in Source mode, select different view states from the Show State pop-up menu.

For components that are not visible in the selected state, the editor displays the code as light grey.

All the code is editable, but de-emphasizing components that do not appear in the selected view state assists in maintaining code for each view state.

- 4 Switch to Design mode for the MXML editor.

Using either the States View or the States pop-up menu, select different view states. The editor displays the components according to properties defined for the selected view state.

- 5 Run the application. Click the top button to cycle through the view states.

For more information on creating and editing states in source code, see [Create and apply view states](#).

Creating a transition

When you change the view states in your application, the components appear to jump from one view state to the next. You can make the change visually smoother for users by using transitions. A transition is one or more effects grouped together to play when a view state changes. For example, you can define a transition that uses a Resize effect to gradually minimize a component in the original view state, and a Fade effect to gradually display a component in the new view state.

- 1 Make sure you create at least one view state in addition to the base state.
- 2 In Source View of the MXML editor, define a Transition object by writing a `<s:transitions>` tag and then a `<s:Transition>` child tag, as shown in the following example:

```
<s:transitions>
    <mx:Transition id="myTransition">
    </mx:Transition>
</s:transitions>
```

To define multiple transitions, insert additional `<s:Transition>` child tags in the `<s:transitions>` tag.

- 3 In the `<s:Transition>` tag, define the change in view state that triggers the transition by setting the tag's `fromState` and `toState` properties, as in the following example (in bold):

```
<s:transitions>
    <mx:Transition id="myTransition" fromState="*" toState="checkout">
    </mx:Transition>
</s:transitions>
```

In the example, you specify that you want the transition to be performed when the application changes from any view state (`fromState="*"`) to the view state called `checkout` (`toState="checkout"`). The value `"*"` is a wildcard character specifying any view state.

- 4 In the `<mx:Transition>` tag, specify whether you want the effects to play in parallel or in sequence by writing a `<mx:Parallel>` or `<mx:Sequence>` child tag, as in the following example (in bold):

```
<mx:Transition id="myTransition" fromState="*" toState="checkout">
    <b><mx:Parallel>
    </mx:Parallel>
</mx:Transition>
```

If you want the effects to play simultaneously, use the `<mx:Parallel>` tag. If you want them to play one after the other, use the `<mx:Sequence>` tag.

- 5 In the `<mx:Parallel>` or `<mx:Sequence>` tag, specify the targeted component or components for the transition by setting the property called `target` (for one target component) or `targets` (for more than one target component) to the ID of the target component or components, as shown in the following example:

```
<mx:Parallel targets="{ [myVGroup1, myVGroup2, myVGroup3] } ">
</mx:Parallel>
```

In this example, three VGroup containers are targeted. The `targets` property takes an array of IDs.

- 6 In the `<mx:Parallel>` or `<mx:Sequence>` tag, specify the effects to play when the view state changes by writing effect child tags, as shown in the following example (in bold):

```
<mx:Parallel targets="{ [myVBox1, myVBox2, myVBox3] }" >
    <mx:Move duration="400"/>
    <mx:Resize duration="400"/>
</mx:Parallel>
```

For a list of possible effects and how to set their properties, see Introduction to effects.

- 7 To test the transition, click the Run button in the Flash Builder toolbar, then switch states after the application starts.

Binding controls to data

When accessing a data service, Flash Builder provides tools to bind data to data controls, such as a DataGrid. Flash Builder creates columns in a DataGrid based on the data returned from the service. Typically, you need to configure the generated DataGrid columns. Flash Builder provides an editor to configure columns of a DataGrid and Advanced DataGrid component.

For more information on binding data to data controls, see Binding service operations to controls.

Configuring DataGrid and AdvancedDataGrid components

You configure DataGrid columns in Design mode of the MXML editor. The following procedure shows how to configure the columns of a DataGrid component that access a data service. Similarly, you can configure the columns of an AdvancedDataGrid component.

Configure DataGrid columns

- 1 In Design mode of the MXML editor, add a DataGrid (or AdvancedDataGrid) control. Bind the control to data returned from a data service.
For information, see Binding service operations to controls.
- 2 Select the DataGrid and then select Configure Columns from the Property Inspector.
You can also select Configure Columns from the DataGrid's context menu.
- 3 In the Configure Columns dialog, use the Add, Delete, Up, and Down buttons to add, remove, or reorder the columns.
- 4 Use the Standard View of the Configure Columns dialog to edit the commonly used properties of a selected column.
 - Data Binding

Select the data field to display in the column. The Bind To Field combo box displays all available fields from the returned data. If the DataGrid is editable, you can select whether data in this column is editable.

