

1. LiveCycle Collaboration Service	2
1.1 01 Getting Started	2
1.1.1 1.1 What is the LiveCycle Collaboration Service?	2
1.1.2 1.2 Set up your developer environment	3
1.2 02 Architecture	8
1.2.1 2.1 Terminology	8
1.2.2 2.2 Architecture overview	9
1.2.3 2.3 Walking through the layers	10
1.3 03 Messaging and Permissions	13
1.3.1 3.1 Shared Models	13
1.3.1.1 3.1.1 Shared model requirements	14
1.3.1.2 3.1.2 Logical destinations	14
1.3.1.3 3.1.3 NodeConfigurations	15
1.3.1.4 3.1.4 UserRoles	15
1.3.1.5 3.1.5 MessageItems	16
1.4 04 Developer tools	17
1.5 05 Room Console	20
1.5.1 5.1 Room management and monitoring	20
1.5.2 5.2 Managing accounts and rooms	21
1.5.3 5.3 Managing room settings	25
1.5.4 5.4 Managing room collections and nodes	29
1.5.5 5.5 Viewing logs and service usage	33
1.6 06 Deploying Applications	34
1.6.1 6.1 Deployment Scenarios	34
1.6.2 6.2 Provisioning rooms	35
1.6.3 6.3 Templating rooms	37
1.6.4 6.4 Authentication setup	38
1.7 07 Server to Server APIs	40
1.7.1 7.1 About Server to Server functionality	40
1.7.2 7.2 High level workflow	41
1.7.3 7.3 Revisiting the chat example	43
1.7.4 7.4 Server to Server API Reference	43
1.7.4.1 7.4.1 Calls	44
1.7.4.2 7.4.2 Queries	47
1.7.4.3 7.4.3 Hook APIs	49
1.8 08 Audio, WebCam, and Screen Sharing	51
1.8.1 8.1 Audio	52
1.8.2 8.2 Web Camera	53
1.8.3 8.3 Screen Sharing	54
1.8.4 8.4 RTMFP vs RTMP	55
1.9 09 Peer-to-Peer Data Messaging and AV Multicasting	56
1.10 10 Recording and Playback	57
1.10.1 10.1 Overview	57
1.10.2 10.2 Repository Setup	59
1.10.3 10.3 Starting and Stopping Recording	61
1.10.4 10.4 Building a Playback Application	62
1.10.5 10.5 Authenticating Playback Applications	63
1.10.6 10.6 Administration	64
1.11 11 Tutorials	65
1.11.1 11.1 Building your first application	65

LiveCycle Collaboration Service

1. [Getting Started](#)
2. [Architecture](#)
3. [Messaging and Permissions](#)
4. [Developer tools](#)
5. [Room Console](#)
6. [Deploying Applications](#)
7. [Server to Server APIs](#)
8. [Publishing and Subscribing to Audio and Video](#)
9. [Peer-to-Peer Data Messaging and AV Multicasting](#)
10. [Recording and Playback](#)
11. [Tutorials](#)

01 Getting Started

- [What is the LiveCycle Collaboration Service?](#)
- [Set up your developer environment](#)

1.1 What is the LiveCycle Collaboration Service?

The LiveCycle Collaboration Service (LCCS) is a platform-as-a-service that allows developers to easily add real-time social capabilities to rich Internet applications (RIAs) as well as other new and existing applications. Its goal is to allow you to quickly build multi-user, real-time collaborative (RTC) applications by providing a mature set of easy to understand components, tools, and services.

Comprised of both Flex-based client components, an SDK, and a hosted services infrastructure, LCCS provides a quick track to success. Since Adobe hosts the service, issues like deployment, maintenance, and scalability are taken care of for you.

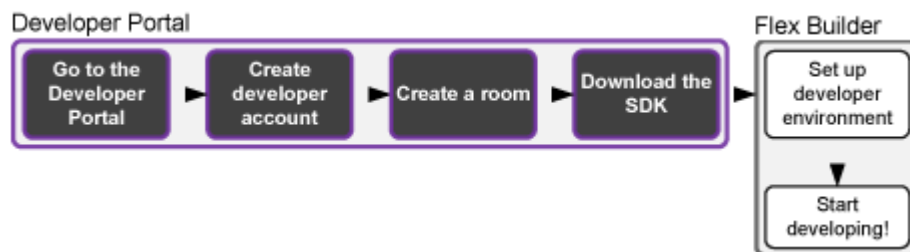
- [Fast track path to development...](#)
- [The slower "Let's get oriented" path to development](#)

Fast track path to development...

LCCS provides a quick and easy path to development. At its most basic level, you simply go to the Developer Portal to get your account, first meeting room, and the SDK. From there, you can start developing. In fact, novices could be running working examples in their own meeting room within ten minutes. Using the default pods (pre-built application components with a simple user interface) you could be running your own working code in about the same time.



Of course, a little more knowledge can speed your development even further. Go to [The slower "Let's get oriented" path to development](#).



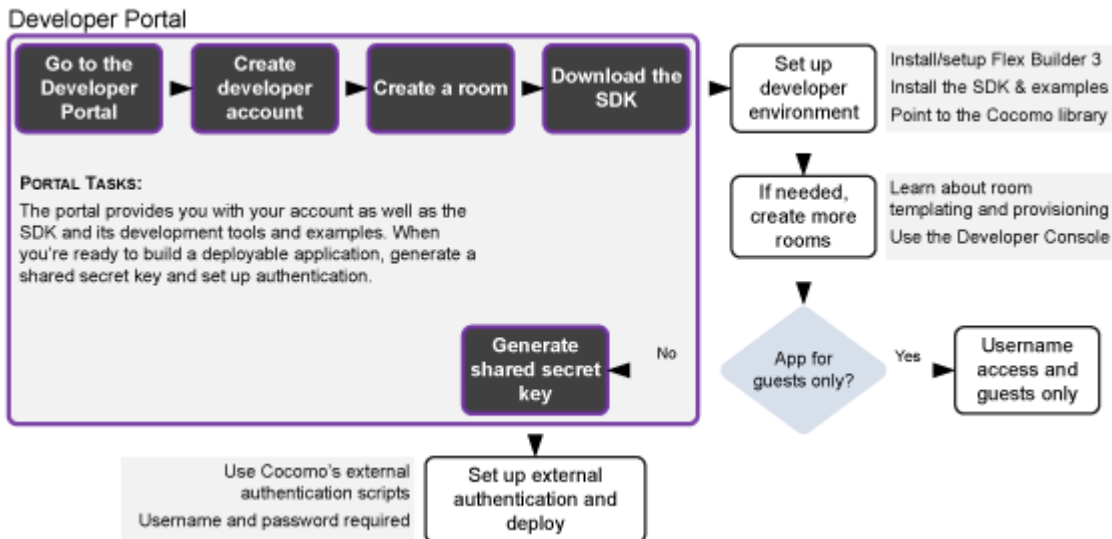
Workflow: Getting started simplified

The slower "Let's get oriented" path to development

In the long run, a little knowledge saves time, so take a moment to get the big picture. There are a couple of tools you'll want to know about and use right off the bat, so the following (and slightly longer) track will likely facilitate your development efforts:

1. [Set up your developer environment](#)
2. Use the SDK Navigator to:
 - Watch the instructional video(s).
 - Run the demo application and view its source code.

- Peruse the tools.
- Go through the Developer Guide's tutorial.
 - Set up and run a few examples. You'll need to enter your Adobe ID username and password as well as one of your room URLs. Notice that some of the examples contain less than 20 or 30 lines of code. DefaultPods, for example, provides fairly substantive features that could be written in minutes.
 - Start developing your own custom application. You'll likely use the following:
 - LCCS APIs to provide the application functionality and leverage the LCCS hosted service. Flex and LCCS UI components to build the user interface.
 - Authentication mechanisms. You should learn the difference between the authentication options pre and post deployment: During development you'll use your Adobe ID. Deployed applications will either only allow guests that log in with a username, or they will depend on external authentication to achieve enterprise-class authentication which leverages your existing systems.



Workflow: Getting started

1.2 Set up your developer environment

To get started, do the following:

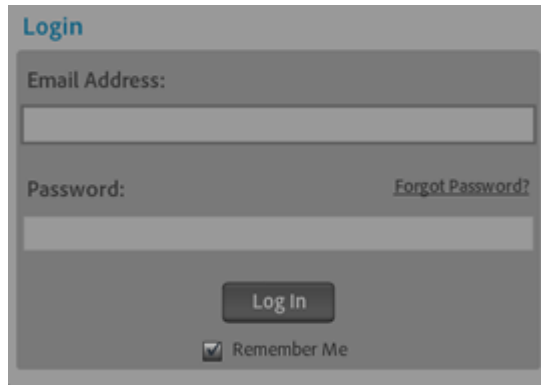
- Create a developer account
- Log in to the Developer Portal
- Create a room
- Generate a shared secret key
- Download the LCCS SDK Navigator
- Verify your IDE compatibility
- Install the SDK
- Installing a debug version of the Flash Player
- Verifying your Flash versions
- Project settings

1. Create a developer account

i If you already have a developer account, go to 2. Log in to the Developer Portal.

Access to LCCS's service and SDK is granted through the Developer Portal. You'll need an Adobe ID and a developer account. The developer account is a unique account name that is prepended to your room URL. It **MUST** be unique from any ConnectNow account name you have, if any. Keep it simple, make it intuitive, and follow standard URL character rules.

- Go to the Developer Portal <https://collaboration.adobelivecycle.com>.
- Choose **New Dev? Sign up!**



Login

Email Address:

Password: [Forgot Password?](#)

Log In

☒ Remember Me

Developer Portal: Logging in

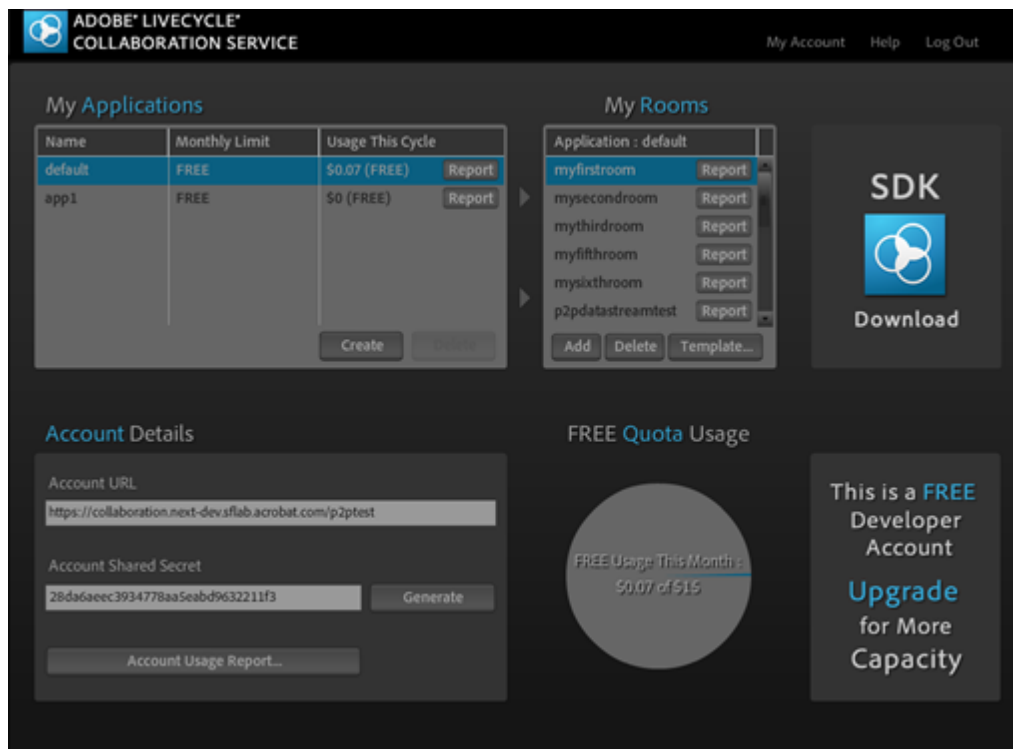
1. Fill in the information in the Create a Developer Account dialog.
2. Choose **Create**. You will be automatically directed to the Developer Portal.

