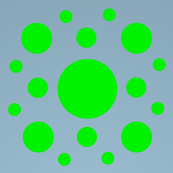


SYNX
PLATFORM

Developing for FMS just got easier.

USER GUIDE
VERSION 1.0

TABLE OF CONTENTS

Your guide to this guide.

1 WHAT IS THE SYNX PLATFORM?

The best feature FMS doesn't have.

Simply put, the Synx Platform is a micro-architecture for building applications that use Flash Media Server. It supports the audio/video streaming and collaborative features found in Flash® Media Interactive Server (FMIS), and makes development of applications that utilize advanced features of FMIS painless. It also makes building applications more fun.

Why did we build Synx?

How many times have you been working on an application and realized that if you needed to build a similar application in the future, you'd be writing a lot of the same code again? Since we build applications around FMS, we saw a need for a lightweight framework to handle the mundane common coding we encounter when working with FMS. However, once we got started, we began dreaming and making it do more than we'd ever imagined. Then we decided to give it to you to see what you could build with it.

How much do I need to know to use Synx?

Honestly, the guide you're holding in your hand (or reading on your screen) can take you from zero to hero in Synx in only a couple of hours. Anybody with a basic knowledge of Actionscript should be able to get going with the platform by running through the walkthroughs and examples contained in this guide.

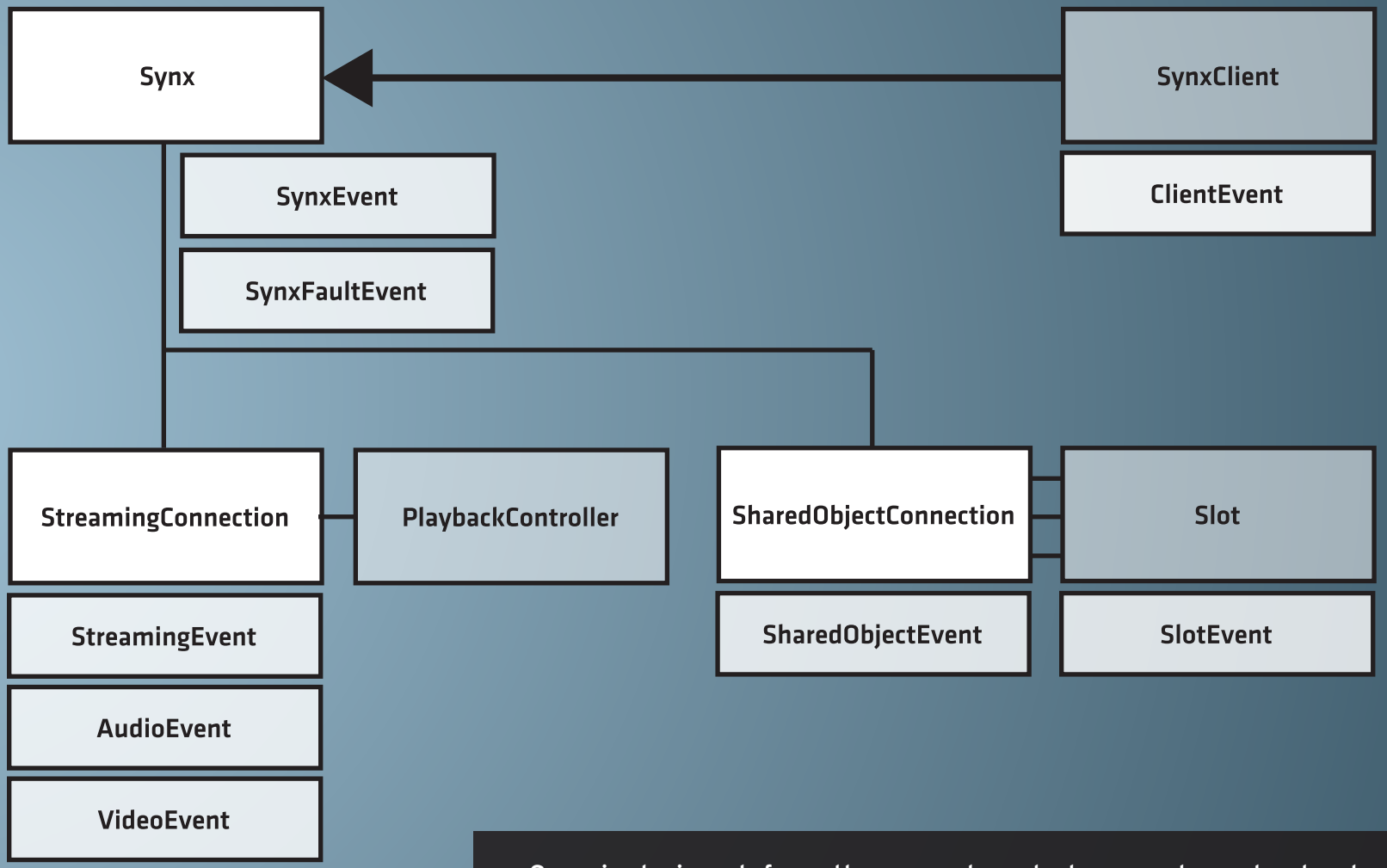
How much does it cost?

Nothing. Nada. Zip. Zilch. Free, as in beer. All we ask is that you let people know you used the Synx Platform to build your application. We'd love for you to host your Synx applications with Influxis, but we don't require it. And you can get the full source code for Synx by following the link on the Synx Platform website. It's an open source project, though we're keeping code commiters internal for now to ensure quality. This policy will be reviewed, however, as the community begins to use the platform and provide us with feedback.

Really?

Yes, we're serious about keeping Synx free. However, Influxis customers may have access to additional functionality that relies on our server-side code in the future.

SIMPLICITY BY DESIGN



Synx is designed, from the ground up, to be easy to understand and use. This diagram shows the core classes for the entire platform. There aren't many of them, are there? The best part, though, is that you'll only need to know how to use three of these classes for most applications.

GETTING STARTED

Once you know how, it's like riding a bike.