If the data to display does not come from a data service, then the Bind to Field is simply a text box. Use the Bind to Field to represent columns defined in the data source. For example, you could specify the names of columns defined in an XMLList. If the specified name does not correspond to a defined data source, the value is ignored and the column remains empty.

Columns that are programmatically added to a DataGrid cannot be configured using the Data Binding features.

- General Properties

Specify the header text and width of the column. Also, whether the column can be resized or sorted.

Specify Width in pixels. The default width is 100 pixels. If the DataGrid's `horizontalScrollPolicy` property is `false`, all visible columns are displayed. To ensure that all visible columns are displayed, the DataGrid does not always honor the specified value for width.

- Text Properties

Specify text formatting styles for the text in the column.

- 5 Use the Advanced View of the Configure Columns dialog to view and edit the settings for all properties of a selected column.

Adding charting components

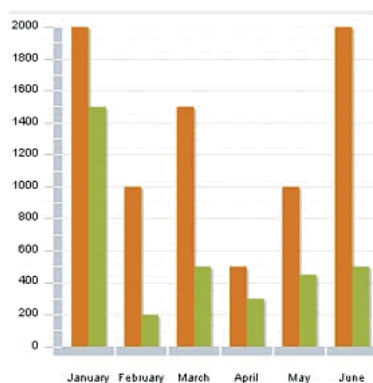
You can use Flash Builder to add charting components to display data in your user interface. The Flex charting components let you create some of the most common chart types, and also give you extensive control over the appearance of your charts. For an overview of the different charts available, see [Chart types](#).

This section describes how to add charting components to your user interface. For information on defining chart data, formatting chart elements, and manipulating other aspects of charts, see [Introduction to Charts](#).

Add a charting component

- 1 In the MXML editor's Design mode, drag a charting component from the Components view into the design area.
- 2 Enter an ID for the chart.
- 3 To display more than one series of data in your chart, click the Add button and enter the new series name in the dialog box that appears.

For example, the following ColumnChart control has two data series. The bar on the left represents the gross profit for six months, and next one is the net profit during the same period.



Remove a data series by selecting it in the list and clicking the Remove button.

- 4 (Optional) Select the Include Legend option.

The Include Legend option lets you add a Legend control to the chart that displays the label for each data series in the chart and a key showing the chart element for the series.

- 5 Click OK to insert the chart.

Adding interactivity with effects

An *effect* is a visible or audible change to the target component that occurs over a time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

Effects are initiated in response to an event, where the event is often initiated by a user action, such as a button click. However, you can initiate effects programmatically or in response to events that are not triggered by the user.

For example, you can create an effect for a TextInput component that causes it to bounce slightly when the user tabs to it, or you can create an effect for a Label component that causes it to fade out when the user passes the mouse over it.

You can define effects in Flash Builder as a property of an MXML component. Use Source view of the MXML editor to implement the effect.

Effects are implemented differently for Spark and MX components. For information on creating effects in MXML and ActionScript code, see Introduction to effects.

Creating an effect for a component

Typically, you define effects in Source mode of the MXML editor. Effects are often invoked from a component's event handler. For example, you can use the click event handler for a Button to invoke an effect. See Applying effects.

However, in Flash Builder you can define an effect for a property of an MXML component.

- 1 In the MXML editor's Design mode, click on a component in the design area.
- 2 Define the effect property for Spark components:
 - a In the Properties view, select the Category View icon.
 - b Select a property in the Effects category and specify an effect.

For example, for a Button's `rollOverEffect` property, you can specify the Fade effect. For a list of available effects, see Available effects.
- 3 Save and run the file to see the effect.

Chapter 10: Working with data in Flash Builder

In Adobe® Flash® Builder™, you interact with data and the data-driven controls directly in your MXML and ActionScript code. You can work with data, automatically generate database applications, generate and use proxy code for web services, and generate and use code that works with the Flex Ajax Bridge. You can also manage Adobe Flash Player data access security issues and use Flash Builder with a proxy service.

About working with data in Flash Builder

You work with data in Flash Builder by directly modifying your MXML and ActionScript application code.

Data-driven controls and containers

Flex provides control and container components from which you build your Flex application user interface. A number of these components present data, which users can select and interact with when using the application. Here are a few examples of how data-driven controls are used:

- On an address form, you can provide a way for users to select their home country (or other typical form input) by using the ComboBox or List controls.
- In a shopping cart application, you can use a Spark List component to present product data that includes images. For the List component you can specify the layout as VerticalLayout, HorizontalLayout, or TileLayout.
- You can provide standard navigation options by using containers such as the TabBar and ButtonBar controls.

You provide data input to all of the data-driven controls with a *data provider*.

For information about using the data-driven controls, see Spark list-based controls.