2. Log in to the Developer Portal

The Developer Portal provides a way to monitor account and room usage, create new rooms, generate a shared secret (if you're using external authentication) and perform other basic tasks.

To log in:

1. Go to the Developer Portal <https://collaboration.adobelivecycle.com>.
2. Enter your Adobe ID and log in.



ADOBE LIVECYCLE COLLABORATION SERVICE My Account Help Log Out

My Applications

Name	Monthly Limit	Usage This Cycle	
default	FREE	\$0.07 (FREE)	Report
app1	FREE	\$0 (FREE)	Report


[Create](#) [Delete](#)

My Rooms

Application : default

myfirstroom	Report
mysecondroom	Report
mythirdroom	Report
myfifthroom	Report
mysixthroom	Report
p2pdatastreamtest	Report

[Add](#) [Delete](#) [Template...](#)



SDK
Download

Account Details

Account URL
<https://collaboration.next-dev.sflab.acrobat.com/p2ptest>

Account Shared Secret
[28da6aee3934778aa5eabd9632211f3](#) [Generate](#)

[Account Usage Report...](#)

FREE Quota Usage

FREE Usage This Month : **\$0.07 of \$15**

This is a **FREE** Developer Account

[Upgrade](#) for More Capacity

Developer Portal

3. Create a room

A room is LCCS's name for a virtual location on the service which is represented in your applications and to others as an URL. Clients connect to the room and send and receive messages (data) to other present clients. Think of rooms as meeting places at some URL. While you can create rooms programmatically or via the Room Console, the Developer Portal is a good place to start. Create one now so that you can run the SDK examples and test example code. Your account is initially provisioned with a default room named "myfirstroom".

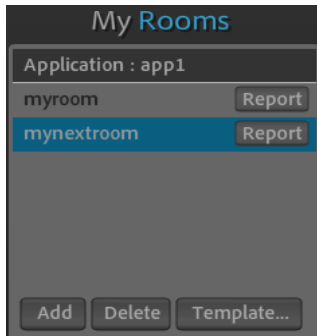
To create a room:

1. In the My Rooms panel, choose **Add**.
2. Provide a room name.



The name is appended to your room's root URL, so keep it simple and follow standard URL character rules. The URL format is: <LCCS root URL>/<account name>/<room name>. The application name is not used in the room URL.

3. Choose **Save**. The service automatically creates an empty room based on the default application.



Developer Portal: Applications and rooms

4. Generate a shared secret key



A secret key is needed only when you are ready to deploy an application that uses external authentication, thus leveraging your organization's existing authentication infrastructure (e.g. LDAP). You can create your key now or later.

From the Developer Portal you can generate an account shared secret (or create a new one if your existing secret is compromised).

During development and testing, users (usually just developers) either enter as guests with just a username, or enter as a full user with both a username and password. LCCS only understands Adobe ID passwords; therefore, it is the de facto case that when someone enters both a username and password, both must belong to an Adobe ID.

At deployment time however, while some developers will create applications that only need to recognize guest users (who can enter any username), most applications will require some type of authentication. There are two requirements that necessitate the solution that a secret key solves:

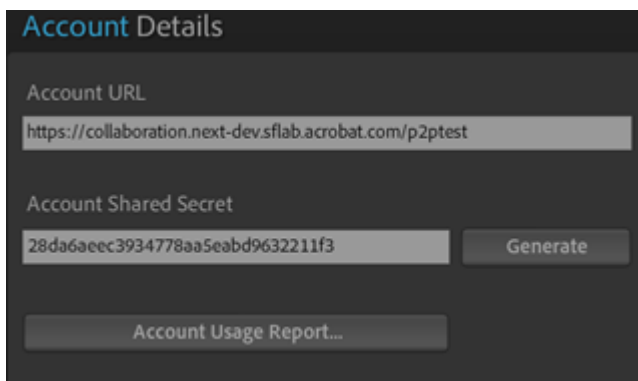
- Adobe does not want to know, and you would not want Adobe to know, your users' passwords.
- You will want to create and manage authentication credentials using your own tools and infrastructure (for example, your databases and LDAP servers).

With LCCS's provided external authentication scripts, the shared secret key allows you to take in a username and password, authenticate it on your own system, and then pass it to your client application. The key enables automatic authentication, authentication privacy, and single sign on. When you're ready to set up external authentication, see [Authenticating on your own systems](#).

If you have an existing key, creating a new key replaces the old key. Only one key is available for each user with developer credentials; that is, for each account can be associated with one key. Your key is always displayed at the top of the portal's user interface.

To generate a key:

1. Go to the Account Details panel.
2. Choose **Generate**.
3. When the confirmation dialog appears, choose **Yes**.



Developer Portal: Secret key generation


5. Download the LCCS SDK Navigator

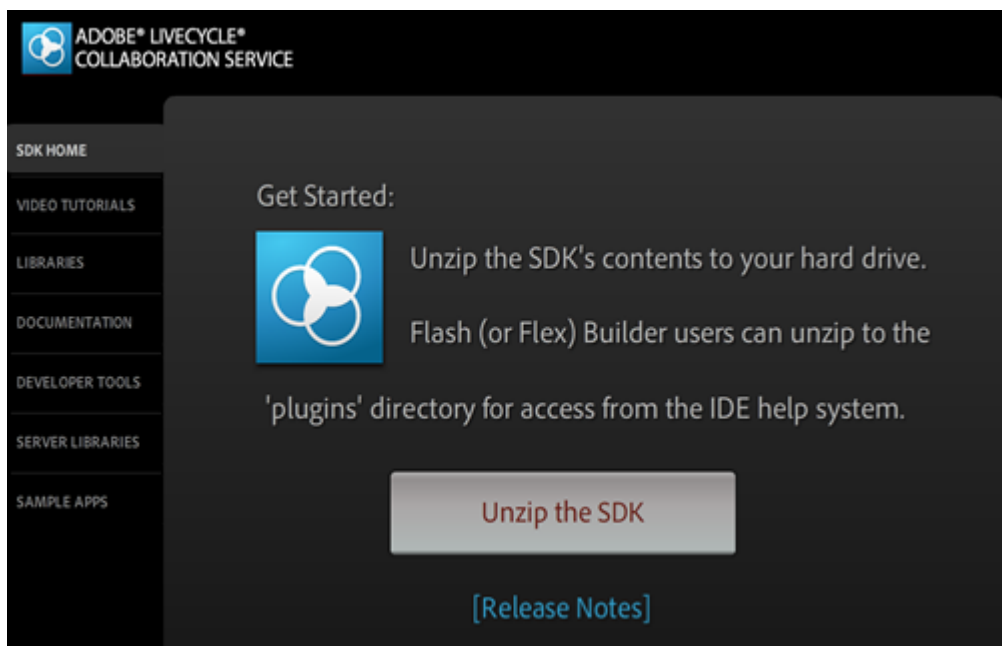
The SDK Navigator contains the zipped SDK, links to instructional videos, tools, sample applications, and other resources to put your project on the fast track to development.

- **Installable SDK components:** SWCs, source code, examples, scripts, and documentation.
- **Links to video tutorials**
- **Running demo applications**
- **Tools:**
 - **Room Console:** Provides a UI-based way monitor and manage templates, rooms, and users.
 - **Local Connection Server:** Enables offline development and testing of non-streaming components so that you can run your applications without connecting to the service.

To do so:

1. Choose **Download the SDK**.
2. Navigate through the installer to install the application to any location.

 You can now close the portal. Either write down your account URL and shared secret, or return here when you start developing. You'll need your room URL to do almost anything. You'll need your shared secret when you deploy your application.




SDK Navigator home page

6. Verify your IDE compatibility

LCCS currently supports:

- Flex Builder 3
- Flash CS3 for the Flash-only SDK.
- Flash CS4
- Flash Builder 4

 For any product you use, it's always a good idea to install the latest version and download the latest updates. Flex Builder 2 is not supported.

7. Install the SDK

These instructions install the SDK to Flex Builder's plugins/com.adobe.lccs directory so that you can browse the documentation from Flex Builder's help system. The sample applications, SWC libraries, and other resources reside there as well. However, you can choose a location of your choice.

The SDK contains a number of items you will unzip and install to a convenient location where you can access them later, including:

- **Source Code:** Some source code is provided to assist with debugging.
- **Compiled Libraries (SWCs):** Precompiled SWC files, including Flash Player 9/10/10.1 for both Flex and Flash.



One of your first decisions should be whether or not to use the player 10 SWC files. One of the primary advantages of these files is their support for RTMFP. For details about RTMFP, see [RTMFP vs. RTMP](#).

- **Server Scripts:** Provided server integration scripts allow you to automatically connect to LCCS services and manage accounts as well as list, create, and delete rooms and templates. Languages include e Python, Ruby, Cold Fusion, Java, PHP, and Groovy.
- **Sample Applications:** Sample applications provide working code you can install and run. Refer to the LCCS SDK Navigator's Sample Applications directory for more demos.
- **Documentation:** Includes the developer guide, release notes, and readmes.

To install the SDK:

1. Start the LCCS Navigator.
2. Install its zipped SDK resources to any location.
 - a. On the Home tab, choose **Save**.
 - b. Browse to Flex Builder's plugin directory. For example:
 - **Windows:** C:\Program Files\Adobe\Flex Builder 3\plugins\
 - **Macintosh:** /Applications/Adobe Flex Builder 3/plugins/
 - c. Choose **OK**.
 - d. Restart Flex Builder.

8. Installing a debug version of the Flash Player

Optional.

The debug version of the Flash player may facilitate application development. Various versions of the debug player are available depending on your platform, whether or not you use Flash Pro, and so on.

The following example describes one possible scenario.