2

DOWNLOADING SYNX

The easiest way to get started with Synx is to download the latest Synx Platform binary (.swc) from our website at synxplatform.com. You can also download the source code from the SVN repository, and reference it directly in your application.

Download Synx from http://www.influxis.com/synx/bin/SynxSWC.zip

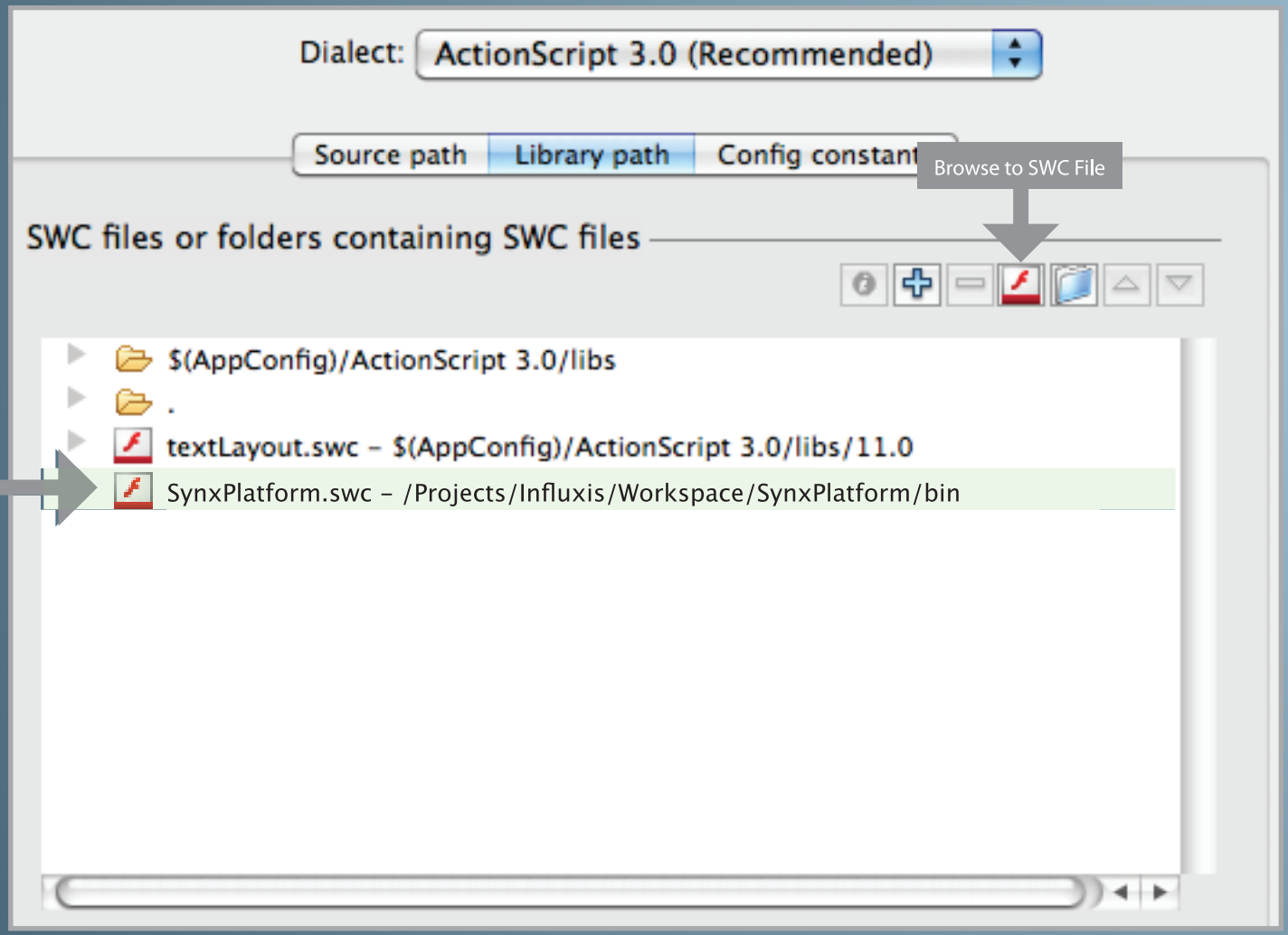
View the repository at https://influxis.svn.beanstalkapp.com/synx/