Data providers and collections

A *collection* object contains a data object, such as an Array or an XMLList object, and provides a set of methods that let you access, sort, filter, and modify the data items in that data object. Several Adobe Flex controls, known as *data provider controls*, have a `dataProvider` property that you populate with a collection.

The following simple example shows how a data provider is defined (as an ActionScript ArrayCollection) and used by a control:

```

<!-- Simple example to demonstrate the Spark ComboBox control -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/halo">

    <fx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var complexDP:ArrayCollection = new ArrayCollection(
                [
                    {ingredient:"Salmon", category:"Meat"},
                    {ingredient:"Potato", category:"Starch"},
                    {ingredient:"Cucumber", category:"Vegetable"},
                    {ingredient:"Steak", category:"Meat"},
                    {ingredient:"Rice", category:"Starch"},
                    {ingredient:"Cumin", category:"Spice"}
                ]
            );

            <!-- Function to handel custom input strings -->
            private function myLabelToItemFunction(input:String):*
            {
                <!-- Returns object that matches items in dataProvider -->
                return {ingredient:input, category:"mystery"};
            }
        ]]>
    </fx:Script>

    <s:Panel title="Spark ComboBox Example" width="75%" height="75%">
        <s:layout>
            <s:VerticalLayout paddingTop="10" paddingLeft="10"/>
        </s:layout>

        <!-- Label that displayed current property values -->
        <s:Label text="Index : {cb.selectedIndex}
                    Item : {cb.selectedItem.ingredient}
                    Type : {cb.selectedItem.category}"/>

        <!-- ComboBox with custom labelToItem function -->
        <s:ComboBox
            id="cb"
            dataProvider="{complexDP}"
            width="150"
            labelToItemFunction="{myLabelToItemFunction}"
            selectedIndex="0"
            labelField="ingredient"/>
    </s:Panel>
</s:Application>

```

For more information about data providers and collections, see [Data providers and collections](#).

Remote data access

Flex contains data access components that are based on a service-oriented architecture (SOA). These components use remote procedure calls to interact with server environments, such as PHP, Adobe ColdFusion®, and Microsoft ASP.NET, to provide data to applications and send data to back-end data sources.

Depending on the type of interfaces you have to a particular server-side application, you can connect to a Flex application by using one of the following methods:

- HTTP GET or POST by using the HTTPService component
- SOAP-compliant web services by using the WebService component
- Adobe Action Message Format (AMF) remoting services by using the RemoteObject component

Note: When you use Flash Builder to develop applications that access server-side data, use a `cross-domain.xml` file or a proxy if data is accessed from a domain other than the domain from which the application was loaded. See [“Managing Flash Player security”](#) on page 229.

You can also use Flash Builder to build applications that use Adobe LiveCycle® Data Services ES, a separate product that provides advanced data service features. LiveCycle Data Services ES provides proxying for remote procedure call (RPC) service applications as well as advanced security configuration. LiveCycle Data Services ES also provides the following data services:

Data Management Service Allows you to create applications that work with distributed data and also to manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships.

Message Service Allows you to create applications that can send messages to and receive messages from other applications, including Flex applications and Java Message Service (JMS) applications.

Flash Builder provides wizards and tools that to connect to data services and bind data service operations to application components. See [Building data-centric applications with Flash Builder](#).

Data binding

In the code example in [“Data providers and collections”](#) on page 221, you may have noticed that the value of the ComboBox control’s `dataProvider` property is `{complexDP}`. This is an example of data binding.

Data binding copies the value of an object (the source) to another object (the destination). After an object is bound to another object, changes made to the source are automatically reflected in the destination.

The following example binds the text property of a TextInput control (the source) to the text property of a Label control (the destination), so that text entered in the TextInput control is displayed by the Label control:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <s:TextInput id="LNameInput" x="10" y="10"/>
  <s:Label text="{LNameInput.text}" x="10" y="50"/>
</s:Application>
```

To bind data from one object to another, you use either the curly braces (`{ }`) syntax (as shown in the example) or the `<fx:Binding>` tag. For more information, see [Using data binding with data models and Data binding in *Using Adobe Flex 4*](#).

Data models

A *data model* is an object that you can use to temporarily store data in memory so that it can be easily manipulated. You can define a data model in ActionScript, in MXML by using a tag such as `<fx:Model>`, or by using any object that contains properties. As an example, the following data model shows information such as a person's name, age, and phone number:

```
<fx:Declarations>
  <fx:Model id="reg">
    <registration>
      <name>{nme.text}</name>
      <email>{email.text}</email>
      <phone>{phone.text}</phone>
      <zip>{zip.text}</zip>
      <ssn>{ssn.text}</ssn>
    </registration>
  </fx:Model>
</fx:Declarations>
```

The fields of a data model can contain static data (as in the example), or you can use data binding to pass data to and from the data model.

You can also define the data model within an XML file. You then reference the XML file through the file system or through a URL by using the `<fx:Model>` tag's `source` property, as the following example shows:

```
<fx:Model source="content.xml" id="Contacts"/>
<fx:Model source="http://www.somesite.com/companyinfo.xml" id="myCompany"/>
```

For more information about data models, see [Storing Data](#).

Data validation

You use data validation to ensure that the data the user enters into your application is valid. For example, if you want the user to enter a valid ZIP code you use a ZIP code *data validator*.

Flex provides predefined data validators for the following types of data: credit card, currency, date, e-mail, number, phone number, regular expression, social security, string, and ZIP code.

Data validators are nonvisual Flex components, which means that you do not access them from the Components panel. Instead, you work with them in code, as the following MXML example shows:

```
<!-- Define the ZipCodeValidator. -->
<mx:ZipCodeValidator id="zcV" source="{zipcodeInput}" property="text"/>
<!-- Define the TextInput control for entering the zip code. -->
<s:TextInput id="zipcodeInput"/>
```

In this MXML example, the validator is defined with the appropriate MXML tag, and it is bound to the ID property of a TextInput control. At run time, when the user enters the phone number into the TextInput control, the number is immediately validated.

You can, of course, use data validators in ActionScript by defining a variable as an instance of a validator class and then creating a function to bind it to the input control.

Data validators are often used with data models. For more information, see [Validating Data](#).

Data formatting

To display the proper format of certain types of data in your application, you use a *data formatter*. Flex provides predefined data formatters for the following types of data: currency, date, number, phone, and ZIP code.

Data formatters are bound to input controls, and they format data correctly after the user enters it. For example, a user might enter a date in this format:

120105

Bound to the text input control is a data formatter that stores and displays the date in this format:

12/01/05

As with data validators, data formatters are nonvisual Flex components that you can work with either as MXML tags or as ActionScript classes.

For more information, see [Formatting Data](#).

Automatically generating Flex Ajax Bridge code

You use the Create Ajax Bridge feature to generate JavaScript code and an HTML wrapper file that let you more easily use a Flex application from JavaScript in an HTML page. This feature works in conjunction with the Flex Ajax Bridge JavaScript library, which lets you expose a Flex application to scripting in the web browser. The generated JavaScript code is very lightweight, as it is intended to expose the functionality that the Flex Ajax Bridge already provides. For more information about the Flex Ajax Bridge, see [Flex Ajax Bridge](#).

The Create Ajax Bridge feature generates JavaScript proxy code that is specific to the Flex application APIs that you want to call from JavaScript. You can generate code for any MXML application or ActionScript class in a Flash Builder project.

For MXML application files, you can generate code for any or all of the following items in the MXML code:

- List of inherited elements, which can expand non-recursively
- Public properties, including tags with `id` properties
- Public constants
- Public functions, including classes defined in line

For ActionScript classes, you can generate code for any or all of the following items:

- List of inherited elements
- Public properties; for each property, a get and set method is displayed
- Public constants
- Public methods

In a directory that you specify, the Create Ajax Bridge feature generates *.js and *.html files that correspond to the MXML applications and ActionScript classes that you select for generation; it places a copy of the Flex Ajax Bridge library (fabridge.js) in a subdirectory of the code generation directory. This feature also generates MXML helper files in the project's `src` directory; these files are used to complete the JavaScript code generation.

Generating Ajax Bridge code

- 1 Right-click a project in the Flex Package Explorer and select Create Ajax Bridge.

- 2 In the Create Ajax Bridge dialog box, select the MXML applications and ActionScript classes for which you want to generate JavaScript code. You can select the top-level check box to include the entire object, or you can select specific members.
- 3 Specify the directory in which to generate proxy classes.
- 4 Click OK to generate the code. The following example shows a .js file generated for an application that displays images:

```

*
* You should keep your JavaScript code inside this file as light as possible,
* and keep the body of your Ajax application in separate *.js files.
*
* Do make a backup of your changes before regenerating this file. (Ajax Bridge
* display a warning message.)
*
* For help in using this file, refer to the built-in documentation in the Ajax Bridge
application.
*
*/

/**
 * Class "DisplayShelfList"
 * Fully qualified class name: "DisplayShelfList"
 */
function DisplayShelfList(obj) {
    if (arguments.length > 0) {
        this.obj = arguments[0];
    } else {
        this.obj = FABridge["b_DisplayShelfList"].
            create("DisplayShelfList");
    }
}

// CLASS BODY
// Selected class properties and methods
DisplayShelfList.prototype = {

    // Fields form class "DisplayShelfList" (translated to getters/setters):