To install the debug version of the Flash player when you are not using Flash Pro:

1. Uninstall existing versions of the Flash player as described in http://kb.adobe.com/selfservice/viewContent.do?externalId=tn_14157&slcid=1.
2. Install the requisite version of the debug player from <http://www.adobe.com/support/flashplayer/downloads.html>. For example, Windows Flash Player 10 ActiveX control content debugger for IE.
3. Verify the version as described in [9. Verifying your Flash versions](#).

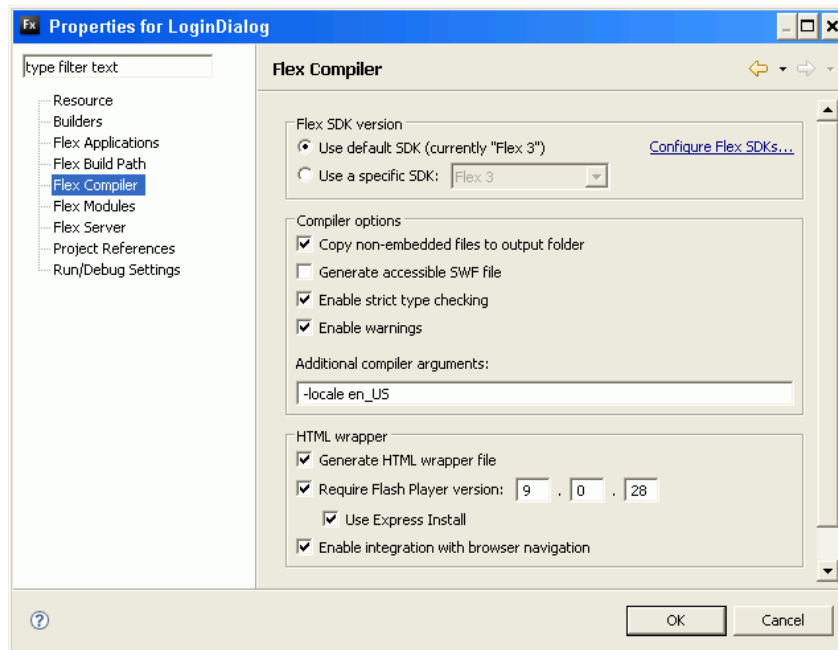
9. Verifying your Flash versions

These instructions use Flex Builder. Use steps which are applicable to your IDE.

If you're in doubt about the versions of Flash you are running, check them. To check the version your browser uses, right click on any Flash item and read the following: *About Adobe Flash Player <version>*.

To check the runtime version Flex Builder uses, do the following:

1. Choose **Project > Properties**.
2. Choose **Flex Compiler**.
3. Verify the Flash version in the HTML wrapper panel.



Flash player version settings

10. Project settings

For each project, point to the appropriate SWC and source code (for debugging) as described in [3. Start a new project](#) in the [Building your first application](#) tutorial.

RTMFP and Flash Player 10 SWC files

One of your first decisions should be whether or not to use the Flash Player 10 SWC files. One of the primary advantages of these files is their support for RTMFP. For details about RTMFP, see [RTMFP vs. RTMP](#).

02 Architecture

This section is intended as a companion to the LCCS API Reference which provides language-level descriptions of the materials presented here. The intention is to introduce the important pieces, how they fit together, and where developers should begin in order to start playing with the SDK.

- [Terminology](#)
- [Architecture overview](#)
- [Walking through the layers](#)

2.1 Terminology

LCCS uses terms like "session," "room," and "template." Here's a quick guide:

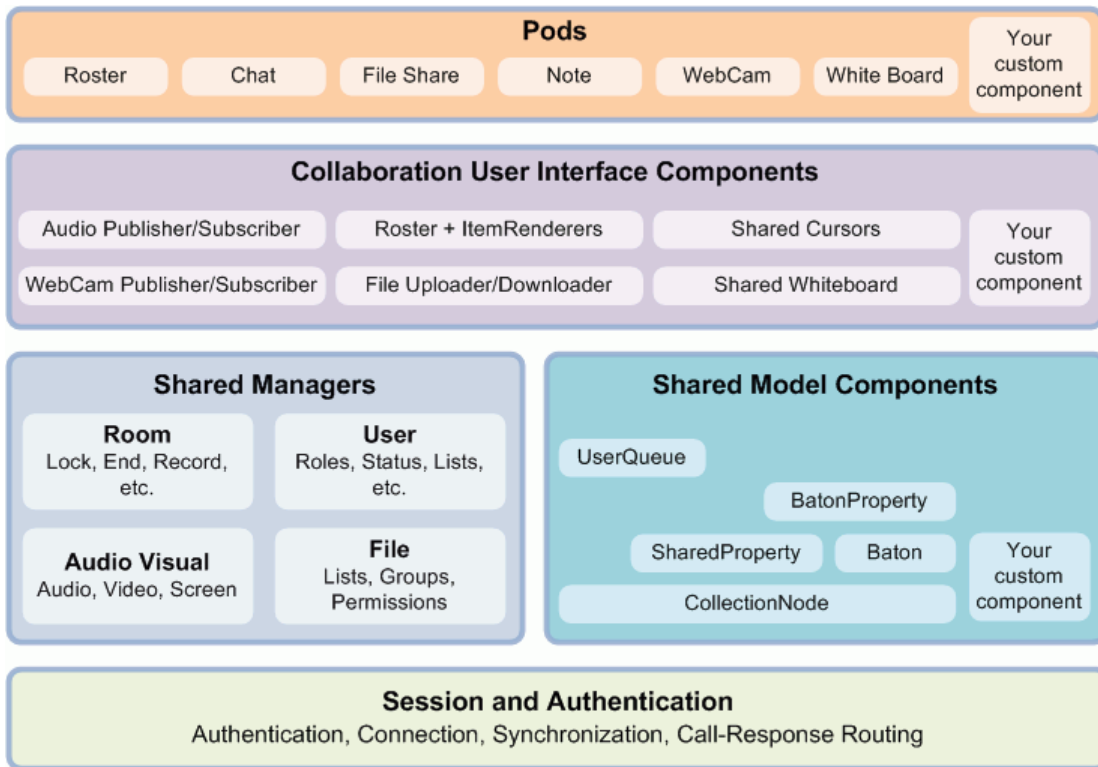
Term	Description
application	An application is a means of organizing sets of rooms on the service. Applications set billing limits for those rooms and provide templates for new rooms created in those applications.
room	A room is a virtual location on the service to which clients can connect. The room is represented in your applications and to others as a URL. Someone (or some client) in a room can send and receive messages (data) to anyone else in the room. Think of rooms as meeting places at some URL. Applications require a room to enable people to meet and exchange information. You can create rooms programmatically, via the Room Console, or via the Developer Portal.
session	A session is a client's connection to a room. It exists as long as the connection exists. A session exists on a client.
application template	An application template is a saved room configuration which includes collectionNodes, nodes, configurations, items, and room settings. Rooms can be saved as application templates so that you can create new rooms with pre defined elements. You can create templates via the Room Console or via the RoomTemplater API.
collection node	As the top-level element in the node hierarchy, collection nodes are the primary destination for room messages. In general, the model classes in sharedModel, pod, and sharedManager (for example, a chat pod or the UserManager) instantiate and manage their own, unique collection nodes which represent them on the root of the room.

node	Collection nodes are subdivided into one or more nodes. Nodes publish message items which may be stored on the service. By default, a node maps roughly to a set of APIs within its collection node model with the same permissions and storage. They support node configurations which describe settings for working with message permissions and storage policies as well as the setting of user roles on individual nodes
provisioning	Creating a new room and optionally furnishing a room with the needed components and settings.
templating	Saving a room's components, settings, and details as a template for an application on the service. The template can be used to create new rooms automatically.
room settings	Every room has basic settings for bandwidth, auto-promoting users to publishers and so on.

2.2 Architecture overview

LCCS's primary principles and goals include the following:

- Lowered barrier to entry into real time collaboration for both end users and developers.**
 End users typically must install an application in order to participate in anything resembling real time collaboration (RTC). LCCS eliminates such resource intensive efforts by providing RTC that leverages the web via Flash/Flex. Sign up and participation is simple and seamless.
 For developers, building existing RTC applications have been prohibitive from a cost, operational, and complexity standpoint. Adobe's RTC service allows you to economically leverage existing expertise and scale efficiently by off loading operations to the people and infrastructure with the resources to operate on a global scale. Since the backend is not hosted on your end, Adobe's services reduce complexity and are available on a pay-for-what-you-use model.
- A simple architecture where applications reside on the client and services are hosted on the server.**
 LCCS embraces software as a service and therefore enables developers to build collaborative applications completely on the client. These components expose a runtime model for real-time collaboration which is abstracted from server logic. Adobe's hosted services are the primary backend for the real-time aspects of LCCS-enabled applications as well as the channel through which LCCS components communicate. This architecture provides new opportunities for rich collaborative applications, from embedding RTC functionality into existing applications with their own backends to completely new applications that take Flash and Flex far beyond where they are today.
- Build permissions and security into all collaboration.**
 Without permissions, RTC can quickly descend into chaos. Not every person in a collaborative application has the right to perform every action, and those with application "owner" rights need the ability to manage other users' access to features. Every component provided by LCCS has a built-in notion of authenticated identity, permissions, and security. Rights for components are manageable by application owners; for example, the teacher in a virtual classroom can "pass the baton" to allow a student to ask a question using VOIP, or the host of a seminar can demote a nuisance user in a chat to disable further interruption. All communication is authenticated with permission checks at the client and server to effectively prohibit unauthorized access or action.
- Provide high level building block components as well as low level model components.**
 Many use cases for the LCCS SDK are supported with a high level set of pod components such as roster lists, VOIP publishing user interfaces, and whiteboards. These components allow simple assembly of rich applications with little low level coding. For more customized or advanced use cases, LCCS supports the creation of new components with a set of low level, shared model components useful for communicating the shared application state. All LCCS components support extension through inheritance and composition.