CONFIGURING SYNX IN YOUR DEVELOPMENT ENVIRONMENT

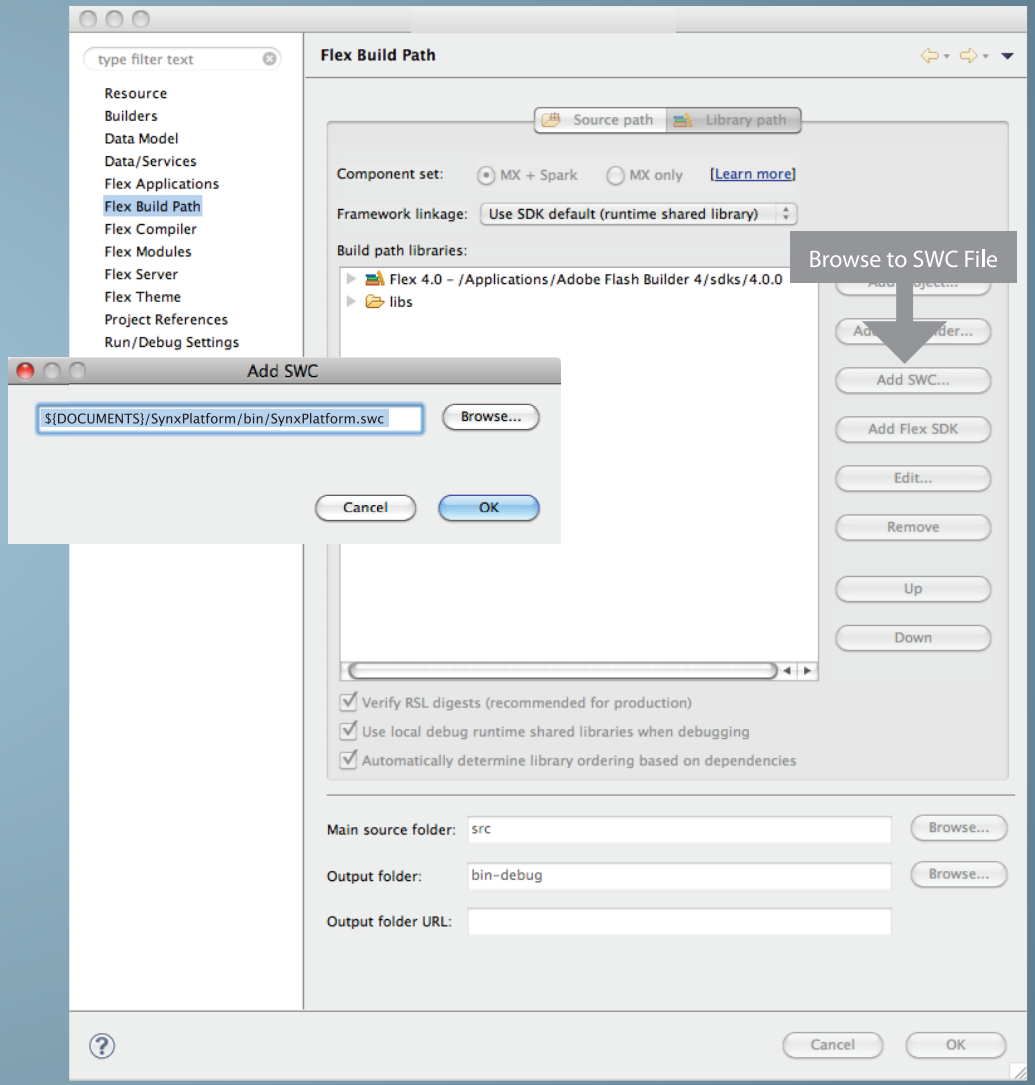
The next few pages cover how to import the Synx Platform binary in the three most popular development environments for Flash. We'll start with Adobe Flash Professional, cover importing the library in Adobe Flash Builder 4 and Adobe Flex Builder, then finish with instructions for using the library in FDT.

CONFIGURING FLASH PROFESSIONAL

Create a new Actionscript 3.0 Project in Flash. Once you've created your project, open the File menu and choose ActionScript Settings. You'll see three tabs across the middle of the settings window; Source Path, Library Path and Config Constants. Select Library Path, then click the fourth button in the button bar, Browse to SWC File. Choose SynxPlatform.swc from your download location, and you're good to go!



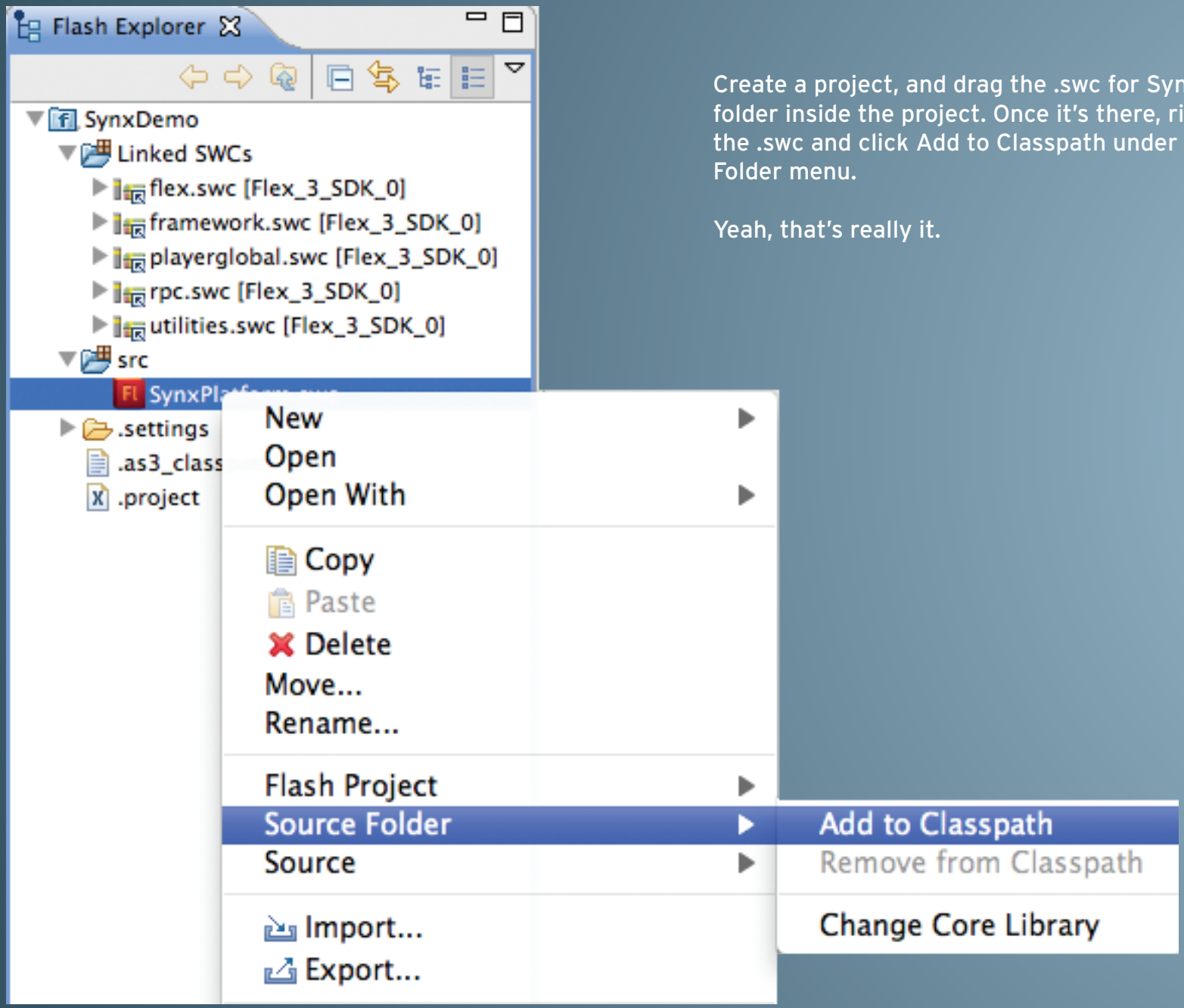
CONFIGURING FLASH BUILDER



Create a new project (Flex or Actionscript) in Flash Builder, or select an existing project you'd like to use with Synx. Right click on the project in the Package Explorer (Project Navigator in FB3) and click on Properties or select Properties from the Project menu. Choose Flex Build Path from the list on the left side of the Project Properties dialog, and select the Library Path tab if it isn't already selected.