    // Methods form class "DisplayShelfList":

    getAngle : function() {
        return this.obj.getAngle();
    },

    setAngle : function(argNumber) {
        this.obj.setAngle(argNumber);
    },

    setCurrentPosition : function(argNumber) {
        this.obj.setCurrentPosition(argNumber);
    },

    setSelectedIndex : function(argNumber) {
        this.obj.setSelectedIndex(argNumber);
    }
};

```

```
    },

    setPercentHeight : function(argNumber) {
        this.obj.setPercentHeight(argNumber);
    },

    setPercentWidth : function(argNumber) {
        this.obj.setPercentWidth(argNumber);
    },

    DisplayShelfList : function() {
        return this.obj.DisplayShelfList();
    },

    setFirst : function(argNumber) {
        this.obj.setFirst(argNumber);
    },

    setFormat : function(argString) {
        this.obj.setFormat(argString);
    },

    setLast : function(argNumber) {
        this.obj.setLast(argNumber);
    }
}

/**
 * Listen for the instantiation of the Flex application over the bridge.
 */
FABridge.addInitializationCallback("b_DisplayShelfList", DisplayShelfListReady);

/**
 * Hook here all of the code that must run as soon as the DisplayShelfList class
 * finishes its instantiation over the bridge.
 *
 * For basic tasks, such as running a Flex method on the click of a JavaScript
```



```

* button, chances are that both Ajax and Flex have loaded before the
* user actually clicks the button.
*
* However, using DisplayShelfListReady() is the safest way, because it lets
* Ajax know that involved Flex classes are available for use.
*/
function DisplayShelfListReady() {

    // Initialize the root object. This represents the actual
    // DisplayShelfListHelper.mxml Flex application.
    b_DisplayShelfList_root = FABridge["b_DisplayShelfList"].root();

    // YOUR CODE HERE
    // var DisplayShelfListObj = new DisplayShelfList();
    // Example:
    // var myVar = DisplayShelfListObj.getAngle ();
    // b_DisplayShelfList_root.addChild(DisplayShelfListObj);

}

```

- 5 Edit the generated .js files. In the xxxReady() function of the generated .js files, add the code that must run as soon as the corresponding class finishes its instantiation over the Ajax Bridge. Depending on your application, default code may be generated in this method. The bold code in the following example shows custom initialization code for the sample image application:

```

...
function DisplayShelfListReady() {

    // Initialize the root object. This represents the actual
    // DisplayShelfListHelper.mxml Flex application.
    b_DisplayShelfList_root = FABridge["b_DisplayShelfList"].root();

    // Create a new object.
    DisplayShelfListObj = new DisplayShelfList();
    // Make it as big as the application.
    DisplayShelfListObj.setPercentWidth(100);
    DisplayShelfListObj.setPercentHeight(100);
    //Set specific attributes.
    DisplayShelfListObj.setFirst(1);
    DisplayShelfListObj.setLast(49);
    DisplayShelfListObj.setFormat("./photos400/photo%02d.jpg");
    //Add the object to the DisplayList hierarchy.
    b_DisplayShelfList_root.addChild(DisplayShelfListObj.obj);

}

```

- 6 Edit the generated .html files. In the part of the HTML pages that contains the text “Description text goes here,” replace the text with the HTML code that you want to use to access the Flex application from the HTML page. For example, this code adds buttons to control the sample image application:

```

<h2>Test controls</h2>
<ul>
<li><input type="button" onclick="DisplayShelfListObj.setCurrentPosition(0) "
value="Go to first item"/>
</li>
<li><input type="button" onclick="DisplayShelfListObj.setCurrentPosition(3) "
value="Go to fourth item"/>
</li>
<li><input type="button" onclick="DisplayShelfListObj.setSelectedIndex(0) "
value="Go to first item (with animation)"/>
</li>
<li><input type="button" onclick="DisplayShelfListObj.setSelectedIndex(3) "
value="Go to fourth item (with animation)"/>
</li>
<li><input type="button" onclick="alert(DisplayShelfListObj.getAngle()) "
value="Get photo angle"/>
</li>
<li><input type="button" onclick="DisplayShelfListObj.setAngle(15); "
value="Set photo angle to 15&deg;"/>
</li>
</ul>

```

7 Open the HTML page in a web browser to run the application.

Managing Flash Player security

Flash Player does not allow an application to receive data from a domain other than the domain from which it was loaded, unless it has been given explicit permission. If you load your application SWF file from <http://mydomain.com>, it cannot load data from <http://yourdomain.com>. This security sandbox prevents malicious use of Flash Player capabilities. (JavaScript uses a similar security model to prevent malicious use of JavaScript.)

To access data from a Flex application, you have three choices:

- Add a cross-domain policy file to the root of the server that hosts the data service. See [“Using cross-domain policy files”](#) on page 229.
- Place your application SWF file on the same server that hosts the data service.
- On the same server that contains your application SWF file, create a proxy that calls a data service hosted on another server. See [“Setting up Flash Builder to use a proxy for accessing remote data”](#) on page 230.

Using cross-domain policy files

A cross-domain policy file is a simple XML file that gives Flash Player permission to access data from a domain other than the domain on which the application resides. Without this policy file, the user is prompted to grant access permission through a dialog box. You want to avoid this situation.

The cross-domain policy file (named `crossdomain.xml`) is placed in the root of the server (or servers) containing the data that you want to access. The following example shows a cross-domain policy file:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.yourdomain.com" />
</cross-domain-policy>
```