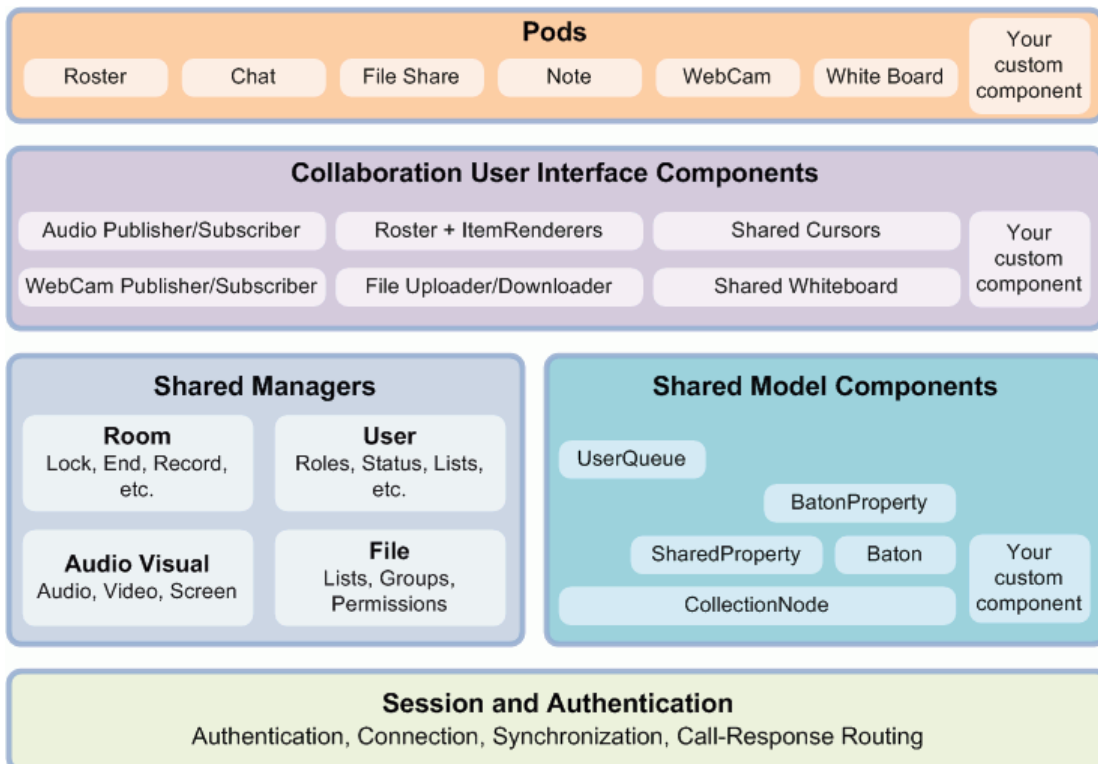


Architecture Diagram

2.3 Walking through the layers

The diagram below illustrates the basic organization of the classes within the LCCS API. It's a "protocol stack"-oriented diagram: low level classes on the bottom and high level on top, with each successive layer dependent on the layers beneath it. The following section deals with the stack starting from the bottom up.

- Sessions and authentication
- Shared model components (`com.adobe.rtc.sharedModel`)
- Shared managers (`com.adobe.rtc.sharedManagers`)
- Collaboration UI components (`com.adobe.rtc.collaboration`)
- Pods (`com.adobe.rtc.pods`)



Sessions and authentication

`com.adobe.rtc.session` and `com.adobe.rtc.authentication`

All LCCS applications require a foundation session and authentication layer. These layers are responsible for:

- Connecting to an appropriate server and maintaining that connection (see `com.adobe.rtc.session.ConnectSession`).
- User authentication with that server (see `com.adobe.authentication.AdobeHSAuthenticator`).
- Physical routing of all calls and responses to and from the service.
- Reconnecting and failover when needed.

The actual set of calls and responses to the service is designed to be very thin; the session layer supports only the generic set of RMI calls required to support messaging. In this way, the session layer is easily implemented for a variety of platforms and has a smaller surface area for exploit.



Application code built using LCCS is completely abstracted from the choice of session management thereby assuring that the code above the session layer be very portable.

Shared model components (`com.adobe.rtc.sharedModel`)

This group consists of low level, model-oriented components without a user interface that help developers build a shared state within their components or application classes. These classes are used within LCCS for sharing the model tier of the shared managers, collaboration UI components, and pods. Permissions management is supported as well as specific workflows. For example, baton passing prevents users from making conflicting changes to any part of a shared model.

For more information about building your own collaboration components or pods, see the following:

- [03 Messaging and Permissions](#)
- LCCS API Reference in the SDK: In particular, `CollectionNode` is the foundation of these classes. For details, see `com.adobe.rtc.sharedModel.CollectionNode`.
- For an example, see the `sharedModel` example in the SDK's `sampleApps` directory.



Developers are welcome and encouraged to create new shared model components.

Manager	Description
Baton	Baton is a model class which provides a workflow between users.
BatonProperty	BatonProperty is a model component which manages a property of any type that only one user can edit at a time.
CollectionNode	CollectionNode is the foundation class for building shared models requiring publish and subscribe messaging.
SharedCollection	SharedCollection is a simple <code>ListCollectionView</code> which is shared across the LCCS services.
SharedObject	SharedObject is used to store data in an unordered hash (key-value) across the LCCS services; elements can only be accessed using its key.
SharedProperty	SharedProperty is a model (and GUI-less component) that manages a variable of any type and which is shared amongst all users connected to the room.
UserQueue	UserQueue is a model class that can be used to create and manage queues of users who are making requests.

sharedModel example: Baton

A baton is simply a signifier that allows the assignment or changing of roles based on whether a room member is holding it or not. The Baton class provides a workflow between users by tracking the holder of a given resource and providing the APIs for grabbing, releasing, and passing the baton to others. Users with an owner role always have super powers with respect to the baton, while users with a publisher role must wait according to the grabbable property:

- If the baton is set to grabbable, they may grab the baton as soon as it is available (since it will then have no controller).
- If the baton is not grabbable, the owner must explicitly pass the baton to someone else.
By default, a baton will timeout in five seconds and be released. This timeout can be adjusted in the constructor and extended during use of the resource in question using `extendTimer`.

Note that users with an owner role may adjust the roles of other users relative to the baton using `allowUserToGrab` (which makes that user a publisher) and `allowUserToAdminister` (which makes that user an owner).

Shared managers (`com.adobe.rtc.sharedManagers`)

Shared managers provide the four pillars of every LCCS application: these are singleton manager classes that are used to access and

manipulate the shared state of the application as a whole. The APIs for all sharedManagers are asynchronous; that is, changes are not reflected in the client until the server has validated them and returned the result to all clients. At that point, the manager dispatches the appropriate event.

Manager	Description
RoomManager	Provides APIs to access the room state, end the room session, set its privacy settings, manage its bandwidth, and so on.
UserManager	Provides APIs to access and manage the users in the room, promote or demote users, query for lists of users, and accept or deny users.
FileManager	Provides APIs to access and manage the files associated with the room, including publishing new files and querying for lists of associated files.
StreamManager	Provides APIs to access and manage the AV streams associated with the application, including publishing new streams and querying for lists of streams for consumption.

Like other LCCS APIs, these APIs have permissions management built into them: only users with the requisite permissions may utilize them. To access any of these managers, use the `ConnectSession` component which has a reference to each (for example, `myConnectSession.userManager`).



The Room Console also provides a simple and non-programmatic way to use the APIs of all four of these managers. For details, see [05 Room Console](#).

sharedManager example: RoomManager

For example, the `RoomManager` is responsible for handling all runtime configuration changes to a room. General areas covered by the `RoomManager` include the room's:

- **state:** Open, closed, URL, timeout, and so on.
- **membership settings:** Whether guests must knock to enter, whether users are auto-promoted to publishers and so on.
- **bandwidth settings:** The specified room bandwidth.

Only a user with an owner role may modify room settings. Many of these settings are typically declared the first time a room is created by using the `ConnectSession.initialRoomSettings` property and by specifying a `RoomSettings` object. Each `ConnectSession` handles creation and setup of its own `RoomManager` instance, and the `ConnectSession.roomManager` property is used to access it.



You can configure room settings programmatically or via the Room Console. Some settings are owned by the `RoomSettings` class, and others by the `RoomManager` class. Room settings are saved with any templates created from the room.

Collaboration UI components (`com.adobe.rtc.collaboration`)

Collaboration UI components are multi-user-aware user interface components that can be used to greatly reduce the time it takes to build complex applications and pods. The intention behind this set of components is to provide a simplified API for application development while supporting customization and enhancement through inheritance and composition. Most high level pod components are compositions of these building block components; for example, the `WebcamPublisher/Subscriber` or the `FilePublisher/Subscriber`.

The intention here is to grow the component set over time. Adobe has provided all the lower level classes that are needed.

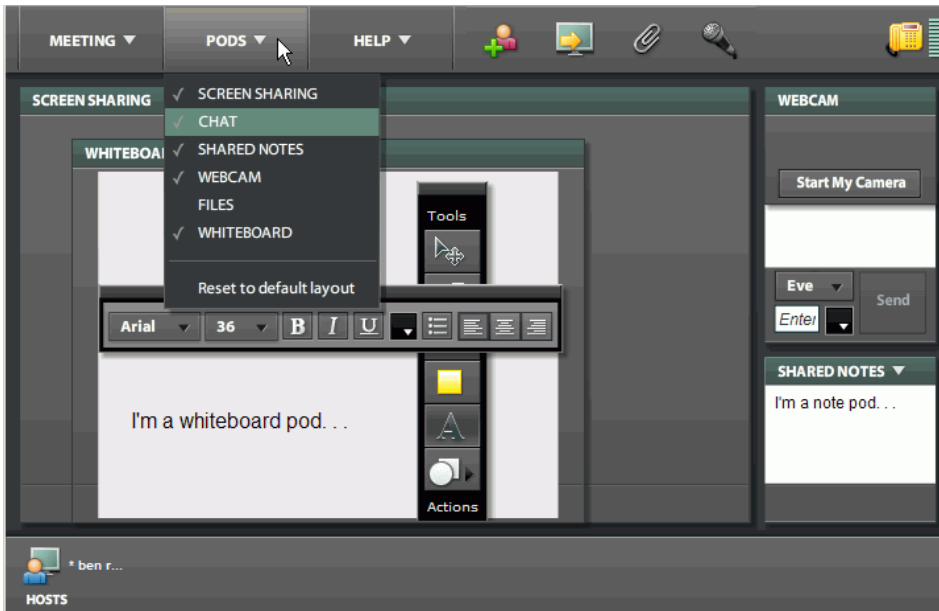


Developers are welcome and encouraged to make new collaboration UI components using various `sharedModels`, their own models, or `sharedManager` classes.

Pods (`com.adobe.rtc.pods`)


Pods are high level, mini-applications which consisting of and provide the most requested functionality that LCCS developers might want to use. They can be used out-of-the box or extended through inheritance to meet unique needs. The current pods include the following:

- **Web camera:** Start and run your camera to share live video.
- **Chat:** Chat with other room users.
- **File sharing:** Share files.
- **Roster:** List participants.
- **Note:** Add meeting notes.
- **WhiteBoard:** Allow one or more users to draw on the virtual whiteboard.



Pods in action

The standard pods provide a rich feature set that enables developers to quickly build a dynamic and highly interactive meeting room. For working examples, run the examples in the SDK or try ConnectNow.

 Other pods are under development, but you can create your own.

03 Messaging and Permissions

Although it's possible to develop real-time-collaboration applications with only the high-level components provided in LCCS, you may also want to create your own collaboration-aware components. For example, you could build applications such as:

- A Poll pod
- A shared image gallery
- Synchronized video playback
- Shared, interactive maps

Before building such collaborative components however, it's expedient to become familiar with the platform's lower-level APIs and understand the way that LCCS treats messaging and permissions.