Click on Add SWC..., then browse to the location where you downloaded the SynxPlatform.swc. Click OK a few times, and you're ready to start using Synx with Flex!

* The process for importing is the same for both Flash Builder 4 and Flex Builder 3.

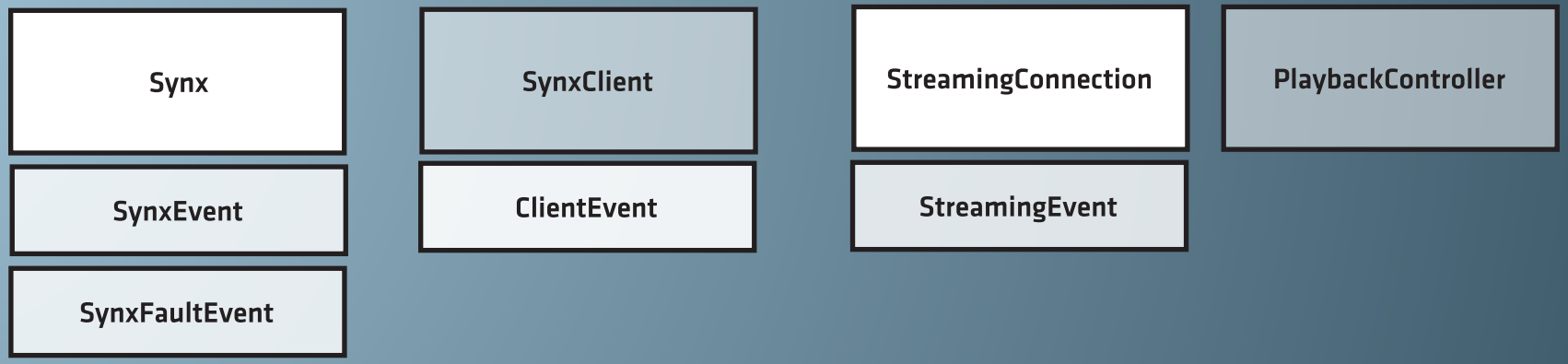


Create a project, and drag the .swc for Synx into a folder inside the project. Once it's there, right click on the .swc and click Add to Classpath under the Source Folder menu.

Yeah, that's really it.

3 STREAMING VIDEO FROM FMS

Because progressive download sucks.



THE SYNX CLASS

To use any part of the Synx Platform, you have to instantiate the platform's Synx singleton class. Because it's a singleton, it can't be instantiated using the new keyword, but is rather instantiated using it's init() method. While this may sound complicated if you're newer to Actionscript, the process is really quite simple:

```
Synx.init("rtmp://youraccount.rtmphost.com/appname/");
```

You would generally call this in an initialization method for your application, either in your document class for Flash, or in the creationComplete handler in a Flex application. Once this is instantiated, it is used by the rest of the platform classes to connect to the FMS server.

So, what does this platform class do, exactly?

The class contains a NetConnection instance that is used to establish your connection to the remote FMS server, sets the the NetConnection's client object to an instance of the SynxClient class, and tells the NetConnection to connect to the URI that you provided to the init() method. This NetConnection is then used by the other Synx classes to connect to your server so that only one instance of the NetConnection class is created for the lifespan of the application.

It also fires events for both success and error messages that come back from the NetConnection, NetStream or SharedObject classes that power the Synx platform. For success messages, it broadcasts a SynxEvent, and for errors, it broadcasts a SynxFaultEvent. This is all handled in the netStatusChangeHandler() method, making it easy to capture the myriad of messages that come back from the FMS server while you're connected.

Let's break this down a little further over the next few pages.

HANDLING SYNX EVENTS

There are a ton of status messages coming back from FMS, and we handle them all in the Synx class, making it easy for you to respond to each one. If the message is a success message, it will become a SynxEvent; if it's an error message, it will become a SynxFaultEvent. The following is a list of all SynxEvent and SynxFaultEvent types:

Class	Description
SynxEvent.READY from NetConnection.Connect.Success	The Synx Platform has been initialized and the initial NetConnection connect attempt has succeeded.
SynxEvent.NC_CONNECTION_CLOSED from NetConnection.Connect.Closed	When you close the connection, this event lets you know that the connection was closed successfully.
SynxEvent.NS_PUBLISH_START from NetStream.Publish.Start	The request to publish the stream was successful.
SynxEvent.NS_RECORD_START from NetStream.Record.Start	The request to record the stream was successful.
SynxFaultEvent.NC_CONNECTION_FAILED from NetConnection.Connect.Failed	The connection attempt failed.
SynxFaultEvent.NC_CONNECTION_REJECTED from NetConnection.Connect.Rejected	The connection attempt did not have permission to access the application
SynxFaultEvent.NC_CONNECTION_INVALID_APP from NetConnection.Connect.InvalidApp	The application name specified during connect is invalid.
SynxFaultEvent.NC_CONNECTION_APP_SHUTDOWN from NetConnection.Connect.AppShutdown	The specified application is shutting down.
SynxFaultEvent.NC_CALL_BADVERSION from NetConnection.Call.BadVersion	Packet encoded in an unidentified format.
SynxFaultEvent.NC_CALL_FAILED from NetConnection.Call.Failed	The NetConnection.call method was not able to invoke the server-side method or command.