For more information about configuring cross-domain policy files, see the following tech note:
http://www.adobe.com/go/tn_14213.

Setting up a proxy to access remote data

Another option for managing Flash Player security (aside from using a cross-domain policy file) is to use a proxy. LiveCycle Data Services ES provides a complete proxy management system for Flex applications. You can also create a simple proxy service by using a web scripting language such as ColdFusion, JSP, PHP, or ASP.

The proxy service processes requests from the application to the remote service and responses from the remote service back to the application (Flash Player).



When developing your applications, a common technique is to host the proxy on your local computer. To do this, you need to run a web server and scripting language on your local development computer.

For more information about creating your own proxy, see the following tech note:
http://www.adobe.com/go/tn_16520.

Setting up Flash Builder to use a proxy for accessing remote data

After you have set up a proxy to access data from a remote service, you place the application files in the same domain as the proxy. In Flash Builder, you can modify both the project build settings and the launch configuration to manage the use of a proxy.

If you use Flash Builder to compile your applications and the proxy server is also set up on your local development computer, you can modify the project build settings to automatically copy the compiled application files to the appropriate location on your web server.

Modifying the project build path

- 1 In the Flex Package Explorer, select a project.
- 2 Right-click and select Properties. The Project Properties dialog box appears.
- 3 Select the Flex Build Path properties page.
- 4 Change the existing output folder by entering a new path or by browsing to the appropriate folder of your web server (for example, C:\inetpub\wwwroot\myApp\).
- 5 Click OK.

To run and debug the application from the web server, modify the project's launch configuration.

Note: If your proxy server is not your local machine, copy the contents of the bin directory to your server before running or debugging your program.

Modifying the launch configuration

- 1 With the project's main application file open in Flash Builder, right-click in the editor and select Run As > Open Run Dialog. The Create, Manage, and Run Configurations dialog box appears.

- 2 From the list of configurations, select the project's launch configuration.
- 3 (Optional) On the Main tab, deselect the Use Defaults option to modify the URL or path to launch.
- 4 In the Debug text box, enter the URL or path to the debug version of the application.
- 5 In the Profile text box, enter the URL or path to the profiler version of the application.
- 6 In the Run text box, enter the URL or path to the main application file.
- 7 Click Apply and then click Close.

Chapter 11: Customizing Flash Builder

Adobe® Flash® Builder™ is a plug-in to the Eclipse development platform. Specify Flash Builder preferences in addition to the general preference you specify for Eclipse. For some features, you specify preferences in both Eclipse and Flash Builder nodes of the Preference dialog. For example, when setting preferences for the Flash Builder debugger, you specify custom behavior under the Eclipse Run/Debug node as well as the Flash Builder > Debug node.

How you open the Preference dialog varies according to platform and whether you are using the standalone or plug-in version of Flash Builder.

Specify Flash Builder preferences

- 1 Open the Preferences dialog:
 - (Windows) Select Windows > Preferences
 - (Macintosh, Standalone) Select Windows > Preferences
 - (Macintosh, Plug-in) Select Eclipse > Preferences
- 2 Expand the Flash Builder node to view and specify user preferences.

Adobe Preferences

Set preferences for Adobe plugin modules.

RDS Configuration

Remote Development Server (RDS) configuration information applies to users of Adobe LiveCycle® Data Services or Adobe BlazeDS. RDS provides access to LiveCycle Data Services and BlazeDS destinations.

The default RDS configuration is a starting point for connecting to data services. Modify the default configuration to provide access to your server destination or database.

See the [Adobe LiveCycle Data Services](#) documentation for information on configuring RDS.

Important: Your RDS configuration has security implications. See the *LiveCycle Data Services documentation for information on application server security implications when specifying an RDS configuration.*

Flash Builder preferences

Flash Builder

Warn before upgrading old Flash Builder projects

Flash Builder updates projects that were created with an earlier version of Flash Builder. Flash Builder updates the project files and structure to conform to the current structure of Flash Builder projects. A converted project is no longer accessible under the earlier version of Flash Builder.