- [Shared Models](#)

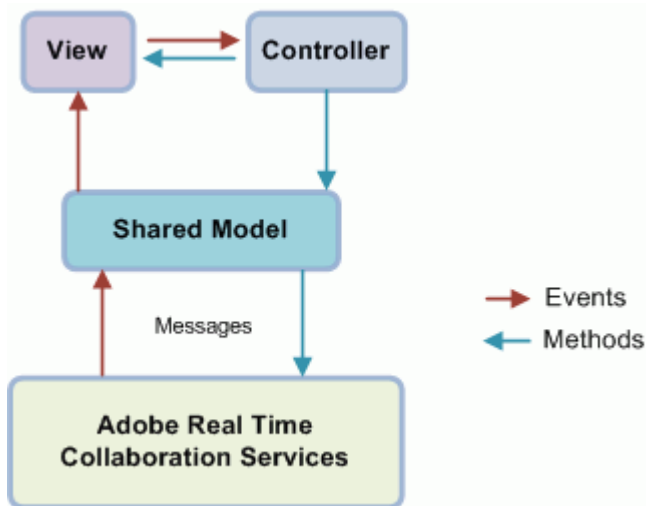
3.1 Shared Models

Developers build custom applications by creating or extending a shared model. Within the model-view-controller (MVC) design pattern, a Model's responsibilities could include:

- Represent an application's state (in this case, the component's state).
- Provide APIs to query and modify that state.
- Notify any listeners when a state change occurs

In LCCS, the "model" concept is augmented by the concept of a "shared model". A shared model adds the following responsibilities:

- It maintains knowledge of the current user's permission to use a given API. For example, the shared model must know whether a user has rights to draw on a whiteboard or add a chat message and it must notify any listeners should the user's permissions change.
- Upon being called via the API to modify its state, a shared model will communicate the potential change by generating a message which it publishes to the service. It does not reflect the change in its state immediately.
- When the service routes an incoming message to the shared model, it interprets the meaning of the message, updates its state to reflect any changes, and notifies any local listeners of the change.



Shared model design pattern

The shared model design pattern shows the basic workflow. A method call from the controller is passed via messaging to the service and is returned from the service via a message event. In this sense, a change coming from the current (local) user is received in the same way as though it was from a remote user: it must round-trip to and from the service in order to be a valid update to the state. Treating all shared state updates this way simplifies the design of RTC applications.

3.1.1 Shared model requirements

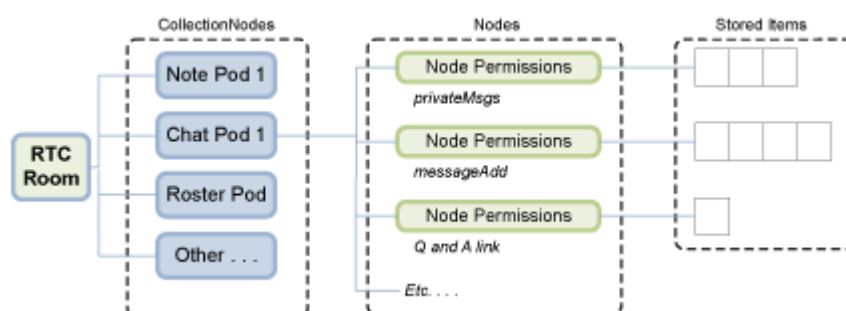
In order to create a shared model that can accomplish the tasks above, we need to provide a way of doing the following:

- **Create a logical destination for a message.**
For example, a chat pod wants to publish to and receive messages from the corresponding chat pods on other users' clients but not messages from the whiteboard or note pods.
- **Provide permissions settings that describe which roles can publish and subscribe messages to that destination.**
For example, by default, anyone can enter a chat message, but only users with a publisher role can draw on the whiteboard. Conversely, the owner of the room may decide, during the collaboration, to lower the role needed to use the whiteboard. Because LCCS applications are social in nature, we want to make sure that these changes can be made dynamically as the social setting demands.
- **Manage a user's role with respect to that destination.**
Similar to the previous case, an owner may wish to promote or demote specific users' roles, either on the room as a whole or for specific components within it. Roles cascade from the "root" of the room to specific components where they may be overridden.
- **Define the storage policy of messages on that destination.**
Not all users arrive in a room at the same time. A chat conversation may have been taking place for hours before a new user joins. In order for that user's chat have an accurately shared model, that user must receive all messages that have been sent previous to the user joining. A shared model must be able to define whether its messages are stored, and how they are stored, in order to handle late-comers. When it is created, the shared model needs to be able to tell when it's synchronized, meaning it has received all stored messages and caught up to the current state in the room. Until a shared model is synchronized, it may not want to accept any commands to change its state.

3.1.2 Logical destinations

The logical destinations within LCCS's shared models are called collection nodes and nodes. Thus, Adobe's RTC rooms basically have a three level hierarchy which consists of the room itself, its collection nodes (which correspond to pods), and the collection node's nodes.

- CollectionNodes
- Nodes



Rooms, nodes, and messaging

CollectionNodes

Collection nodes are the primary level of destination hierarchy in a LCCS room. In general, each shared model (for example, a chat pod or

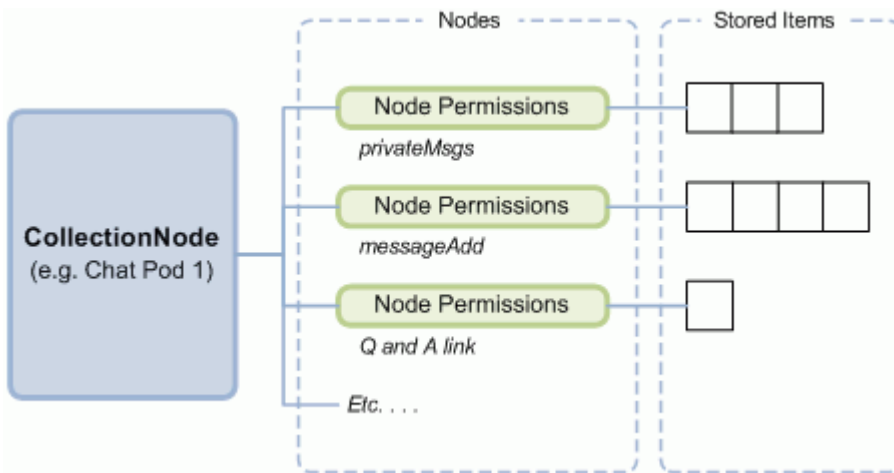
the `userManager`) is represented by one collection node on the root of the room, and any shared model class likely instantiates and manages its own collection nodes.

As described in the LCCS API Reference, the primary class developers use within their shared models is `com.adobe.rtc.sharedModel.CollectionNode`. Note that only room owners are allowed to add new collection nodes to a room or modify existing ones.

Nodes

As the name implies, collection nodes are subdivided further into a set of Nodes used for messaging. Each collection node contains multiple Nodes. Each of these publishes `MessageItems` which may be stored on the service. You can view all of the collection nodes, their nodes, and message items in the Room Console. The Console provides a intuitive interface for adding, deleting, and editing each of these items. This transparency into your data also facilitates debugging.

Just as a collection node maps to a shared model, a node within that collection node roughly maps to a set of APIs within that model with the same permissions and storage. For example, the chat pod may have one node for which all users may publish and access (i.e., the public chat) but which also require the ability to send a private message. In practice, it might be common for users to want to send a message to the room owners only. In this case, you would create a separate node to which anyone could publish but only owners could access. This permissions validation for messages is performed twice: once in the client framework and once on the service itself.



Collection node and its nodes

Nodes have the following functions:

- They support `NodeConfigurations`. These describe the node settings for working with message permissions and storage policies.
- They allow the setting of user roles. `UserRoles` describe the a user's permission's level with respect to that node.
- They may publish, subscribe to, and store `MessageItems` on the service according to their `NodeConfigurations`. Users who come late to a room session will receive any stored messages before the collection node is said to be synchronized.

3.1.3 NodeConfigurations

`NodeConfigurations` are settings applied on a node-by-node basis. Every node has a `nodeConfiguration`, typically the default settings. `NodeConfigurations` are represented by the `com.adobe.rtc.messaging.NodeConfiguration` class and are used in the methods of the `CollectionNode` API.

`NodeConfigurations` are usually static. While they are only set up at the time of node creation, they may also be modified dynamically. In either case, it's important to note that only owners of a given collection node may add, remove, or set `NodeConfigurations` on a node.



For more details, refer to the *LCCS API Reference*; in particular: `com.adobe.rtc.messaging.NodeConfiguration` and `com.adobe.rtc.sharedModel.CollectionNode`.

3.1.4 UserRoles

One of the LCCS's basic tenets is that every messaging call made by any client requires permissions validation. A user role is a numerical expression of a user's permission's "level" for a given room element. Elements of a room include all of the elements in our three level hierarchy:

- The room itself (sometimes referred to the "root" user role)
- A collection node
- A node within a collection node

User roles for a user cascade down from the room level to its collection nodes and on down to its individual nodes. For example, a user may be a viewer for the room at large but is designated as the note taker for a specific note pod. In this case, the note pod's collection node can be set to have role value for that user which is higher than the user's overall role within the room (such as `UserRoles.PUBLISHER`). Other

users for the pod's collection nodes and nodes would just inherit their role from the room.

A node's `NodeConfiguration` has two specific settings which intersect with user roles: `accessModel` and `publishModel`. Respectively, these values express the minimum user role required to subscribe and publish to that node. This allows hosts to lower or raise the role requirements for node users to perform publish and subscribe actions. Most of the shared Model and collaborative classes have API's for changing both the `publishModel` and `accessModel`.

The default user roles are enumerated on the class `com.adobe.rtc.messaging.UserRoles` and consist of the following:

- **UserRoles.VIEWER:** This is the default `NodeConfiguration` value for `accessModel` which allows subscribing to but not publishing messages.
- **UserRoles.PUBLISHER:** This is the default `NodeConfiguration` value for `publishModel` which allows publishing and subscribing, but does not allow configuration.
- **UserRoles.OWNER:** This is the role typically provided to the host or moderator. Users with this role can publish, subscribe, add and remove both collection nodes and nodes, configure nodes, and set user roles.

`UserRoles.OWNER` is the highest role possible, is able to both publish and subscribe to any node, as well as being the only users who may create and delete nodes and collection nodes, and modify user roles and `NodeConfigurations`. The owner is usually the who can add new collection nodes or new functionality to a room.

While `NodeConfiguration` settings for permissions often stay static, user roles are by nature dynamic - they are first assigned as a user is validated and enters the room. Subsequent changes to user roles on collection nodes and nodes typically occur as a host user grants higher (or lower) role values to users as the result of the situation at hand.



The SDK's CustomRoster example demonstrates how to manipulate user roles.

3.1.5 MessageItems

The basic unit of messaging is the `MessageItem`, described by `com.adobe.rtc.messaging.MessageItem`. A `MessageItem` is the payload of any publish command sent through a node. Publishers publish Items via `CollectionNode*.publishItem`, and receive `MessageItems` by listening to the `CollectionNodeEvent.ITEM_RECEIVE*` event. A `MessageItem` may contain a body containing the type of payload data required as well as other message metadata, such as the time it was published, the publisher's user ID, and so on.