Class	Description
SynxFaultEvent.NC_CALL_PROHIBITED from NetConnection.Call.Prohibited	An Action Message Format (AMF) operation is prevented for security reasons. Either the AMF URL is not in the same domain as the file containing the code calling the NetConnection.call() method, or the AMF server does not have a policy file that trusts the domain of the the file containing the code calling the NetConnection.call() method.
SynxFaultEvent.NS_PUBLISH_BAD_NAME from NetStream.Publish.BadName	You've attempted to publish a stream which is already being published by someone else!
SynxFaultEvent.NS_PLAY_FAILED from NetStream.Play.Failed	An error has occurred in playback for a reason other than those listed elsewhere, such as the subscriber not having read access.
SynxFaultEvent.NS_PLAY_STREAM_NOT_FOUND from NetStream.Play.StreamNotFound	The FLV passed to the play() method can't be found.
SynxFaultEvent.NS_PLAY_FILE_STRUCTURE_INVALID from NetStream.Play.FileStructureInvalid	The application detects an invalid file structure and will not try to play this type of file. AIR and player 9.0.115.0 and later.
SynxFaultEvent.NS_PLAY_NO_SUPPORTED_TRACK_FOUND from NetStream.Play.NoSupportedTrackFound	The application does not detect any supported tracks (video, audio or data) and will not try to play the file. AIR and player 9.0.115.0 and later.
SynxFaultEvent.NS_PLAY_INSUFFICIENT_BANDWIDTH from NetStream.Play.InsufficientBW	Flash Media Server only. The client does not have sufficient bandwidth to play the data at normal speed.
SynxFaultEvent.NS_RECORD_NO_ACCESS from NetStream.Record.NoAccess	Attempt to record a stream that is still playing or the client has no access rights.
SynxFaultEvent.NS_RECORD_FAILED from NetStream.Record.Failed	An attempt to record a stream failed.
SynxFaultEvent.NS_SEEK_FAILED from NetStream.Seek.Failed	The seek fails, which happens if the stream is not seekable.
SynxFaultEvent.NS_SEEK_INVALID_TIME from NetStream.Seek.InvalidTime	For video downloaded with progressive download, the user has tried to seek or play past the end of the video data that has downloaded thus far, or past the end of the video once the entire file has downloaded. The message.details property contains a time code that indicates the last valid position to which the user can seek.

Class	Description
SynxFaultEvent.SO_FLUSH_FAILED from SharedObject.Flush.Failed	The 'pending' status is resolved, but the SharedObject.flush() failed.
SynxFaultEvent.SO_BAD_PERSISTANCE from SharedObject.BadPersistance	A request was made for a shared object with persistence flags, but the request cannot be granted because the object has already been created with different flags.
SynxFaultEvent.SO_URI_MISMATCH from SharedObject.UriMismatch	An attempt was made to connect to a NetConnection object that has a different URI (URL) than the shared object.
SynxFaultEvent.UNHANDLED_EXCEPTION	Any message that can't be determined to be one of the above types will fire an unhandled exception.

Remember, you don't have to handle every one of these events, they're just there if you want to listen.

For now, we're only going to use one of these events, SynxEvent.READY, to let us know that our application is ready to go and we can create our StreamingConnection for playback.

PLAYING SOME VIDEO

The StreamingConnection class is used for all streaming to and from Flash Media Server. We'll start off with the basics, playing video back from your FMS server.

As dicussed before, we use the Synx class to get everything set up and ready to use. For playback, we initialize the Synx class with the URI where our videos are located on FMS. After the platform is initialized, we create an instance of the StreamingConnection class and listen for the READY event.

```
import com.synxplatform.core.Synx;
import com.synxplatform.fms.streaming.StreamingConnection;
import com.synxplatform.fms.streaming.events.StreamingEvent;

private var stream:StreamingConnection;

private function init():void
{
    Synx.init("rtmp://youraccount.rtmphost.com/appname/");
    stream = new StreamingConnection();
    stream.addEventListener(StreamingEvent.READY, setup, false, 0, true);
}
```

CONTROLLING PLAYBACK

The setup method is pretty simple, too. We create a Video object, attach the stream object's NetStream to it, then add it as a child to the application. After that, we use the StreamingConnection's PlaybackController object, stream.control, to call the addToPlaylist() method for each video we want to play, in order. Once we've added videos to the playlist, we can call control.play() to start playing the first video.

```
private function setup(event:StreamingEvent):void
{
    var playback:Video = new Video();
    playback.attachNetStream(stream.ns);
    this.addChild(playback);
    stream.control.addToPlaylist("video1");
    stream.control.addToPlaylist("video2");
    stream.control.addToPlaylist("video3");
    stream.control.loopPlayback = true;
    stream.control.play()
}
```

That's all it takes to build a simple video playback application with the Synx Platform. Let's take a look next at some of the other playback features in StreamingConnection and build a simple video player.

BUILDING A CONTROL BAR

So, you've gotten your first taste of streaming by building a simple playback system, but most people want a little more control over video on the web, right? We want total control over playback, including the ability to pause, resume, seek, control the volume and enter a full-screen view of our video content.

All playback control happens through the PlaybackController, as mentioned earlier. There are pre-built methods to handle just about everything you'll need, from toggling between play and pause to panning audio in the stereo field. Let's take a closer look at the relevant parts of the PlaybackController class:

Method	Purpose
`addToPlaylist(location:String):int`	Adds a video to the video playlist. Returns the index position of the last video added to the playlist array.
`play(index:Number=0):void`	Plays the video from a specified point in the playlist. The default is the first item in the playlist.
`togglePlayPause():Boolean`	Toggles between paused and playing state. Returns a Boolean of true if playing, false if paused.
`stop():Boolean`	Closes the NetStream, stopping all playback.
`setVolumeTo(level:uint):void`	Uses the SoundTransform class to set the volume level on the video stream. Requires one attibute, level, which expects an integer value in the range of 1 to 100.
`panAudioTo(degree:int):void`	Uses the SoundTransform class to pan audio in the stereo field. Requires one attribute, degree, which accepts integer values between -90 and 90, with -90 representing a full pan to the left speaker, and 90 representing a full pan to the right speaker.