By default, Flash Builder warns you before making the conversion. Disable this option if you want conversions to happen silently.

Data/Services

The following user preferences are available for Data/Services. These preferences apply to all projects in your Flash Builder developer environment.

You can override these preferences on a project basis. Select Project > Properties > Data/Services to override the preferences specified here.

Code Generator

Flash Builder provides a default code generation utility for generating access to data services. Using DCD extensibility features, you can create your own code generation utilities and add them to Flash Builder as a plug-in.

Any code generation extensions you add as a plug-in to Flash Builder are available from the Code Generator combo box.

Enable Service Manager to use single service instance (singleton) during code generation

By default this option is not enabled. During code generation, each client application in a project creates their own instance of a data service.

If you want a single instance of a service that all client applications in the project share, enable this option.

This option is only available if you specify the default Flash Builder code generation utility.

Debug

The following options for debugging in Flash Builder are automatically enabled. For other options that affect a debugging session see:

- Preferences > General > Web Browser
- Preferences > Run/Debug

Warn when trying to launch multiple debugging sessions

Some platform/browser combinations do not allow concurrent debugging sessions. If you attempt to start a second debugging session, the original debugging session terminates or is disconnected.

Leave this option enabled for Flash Builder to warn you when you attempt to start a second debugging session.

Warn about Security Error after launch

In some cases, a web browser issues a security error because it cannot read Flash Player's security trust file. In most cases, restarting the web browser corrects the security error.

Leave this option enabled if you want Flash Builder to warn you about this type of security error.

See [Tech Note for Flash Player security warning](#).

Automatically invoke getter functions

When stepping through a debugging session, variables representing accessor (getter) functions are automatically evaluated. Typically, this behavior is useful during a debugging session.

Disable this option if you do not want to automatically evaluate variables representing accessor functions when stepping through a debugging session.

Flash Builder Editors

Braces

Flash Builder provides user options for indenting, inserting, and highlighting braces representing blocks of code.

Code assist

When using the MXML or ActionScript source editor, Flash Builder provides code hints to help you complete your code expressions. This assistance includes help in selecting recommended classes, properties, and events available.

- **Enable Auto-activation**

Disable Enable Auto-activation if you do not want code hints to automatically appear as you type. If you disable this option, you can access code hints by pressing Control+Spacebar.

- **Auto-activate After**

Specify the time, in milliseconds, for code hints to appear after you begin typing. The default time is 100 milliseconds.

More Help topics

[“Flash Builder coding assistance”](#) on page 97

General Flash Builder editor preferences

By default, Flash Builder provides code folding and automatic indentation of lines as you type or paste in the source editor. You can disable the default preferences for these features.

Eclipse general editor preferences

Additional general editor preferences are available by selecting Preferences > General > Editors

ActionScript code

When editing ActionScript files in the source editor, Flash Builder provides default features for wrapping and organizing code. By default, Flash Builder places all import statements at the head of an ActionScript file and removes import statements that are not referenced in the code body.

By default, ActionScript reference documentation, when available, appears with code hints or by hovering the pointer over a language element.

Each of these default features can be disabled.

More Help topics

[“Flash Builder coding assistance”](#) on page 97

Design mode

Flash Builder provides the following user preferences for Design mode of the source editor:

- **Automatically show design-related views**

When switching to Design mode of the source editor, Flash Builder automatically opens related views, such as the Properties view, States view, Components view, and Appearance view.

Disable this preference if you want to customize which views are open in Design mode.

- Enable snapping

In Design mode, Flash Builder provides an invisible grid that automatically “snaps” in place components that are added or rearranged.

Disable this preference if you want more control of the placement of components.

- Render skins when opening a file

Design mode draws skinned components that are in the design area.

If you do not want to render the skins of components in Design mode, disable this preference.

- Collapse data binding expressions

In Design mode, data binding expressions typically do not accurately reflect the shape or size of the data to which the expressions are bound.

Enable this preference to allow Design mode of the editor to more accurately render components that are bound to data binding expressions.

- Always update references when changing IDs in Properties view

When you change a component ID in Properties view, it can affect code in your workspace that references that ID.

Enable this preference if you want Flash Builder to automatically update all references to component IDs when you edit a component ID in Properties view.

MXML code

When editing MXML files in the source editor, Flash Builder provides default features for code completion. Each of these features can be disabled.

Flash Builder also provides default behavior for the content assist feature for editing MXML files. When content assist is available, Flash Builder displays hints on available language elements for insertion into the code. You cycle through the elements displayed in the Content Assist window by pressing Control+Spacebar.

Use the Advanced MXML preference to disable any type of language element from the Content Assist display cycle.