An item can be a single string or number or it can be a complex set of hierarchical classes. See `ComplexObjectTransfer` example for sending complex items as messages.

Publishing an item to a node means that any other user whose client has subscribed to that collection node will receive it provided their user role is higher than the `accessModel` value provided in the `NodeConfiguration` of that node. Private messaging is also allowed by specifying one or more recipient IDs in the `MessageItem`, provided that the node has `allowPrivateMessages` set to true in its `NodeConfiguration`.

- [Synchronization](#)
- [MessageItem storage](#)
- [Managing stored MessageItems](#)
- [MessageItem Expiry and Auto-Retracton](#)

Synchronization

Shared models often need to be able to express whether or not the current user is completely up-to-date with other users in the room. If a node has been configured so that `persistItems` is set to true in its `NodeConfiguration`, the node will store `MessageItems` on the service to allow for late-coming users to receive them and then synchronize to the current state. As it is created, collection node will start with an `isSynchronized` value of false, meaning it has yet to obtain all details from the service.

When a late-coming user subscribes to a collection node, the user receives all nodes and their `NodeConfigurations`, user roles and stored `MessageItems` before being allowed to publish or modify any aspect of the collection node. As it gathers this information from the service, the collection node API will fire early events to communicate this "catching up" of state (all while still having an `isSynchronized` value of false) These events include:

- `CollectionNodeEvent.NODE_RECEIVE`
- `CollectionNodeEvent.ITEM_RECEIVE`
- `CollectionNodeEvent.USER_ROLE_CHANGE`

These events are followed by a `CollectionNodeEvent.SYNCHRONIZATION_CHANGE` event to indicate that the user is now "up to state" with the rest of the users. The order of `ITEM_RECEIVE` events for a given collection node will be the order that those `MessageItems` were published.

MessageItem storage

`NodeConfiguration.persistItems` determines whether messages are stored by the service for late-coming users to retrieve later. However, it's often the case that some message items do not need to be stored in order to capture the full state of the shared model. For instance, users working on a shared note are likely to send the entire contents of the text back and forth in `MessageItems`, so really only the last `MessageItem` is needed in order to know the entire text of the note.

To address this, `MessageItems` are stored on the service according to their `itemID` as an associative array or hashtable. Publishing a `MessageItem` with an `itemID` already stored on that node replaces the existing `MessageItem` stored on the service. In some cases, users should not be allowed to modify items published by one another (for example, chat messages). Setting `NodeConfiguration.modifyAnyItem` to false will ensure that users may only replace their own `MessageItems` for that node on the service. The exception is that users with an owner role may always modify other users' items.

Stored `MessageItems` may be deleted using collection node `{.retractItem}`, which is subject to the same permissions rules as publishing.

Managing stored MessageItems

Because of the simple `itemID` rules for storage, developers may pick the `itemID` strategy needed for their shared models. These are supported by three `NodeConfiguration.itemStorageSchemes`. To automatically use one of these schemes, set the `NodeConfiguration's itemStorageScheme` to one of the following:

- **NodeConfiguration.STORAGE_SCHEME_SINGLE_ITEM:** A "Singleton MessageItem" pattern for nodes. In most nodes, a common pattern is to only remember the last message published to the node (the text within the note, for example). In this case, the application need only choose one `itemID` to have all publishers write to as it will be overwritten on subsequent publishing.
- **NodeConfiguration.STORAGE_SCHEME_QUEUE:** Queues of items. Chats, additive lists, and so on, all need a storage model where every subsequent item is added to a queue of items stored on the server. No `itemID` needs to be passed by the client at publish time as it will be auto-generated on the service in a way that is similar to auto-incrementing a primary key in a database table.
- **NodeConfiguration.STORAGE_SCHEME_MANUAL:** Associative Arrays or Hashtables. If the client is capable of building its own unique `itemIDs` for `MessageItems` and wishes to use this node as an associative array or hashtable, then the service will use whatever `itemIDs` are published from the client for use in storage.

MessageItem Expiry and Auto-Retracton

Stored `MessageItems` often need expiry or automatic retraction to deal with situations where a user isn't there to do so manually. For instance, a user who is abruptly disconnected won't be able to retract a `MessageItem` stating that their audio is being published even though the `MessageItem` is now invalid. `NodeConfigurations` offer settings to manage `MessageItem` expiry:

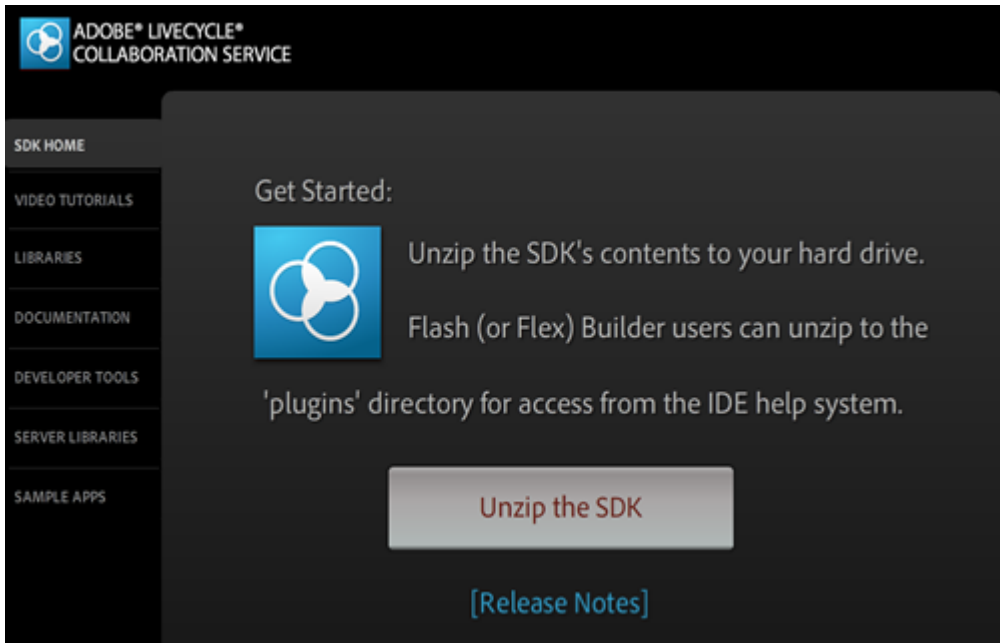
- **NodeConfiguration.persistItems:** If a `MessageItem` is no longer needed immediately after it's published and offers no value to users coming in late to a room, set `NodeConfiguration.persistItems` to false for the node used in publishing that `MessageItem`. The service will then not store `MessageItems` on that node at all. Messages of this sort are said to be transient.
- **NodeConfiguration.userDependentItems:** If a `MessageItem` is only valid for the time its associated user is in the room, set `NodeConfiguration.userDependentItems` to true on a node to automatically retract any items published by that user on that node once that user disconnects. Other users subscribed to that collection node will receive `CollectionNodeEvent.ITEM_RETRACT` events indicating that user's `MessageItems` have been removed from the service.
- **NodeConfiguration.sessionDependentItems:** If a `MessageItem` is only valid for the time the room is active and should be automatically retracted once the room is empty, set this `NodeConfiguration` to true on a node. This can be useful for restoring the room to its original state.

04 Developer tools

Adobe's LCCS provides an ever growing list of resources that will help you develop, test, and monitor your applications. The current list includes:

- [LCCS SDK Navigator](#)
- [Room Console](#)
- [Local Connection Server](#)
- [Source Code](#)
- [Compiled libraries \(SWCs\)](#)
- [Server integration scripts](#)
- [Sample applications](#)

LCCS SDK Navigator



SDK Navigator home page

Location: Download from <http://collaboration.adobelivecycle.com> and install to any location.

The SDK Navigator contains the zipped SDK, links to instructional videos, tools, sample applications, and other resources to put your project on the fast track to development.

- **Installable SDK components:** SWCs, source code, examples, scripts, and documentation.
- **Links to video tutorials**
- **Running demo applications**
- **Tools:**
 - **Room Console:** Provides a UI-based way to monitor and manage templates, rooms, and users.
 - **Local Connection Server:** Enables offline development and testing of non-streaming components so that you can run your applications without connecting to the service.

Room Console

Location: Lives in the SDK Navigator

The Room Console provides a UI-based to monitor and manage templates, rooms, and users. Its user interface provides a quick and easy way to perform tasks that would otherwise have to be performed programmatically. As such, it enables immediate access to some of LCCS's sophisticated features such as templates, setting user roles, viewing audio and camera streams, logging, and so on.

Local Connection Server



The SDK's LocalConnection sample application demonstrates how to build applications that communicate with the LocalConnection Server.

Location: Lives in the SDK Navigator

The Local Connection Server enables offline development and testing of non-streaming components without connecting to the service.

This tool has the following features:

- Applications connect automatically to the running server. No configuration is required.
- All non-streaming components are supported. (excludes Webcam and Audio streaming components).
- The room URL is not used and does not need to be changed.
- Because username and password is not used for authentication, any valid string may be used.
- The server stores no data, so stopping the server cleans the application of data.
- All users are treated as hosts.
- Users can change the roles of others at runtime.

To use the tool, do the following:

1. In your application, change `authentication:AdobeAutheticator` to `authentication:LocalAutheticator`.
2. Start the LCCS SDK Navigator.
3. Choose the Developer Tools tab.
4. Choose **Local Server**.

5. Run your application.

Source Code

Location:

1. for Flex: <SDK install root>/libs/<player version>/src/
2. for Flash: <SDK install root>/libs/flashOnly/<player version>/src/

Some of LCCS's source code is provided to assist with debugging. After setting up your development environment, follow these steps to enable player debugging:

1. In the development UI, go to Project > Properties.
2. Select Flex Build Path.
3. On the Library Path tab, click Add SWC. Choose the path to the LCCS.swc file.
4. Expand the new tree entry for the LCCS.swc file, and select Source Attachment.
5. Click Edit.
6. Navigate to the source path corresponding to your SWC file, and click "OK". The source path ends in /src (for example, C:\Program Files\Adobe\Flex Builder 3\plugins\com.adobe.lccs\libs\playernn\src).

Once the source path is added, you can debug inside LCCS's source code.

The following source is provided:

- **com/adobe/rtc/**: Contains collaborative components and the pods.
- **com/adobe/coreUI/**: Contains stand alone user interface components. These components are used by the various collaborative components along with the shared models to build the application. These components do not use any collaborative feature like collection nodes, messaging, and so on.

Compiled libraries (SWCs)

Location:

- For Flex : <SDK install root>/libs/<player version>/
- For Flash : <SDK install root>/libs/flashOnly/<player version>/

Three precompiled SWCs are provided for both Flash and Flex that support the latest released features:

- player9 provides an LCCS.swc file that supports Flash player 9.
- player10 provides an LCCS.swc file that supports Flash Player 10 features, including improved audio codec, peer-to-peer audio-visual streaming, and others.
- player10.1 provides an LCCS.swc that support all player10 features, plus peer-to-peer data streaming and application multicast for audio-video streaming.