Let's pick things back up where we left off, adding playback controls to our simple video application. We're going to assume that you know how to create buttons and toggle buttons for the play/pause button, and that you're using the slider components from Flash Pro or Flex for your scrub and volume controls. If you're in Flash, use TextFields for current time and duration. In Flex, use Label or RichText controls.

I'm sure you've already noticed that everything needed for your control bar is already in the PlaybackController class, we just need to call it at the appropriate times. So, add some click handlers to your buttons and change handlers to your sliders, similar to those included here:

```
playPause.addEventListener(MouseEvent.CLICK, playPauseHandler);
fullScreen.addEventListener(MouseEvent.CLICK, fullScreenHandler);
scrubBar.addEventListener(Event.CHANGE, scrubHandler);
volumeBar.addEventListener(Event.CHANGE, changeVolume);
```

Writing the handlers for these is really pretty easy, with the exception of the fullScreenHandler method. Let's take a look at the code for the other three first:

```
private function playPauseHandler(event:MouseEvent):void
{
     stream.control.togglePlayPause();
}

private function changeVolume(event:Event):void
{
     stream.control.setVolumeTo(event.target.value);
}

private function scrubVideo(event:Event):void
{
     stream.ns.seek(event.target.value);
}
```

Simple, right? The full screen code is a little more complicated, so let's delve into what's needed for full screen scaling in Flash Player. The first thing we need to do is ensure that the display is not currently scaled to full screen. To do that, we look at the value of stage.displayState and compare it to the StageDisplayState constant, FULL_SCREEN. If the player is not already scaled, we can begin the process of transitioning to full screen.

To scale an object in Flash Player to full screen, we need to define a rectanglular area of the screen that we'd like to scale. To do this, the most common practice is to use a Rectangle and set it's dimensions to our video player's dimensions, including the control bar.

After drawing the rectangle, we then set the stage's fullScreenSourceRect to our Rectangle, and finally set the stage's displayState to the StageDisplayState.FULL_SCREEN constant. When switching back from full screen, we just set the stage's displayState back to StageDisplayState.NORMAL. Here is an example of what that looks like in code:

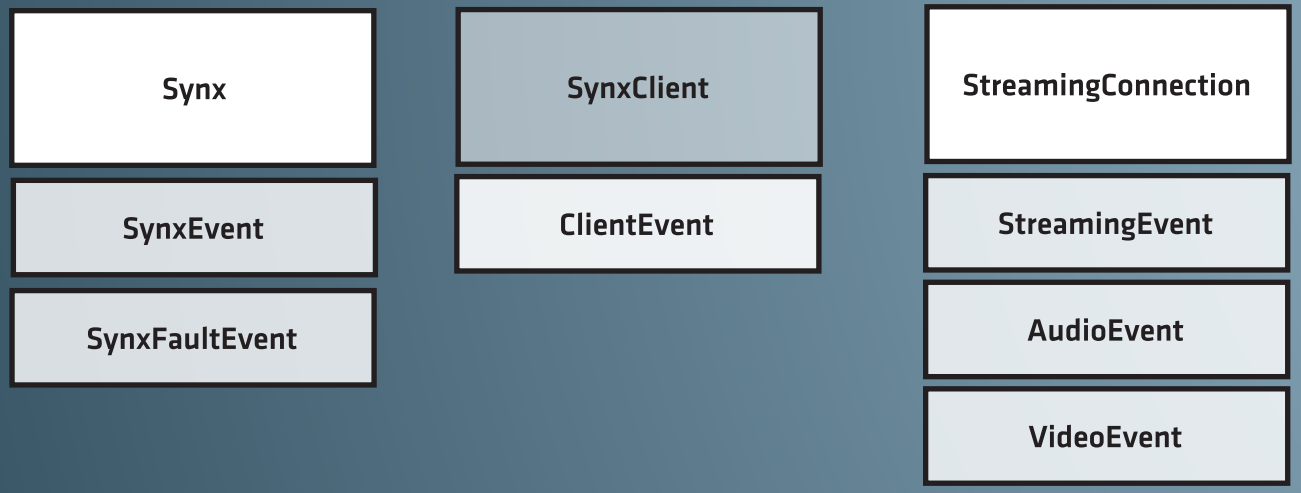
```
private function handleFullScreen(event:MouseEvent):void
{
     if (stage.displayState != StageDisplayState.FULL_SCREEN)
     {
          var screenRectangle:Rectangle = new Rectangle(playbackContainer.x,
                                                        playbackContainer.y,
                                                        playbackContainer.width,
                                                        playbackContainer.height);
          stage.scaleMode = StageScaleMode.NO_SCALE;
          stage.fullScreenSourceRect = screenRectangle;
          stage.displayState = StageDisplayState.FULL_SCREEN;
     }
     else
     {
          stage.displayState = StageDisplayState.NORMAL;
     }
}
```

That's it for basic playback control. We'll be adding more features to the PlaybackController, including more playlist options, like shuffle and repeat playback modes, in future versions of the platform.

STREAMING AND RECORDING TO FMS

Bringing more amateur video to the web.

4



PUBLISHING VIDEO TO FMS

One of the most compelling uses of FMS is live streaming, or webcasting. We're going to show you how to do it in a few ways. The first will be using a video capture application you build to broadcast live video, and the second will, with just a few changes, allow you to record that video to your server. These are the building blocks of social video sites, which allow users to generate video content.

Flash Player has a built-in video encoder, and while it's not the highest quality encoder in the world (it uses the Sorenson Spark codec that's been around since Flash Player 7), it works for smaller videos, especially user generated videos. For any professional usage, like multi-bitrate streaming of live events or high definition streaming, we recommend using Adobe® Flash® Media Live Encoder or any of the other great products for live streaming.