More Help topics

[“Flash Builder coding assistance”](#) on page 97

[“Using Content Assist”](#) on page 101

Syntax coloring

Flash Builder provides default syntax coloring and text highlighting for ActionScript, CSS, and MXML files. You can override these default features. Expand the language node and select a language feature to override the default feature.

General Eclipse editor preferences are also available. See:

- Preferences > General > Text Editors
- Preferences > General > Colors and Fonts

File Templates

Flash Builder uses file templates when creating new MXML, ActionScript, and CSS files. You can customize the templates Flash Builder uses to create these files. You can also import template files and export template files.

In the File Types area select a template for a file type. Click Edit to modify the template. Disable Automatically Indent New Files if you do not want to use the Flash Builder indenting preferences when creating files.

You can change the Flash Builder indentation preferences. See:

Preferences > Flash Builder > Indentation

More Help topics

[“Customizing File Templates”](#) on page 120

[“Indentation”](#) on page 236

FlexUnit

By default, when you run a FlexUnit test, Flash Builder launches in Debug mode.

Use this preference to change the Flash Builder launch mode for FlexUnit tests. Default launch configurations for Flash Builder are the following:

- Run mode
By default, launches Flash perspective.
- Debug mode
By default, launches Flash Debug perspective.
- Profile mode
By default, launches Flash Profile perspective.

The Eclipse preferences for Launching and Perspectives configure launch modes. See:

- Preferences > Run/Debug > Launching
- Preferences > Run/Debug > Perspectives

More Help topics

[“FlexUnit test environment”](#) on page 140

[“About Flash Builder perspectives”](#) on page 7

Indentation

By default, Flash Builder uses tabs for indenting code. You can change the default setting to use spaces. The default tab and indent sizes are each equal to four spaces.

Select ActionScript and MXML in the Indentation preferences dialog to preview indentation settings. You can also customize the indentation heuristics.

Installed Flex SDKs

By default, Flash Builder installs the Flex 4 SDK and the Flex 3.4 SDK.

By default, Flash Builder uses the Flex 4 SDK for projects that do not specify an SDK.

You can add additional SDKs, remove SDKs, and also change which SDK is the default for projects that do not specify an SDK.

By default, Flash Builder prompts you for which SDK to use when importing projects created in an earlier version of Flash Builder. You can disable this prompt for importing projects.

Network Monitor

The Network Monitor preferences page lists the ports on which the Network Monitor captures events and listens to HTTP requests.

By default, the Network Monitor clears all monitoring entries upon startup. You can disable this preference.

You can also enable the following preferences:

- Suspend monitoring on start
- Ignore SSL Security Checks

Enabling this preference is useful when monitoring network traffic from a self-signed server.

Profiler

By default, both memory and performance profiling are enabled.

- Memory profiling

You typically use memory profiling to examine how much memory each object or type of object is using in the application. Use the memory profiling data when to reduce the size of objects, reduce the number of objects that are created, or allow objects to be garbage collected by removing references to them.

Memory profiling uses much more memory than performance profiling, and can slow your application's performance.

- Performance profiling

You typically use performance profiling to find methods in your application that run slowly and that can be improved or optimized.

You can also specify the following preferences for the profiler:

- Connections

Specify the port number that Flash builder listens to when profiling an application. The default port number is 9999. You cannot set it to port 7935, which the Flash debugger uses.

- Exclusion filters

Defines the default packages that are excluded from profiler views.

- Inclusion Filters

Defines the default packages that are included in the profiler views. All other packages are excluded.

- Object References

Specifies the number of back-reference paths to instances of an object to display. The back-reference paths help determine if a path has a back-reference to the garbage collector (GC Root). An instance that was expected to be released, but has references to GC Root, indicates a memory leak.

By default, the profiler displays ten back reference paths. You can specify a different maximum to display or you can specify to show all back-reference paths.

- Player/Browser

You can specify which standalone Adobe Flash Player to use and which Web browser to use for profiling external applications. Use a debugger version of Flash Player to successfully profile the application.

When profiling external applications, by default, Flash Builder uses the following Flash Players:

- URL to a SWF file

Flash Builder launches the application's SWF file using the default browser for the system.

- File system location for a SWF file

Flash Builder opens the application with the default debugger version of the stand-alone Flash Player.

More Help topics

[“Using the profiler”](#) on page 151

[“About the profiler views”](#) on page 156

Extending Flash Builder

The Flash Builder Extensibility API allows you to extend Flash Builder to support custom components. Using the Flash Builder Extensibility API, you can extend Flash Builder Design view and Properties view so they properly handle custom components.

Additionally, you can extend Flash Builder support for services available from the Services wizard.

For more information, see the [Flash Builder 4 Extensibility API Reference](#).