The SWC you choose must match your Flash runtime version as well as the version your browser uses. For details, see [1.2 Set up your developer environment](#).

Server integration scripts

Location: <SDK install root>/serverIntegration/

Provided scripts allow your servers to connect to LCCS services and manage accounts at runtime. These scripts let you:

1. list, create, and delete rooms and templates
2. Manage authentication of your users against LCCS rooms
3. publish and subscribe to collection nodes, nodes, and items within a room

See [07 Server to Server APIs](#). Both the Python and Ruby scripts are set up so that you can run them as is or use them as module which your application can invoke on-the-fly. Scripts are provided for the following languages:

- Cold Fusion
- Java
- PHP
- Python
- Ruby
- Groovy



Documentation currently resides in the scripts themselves.

Sample applications

The SDK delivers a few sample applications that provide working code you can install and run. Some of these demos are rudimentary and focus on a particular technique rather than a rich user interface. Others are more complete and may prove useful for your own applications.



The SDK Navigator contains a running demo with viewable source code.

Refer to the SDK's sampleApplications directory for more demos. The following is a partial list:

- **Audio:** Uses an audio publisher and subscriber and shows to change various audio settings such as gain, silenceLevel, echoSuppression, and so on via AudioPublisher properties. All users other than the person running the demo can hear the published audio.
- **ComplexObjectTransfer:** Demonstrates the use of the registerBodyClass method in MessageItem for passing items containing complex objects over the service.
- **CustomRoster:** Demonstrates how to use UserManager and UserRoles to promote, demote, and remove people by calling various UserManager API's and listening to its events to make user interface changes.
- **DefaultPods:** A simple first application demonstrating the use of the default pods as well as basic authentication and session creation.
- ***ExternalAuthentication:** Demonstrates the use of an application's integration with the SDK's server scripts.
- **FilePublisher:** Shows how to upload, download, delete, and display files via the FileManager, FilePublisher, and FileSubscriber components.
- **KnockingQueue:** Demonstrates the use of a knocking queue. When a user or users arrive at a room whose state is knocking (guestsHaveToKnock = true in RoomManager), and the host is there, the users are placed in a pending queue. The host sees the pending, accepted, denied queue and a chat window. Once a user is accepted, they see a chat; if they are not accepted, they see a blank screen.
- **LocalConnection:** A simple white board application that shows how to connect to the local server tool included in the extras directory. The Local Connection Server, available in the SDK Navigator, allows offline development and testing of non-streaming components.
- **LoginDialog:** A user interface showing login mechanism that allows you to log in as a member or guest.
- **MultipleGroups:** Demonstrates how to use multiple pod groups such as camera, chat, and note within a room. Grouping is useful for building applications that support private chats, private groups, etc. and where users can be divided into groups each of which see a different user interface.
- **SharedCollection:** Shows how to use a SharedCollection model for having synchronized lists and datagrids. In this case, we are using a synchronized datagrid to add, remove, edit items, and any newly arrived user can see the updated list.
- **SharedModel:** A better user interface showing several pod types as well as a standard log in mechanism. SimpleChat: Shows how this shared model can be made easily bindable for MXML.
- **WebCamera:** Demonstrates how the camera component can be used with a publisher and a number of subscribers. The publisher has a big view while subscribers have a small view. A shared property is used to pass the stream to the publisher's user interface. Every user is provided with play and pause handlers.
- **YahooMaps:** A working application using ActionScript 3 components that leverages Yahoo Maps.

05 Room Console

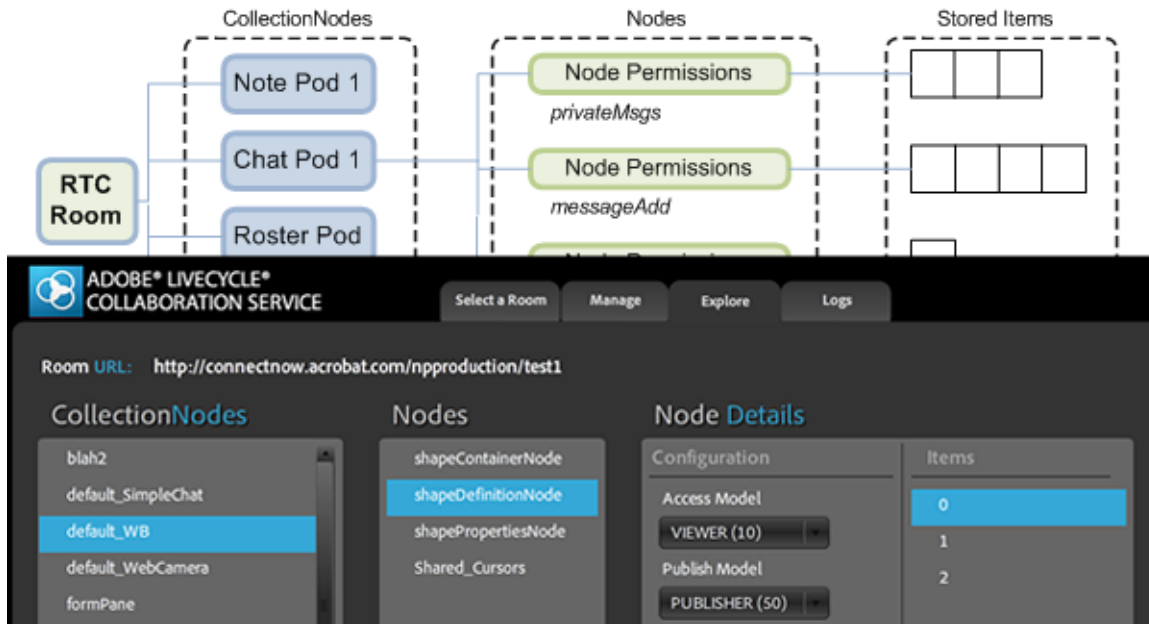
1. [Room management and monitoring](#)
2. [Managing accounts and rooms](#)
3. [Managing room settings](#)
4. [Managing room collections and nodes](#)
5. [Viewing logs and service usage](#)

5.1 Room management and monitoring

The Room Console provides a way for developers to monitor and manage rooms and users. It simplifies tasks that would otherwise have to be performed programmatically. As such, it enables immediate access to some of LCCS's sophisticated features such as templates, setting user roles, viewing streams, logging. The console is packaged with the LCCS SDK Navigator.

The console and the API

The Room Console provides a useful way to learn the LCCS API. Many of the console's user interface items share a name with the classes and properties in the API. The tabs provide a logical layout for panels and lists that are useful for learning the APIs architecture. For example, the Explore tab's panels from left to right are Collections, Nodes, Node Details, and Items--a structure which closely matches the APIs class to property architecture.

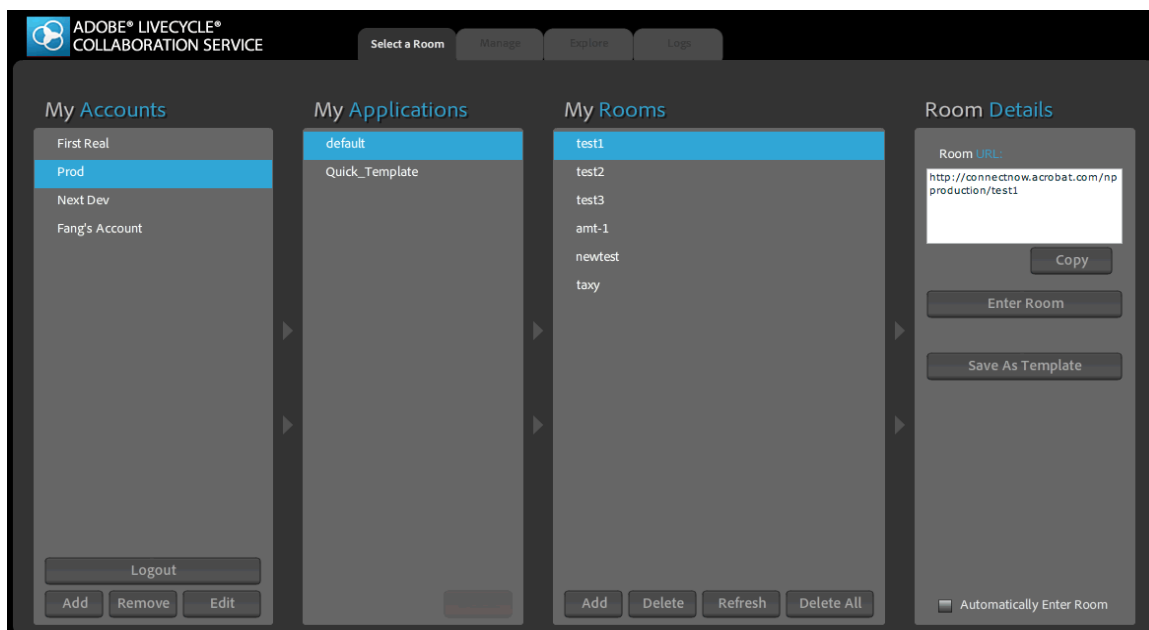


Room Console: Explore tab

5.2 Managing accounts and rooms

Most developers will have only one account; however, any number of accounts are possible. High level account management is available from the console's Select a Room tab. Many of the tabs features call the AccountManager class. These feature include:

- Logging in and out
- Adding an account
- Removing an account
- Editing account details
- Working with application templates
 - Creating an application template from a room
 - Viewing rooms based on a template
 - Deleting an application
- Viewing available rooms
- Entering a room
- Creating a new room
- Deleting a room



Room Console: Select a Room tab

Logging in and out

Logging in requires entering your account authentication information at least once. Logging out is simply a matter of highlighting the current account and choosing **Logout**.

To log in or out:

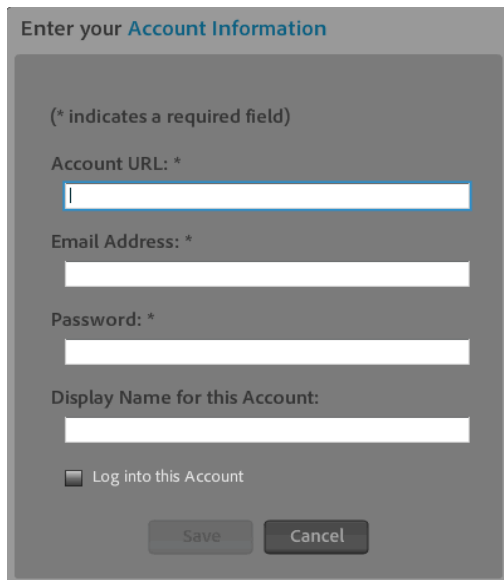
1. Choose the **Select a Room** tab.
2. Highlight an account.
3. Log in or out:
 - Double click the account name or choose **Login**, or
 - Choose **Logout**.

Adding an account

Obtaining an account occurs through the web rather than the Room Console. However, once you have an account, you can add it to the Room Console so that you can manage its rooms.