So, how do we get started with getting video from a client machine to FMS? Well, much the same way as we got video to stream from FMS, the StreamingConnection class. We're also going to set up a few event listeners, which I'll explain after the code:

```
public function init():void
{
     Synx.init("rtmp://youraccount.rtmphost.com/appname/");
     stream = new StreamingConnection("streamname");
     stream.addEventListener(StreamingEvent.READY, setup, false, 0, true);
     stream.addEventListener(AudioEvent.LEVEL, audioMeterUpdate, false, 0, true);
     stream.addEventListener(VideoEvent.LEVEL, videoMeterUpdate, false, 0, true);
}
```

There are a few differences here from last time. First, we're initializing Synx to the application to which we want to stream our videos. We're also specifying a stream name when we instantiate StreamingConnection, which will become the file name for our video stream. Additionally, we're setting up a number of event listeners we haven't seen before. We'll talk more about those shortly.

ACCESSING CLIENT HARDWARE

Once we've created our listeners, we'll use the setup method, as before, to start using our StreamingConnection once it's connected to FMS.

The setup event is again very similar to what we wrote for streaming from FMS, with a few minor additions. This time, we're going to use it to capture the Camera and Microphone objects from Flash Player using our attachAudio() and attachCamera() helper methods, which, called without parameters, attach the default camera and audio devices to the NetStream.

```
private function setup(event:StreamingEvent):void
{
     stream.attachAudio();
     stream.attachCamera();
     var playback:Video = new Video();
     playback.attachCamera(stream.camera);
     this.addChild(playback);
}

private function startStream(event:MouseEvent):void
{
     stream.start(stream.name, true);
}

private function stopStream(event:MouseEvent):void
{
     stream.stop();
}
```

In this example, we're using our Video object to show the video that's being captured by the local player using the Video's attachCamera() method to attach the stream's camera object to the playback surface.

On a click event on a record button, we call stream.start(), passing the stream name and a boolean value that tells FMS whether or not you'd like to record the video to a file as it streams. Finally, we add a button that calls stopStream() when we're ready to stop the broadcast.

MONITORING THE BROADCAST LEVELS

Metering is something that is frequently used in broadcast applications to track microphone volume level or the video activity level, and we've made it easy to capture those levels in Synx with AudioEvent and VideoEvent. Here's some more detail on both events, which we added listeners for earlier:

Event	Purpose
`AudioEvent.ACTIVITY`	Occurs when the state of activity changes on the Microphone object. Contains the isActive boolean property, which allows you to determine if the microphone is active or inactive.
`AudioEvent.STATUS`	Contains the details of a StatusEvent that happened on the Microphone object in the properties code and level. See StatusEvent.STATUS for more details, as this is simply rebroadcasting that event.
`AudioEvent.LEVEL`	Fires every time the input volume level of the Microphone changes. Contains a property, micLevel, that can be used to display the current input volume.
`VideoEvent.ACTIVITY`	Occurs when the state of activity changes on the Camera object. Contains the isActive boolean property, which allows you to determine if the microphone is active or inactive.
`VideoEvent.STATUS`	Contains the details of a StatusEvent that happened on the Camera object in the properties code and level. See StatusEvent.STATUS for more details, as this is simply rebroadcasting that event.
`VideoEvent.LEVEL`	Fires every time the motion level of the Camera changes. Contains a property, activity, that can be used to display the current percentage of change between frames.

So, in our example, we're listenting for the LEVEL events on both of our client devices, the camera and the microphone. As we can see from the descriptions, the AudioEvent has a property called micLevel that we can use to display feedback about the current input volume, and the VideoEvent has a property called activity that can be utilized to show the amount of change between video frames.

Behind the scenes here, it's using a timer to track the changes, and that timer's delay is fully configurable by you as a developer, though you may never want to change it. You see, when we developed this feature, we knew we wanted it to be as CPU friendly as possible, so we automatically turn the timer off when it's not needed (when there's not activity on either the camera or the microphone), and turn it back on as soon as it's needed again. There is one timer for both activity monitors, which reduces overhead even further.

Using a couple of progress bars, we get a pretty clear visual indicator of activity levels without a ton of work:

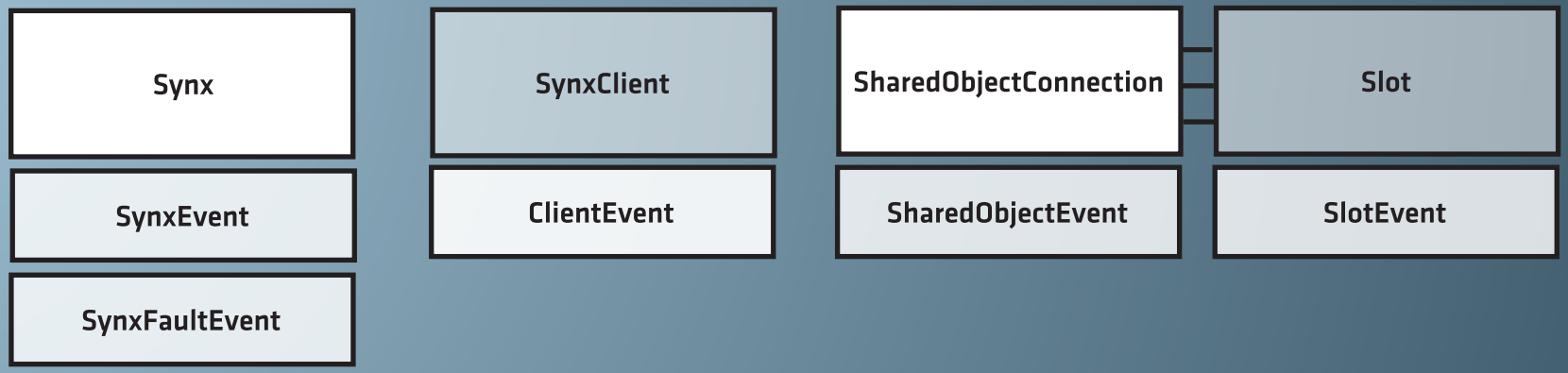
```
private function audioMeterUpdate(event:AudioEvent):void
{
        audioMeter.setProgress(event.micLevel, 100);
        audioMeterLabel.text = String(event.micLevel);
}

private function videoMeterUpdate(event:VideoEvent):void
{
        videoMeter.setProgress(event.activity, 100);
        videoMeterLabel.text = String(event.activity);
}
```

So, now you can stream from and to FMS, but what about all those interactive features, the "I" in FMIS?

5 CREATING COLLABORATIVE APPS

Promoting togetherness one app at a time



SHARING WITH OTHERS

After all these chapters, I'll bet you know where to start your app now. That's right, the Synx class. But we're not going to be using StreamingConnection in this application, we'll be using a brand new class, SharedObjectConnection. SharedObjectConnection wraps all the details of working with remote SharedObjects into a nice, neat package, which makes building collaborative applications easier than ever before. As always, let's start with the initialization method:

```
private var sChat:Slot;

public function init():void
{
     Synx.init("rtmp://youraccount.rtmphost.com/appname/");
     soConn = new SharedObjectConnection();
     soConn.addEventListener(SharedObjectEvent.READY, initSlots, false, 0, true);
}
```

Pretty simple so far, right? One of our core design goals for Synx was to have everything work in a similar way, so we didn't want to make you relearn how everything works just to switch classes. We're listening for a READY event, just like with StreamingConnection, except we're calling our setup method initSlots(). To understand why, lets talk for a minute about how SharedObjects work.

A METAPHOR-FILLED LOOK AT SHARED OBJECTS

So, what exactly is a remote SharedObject? Well, it's very similar to local SharedObjects, in that it stores data. This data is stored in a slot, and a SharedObject can have multiple slots for storing multiple types of information. For example, in a chat application you may use one slot to store the messages that are being sent by users and another slot to store a list of active users in the chat room. Think of a SharedObject as a file drawer, with each slot being a folder. Now that you've gotten that picture, let's talk about the issues with managing slots in traditional FMS applications.

Honestly, the biggest problem with using SharedObjects without Synx is that they're really, really hard to keep track of. In fact, the first part of Synx we built was the SharedObject mechanism, because we knew it would be the hardest to get right. After about 15 or 16 iterations, though, we think you'll like what we've come up with.

The SharedObjectConnection establishes a remote SharedObject using the getRemote() method of the SharedObject class when you instantiate the connection. If you provide a name parameter to the SharedObjectConnection's constructor, it will use that as the name of the remote SO. If you don't pass the parameter, it uses a default value.

Once your connection is established, you create Slot objects for each individual data repository you want to use in your application by calling the SharedObjectConnection's addSlot() method. You can then add event listeners to each of these slots, listening for the five different message types for a SlotEvent. Let's take a quick look at what those are:

Event	Purpose
`SlotEvent.CHANGE`	The data on the remote Slot has changed, and the client is being notified of that change.
`SlotEvent.SUCCESS`	The client successfully published new data to the remote Slot.
`SlotEvent.REJECT`	The client's requested change to the remote Slot was rejected, either by validation or because another change took precedence.
`SlotEvent.DELETE`	The attribute the client requested from the remote SharedObject was deleted.
`SlotEvent.CLEAR`	Either you have successfully connected to a remote shared object that is not persistent on the server or the client, or all the properties of the object have been deleted

For most simple applications, including the demo we're writing, we'll only focus on the first two events, CHANGE and SUCCESS. Slots have a data property, and when you set that data property, they automatically send, or sync, the new value to the remote server.

PLAYING THE SLOTS

Let's set up our slot for our simple chat application:

```
public function initSlots(event:SharedObjectEvent):void
{
     sChat = soConn.addSlot("chatDemo");
     sChat.addEventListener(SlotEvent.CHANGE, updateChatLog, false, 0, true);
     sChat.addEventListener(SlotEvent.SUCCESS, clearAndUpdate, false, 0, true);
}
```

Once we have our event listeners in place, we can start to use our Slot. Let's look at what sending a simple update to our Slot would look like:

```
public function sendChatMessage(user:String, message:String):void
{
     var chatMsg:Object = new Object();
     chatMsg.user = user;
     chatMsg.message = message;
     sChat.data = chatMsg;
}
```

In the sendChatMessage method, we create an object, add a user and message property to it, and set the Slot's data property to our object. Because of the automatic syncing, that's it!

So, what's left to do? Well, we have to write a couple more methods. Let's assume that the user interface has been created already, and there are three TextFields. One has an instance name of usernameInput, the second is messageInput, and the last is named chatLog. We have two Buttons, one named sendButton, and one named setUsernameButton. We simply add a few more variables to our document class that match those instance names and types, and write our remaining methods.

The first is handleSetUsername, which sets a username variable to the text content of the usernameInput TextField (or TextInput). We attach that as a listener to setUsernameButton's click event. We also create a method called handleSendButton, which calls sendChatMessage, passing this.username and the text content of the messageInput TextField as it's two parameters, user and message.

We also need to create the handler methods for our two SlotEvent listeners, clearAndUpdate() and updateChatLog(). In clearAndUpdate(), we call updateChatLog passing the SlotEvent object, and then set the text property of messageInput to an empty string. In updateChatLog, we concatenate the existing text in the log with the username and message from the event.data property, which had been set to our generic object earlier.

```
public function handleSetUsername(event:MouseEvent):void
{
     this.username = usernameInput.text;
}

public function handleSendButton(event:MouseEvent):void
{
     sendChatMessage(this.username, messageInput.text);
}

public function clearAndUpdate(event:SlotEvent):void
{
     updateChatLog(event);
     messageInput.text = "";
}

public function updateChatLog(event:SlotEvent):void
{
     chatLog.text = chatLog.text + event.data.user + ": " + event.data.message + "\n";
     chatLog.scrollV = chatLog.maxScrollV;
}
```

Now you should have a working chat room, running on Synx! You can also set the data property to other objects with more complex data, just remember to keep it as small and agile as possible. With that, we've explored the three major features of Synx, and all we have left is to talk about what's coming next.

PLATFORM ROADMAP

It's actually more like a hand-drawn map.

6

Here's a glimpse at where we're going from here. Let us know what you think!

Thank you for taking the time to evaluate the Synx Platform. We hope that it makes you as productive as it's made us, and that you find as much joy in coding for FMS with it as we have.


INFLUXIS