To add an account:

1. Choose the **Select a Room** tab.
2. In the Accounts panel, choose **Add**.
3. When the account details panel appears, enter the following:
 - **Account URL:** The root URL associated with this account. The URL was provided when you signed up for a developer account.
 - **User Name:** The developer account's Adobe ID.
 - **Password:** The developer account's password.
 - **Display Name for this Account:** The name to display in the Room Console.
4. Choose **Log in to this Account** if you would like to automatically log when you save the new account information.
5. Choose **Save**.

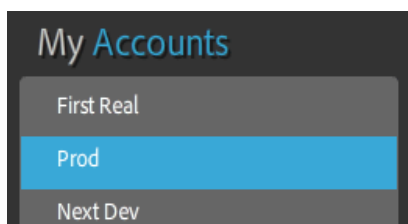
A screenshot of a dialog box titled "Enter your Account Information". It contains several input fields: "Account URL: *" with a text box, "Email Address: *" with a text box, "Password: *" with a text box, and "Display Name for this Account:" with a text box. Below these is a checkbox labeled "Log into this Account". At the bottom are "Save" and "Cancel" buttons. A note at the top left says "(* indicates a required field)".

Room Console: Add Account

Removing an account

To remove an account:

1. Choose the **Select a Room** tab.
2. Highlight an account.
3. Choose **Remove**.



Room Console: Account panel (cropped)

Editing account details

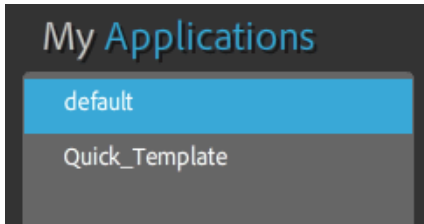
If you have trouble logging in or want to change the account(s) for which you automatically log in, verify your account details.

To edit account details:

1. Choose the **Select a Room** tab.
2. Highlight an account.
3. Choose **Edit**.
4. Review the details and change those which may be incorrect.
5. Choose **Save**.

Working with application templates

While you could create templates programmatically with the RoomTemplater class, the Room Console provides an intuitive user interface to get you going quickly. When you save a room as a template, you build a library of components with which you can quickly build or extend applications. Only an owner of the account can add an application template to the service.



Room Console: Applications panel

When you create a room you'll add one or more pods such as a chat, note, and whiteboard, you'll configure the room settings as well as the node details. That room's collection nodes and other configurations are stored on the service. Saving all of this room information as an application template allows you to create new rooms on-the-fly without the constraint of needing a room owner to be in the room to recreate them.

For example, a company might need hundreds of room instances that need to be provisioned on the fly, with each room having the same components and configurations. In the case where many clients (distributed applications) need to create rooms with a chat window, file sharing pod, and specific room settings, then they could do so by leveraging the owner's (developer's) pre saved work (the application templates) without getting the developer involved. Since the application template is saved on the service, it can be called in real-time to create new rooms automatically.

Creating an application template from a room

The default application is simply a blank room. However, any room you've created can be saved as a template.

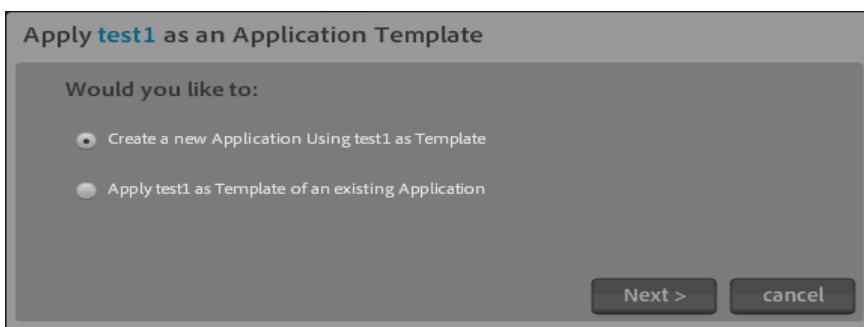
To create a template:

1. Choose the **Select a Room** tab.
2. Highlight an account to display its templates and instances.
3. Select the room instance you want to save as a template.
4. Choose **Save as Template**.
5. To create a new Application, select the first radio option and click "Next"
6. When the Save as Template dialog appears, enter a template name.



A template name has some restrictions. The room name can only contain letters and numbers, must begin with a letter, contain no spaces, and must be four or more characters long but no longer than 21 characters.

7. If you want to save your template over an existing Application, choose the second radio option, and select an Application to template. Note that this template will only affect **new** rooms built from that Application; it will not affect existing rooms.
8. Choose **Save**.



Room Console: Application Template dialog

Viewing rooms based on a template

If you have any custom templates, then the Rooms panel displays the rooms that are based on the selected application. To view rooms based on a particular application, follow the steps described in [Viewing available rooms](#).

Deleting an application

Deleting an application removes it from the service. Because rooms based on that application template are not deleted, those rooms are moved under the default application. However, the actual room components do not change.

To delete an application:


1. Choose the **Select a Room** tab.
2. Highlight an application.
3. Choose **Delete**.
4. When the confirmation dialog appears, choose **Yes**.

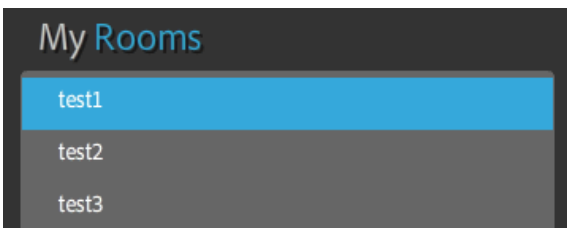
Viewing available rooms

When you log in to an account, all of the account's room are listed in the Room Instances panel.

To view available rooms:

1. Choose the **Select a Room** tab.
2. Log in to an account. If you did not automatically log in, highlight an account and choose **Login**.
3. Highlight an application. The application's rooms are listed in the Rooms panel.

 The Rooms panel lists only the rooms that are based on the selected application.




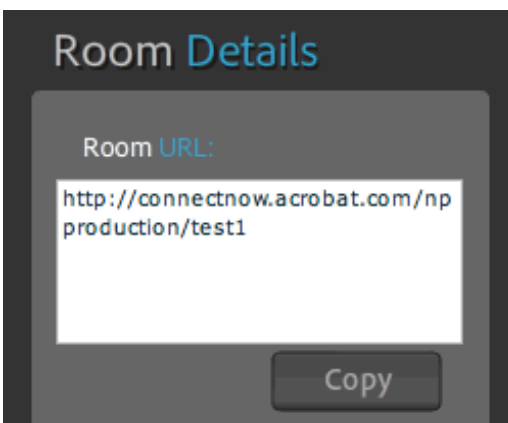
Room Console: Room instances

Entering a room

To enter a room:

1. Choose the **Select a Room** tab.
2. Double click on the room name or choose **Enter Room**.

 You can configure the console to automatically log you into the last room you entered. To do so, check **Automatically Enter Room**.



Room Console: Room Details panel (cropped)

Creating a new room

You can add a room simply by entering a new room name in the Rooms panel. The room name is appended to your account's root URL.

To do so:

1. Choose the **Select a Room** tab.
2. Log in to an account.
3. Highlight an application.
4. Choose **Add**.
5. Enter a room name.



Because the room name becomes part of the room URL, it has some restrictions. The room name can only contain letters and numbers, must begin with a letter, contain no spaces, and must be four or more characters long but no longer than 21 characters.

6. When the confirmation dialog appears, choose **Yes**.

Deleting a room

Deleting a room permanently removes from the service. The room is not recoverable. If there are users in the room, then they can continue to use the room until they leave it, at which point it becomes permanently unavailable.

To do so:

1. Choose the **Select a Room** tab.
2. Log in to an account.
3. Highlight a room.
4. Choose **Delete**.
5. When the confirmation dialog appears, choose **Yes**.

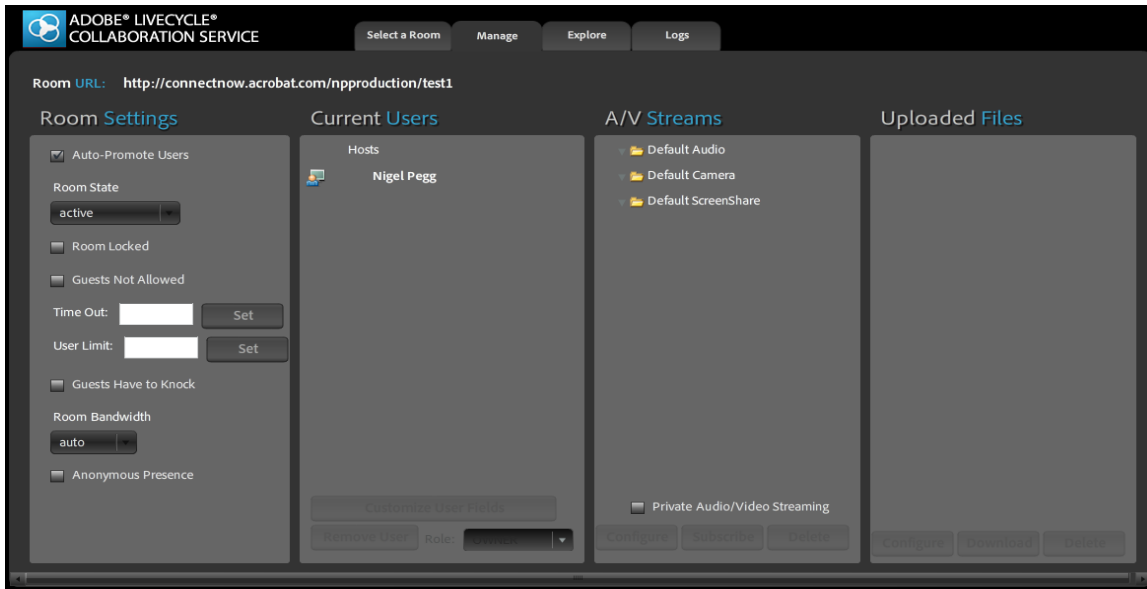
5.3 Managing room settings

The Manage tab provides the means to configure basic settings, set user roles, and manage streams and files. These properties can be set programmatically with the RoomManager class or simply by using the Room Console.



The Manage user interface manipulates the API's of the four fundamental Shared Manager classes.

- Auto-promoting users
- Setting up knocking for guests
- Setting the room state
- Setting the room bandwidth
- Locking a room
- Prohibiting guests
- Setting a timeout
- Limits on users
- Managing room users
- Changing a user's role
- Removing a user from a room
- Managing room streams
 - Subscribing to streams
 - Deleting streams
- Managing files
 - Downloading files
 - Deleting files



Room Console: Manage tab

Auto-promoting users

Turning on auto promote sets the boolean `autoPromote` property to true so that all users with a viewer role are promoted upon entry to a publisher role.

To turn auto promote on:

1. Choose the **Manage** tab.
2. Check **Auto-Promote Users** in the Room Settings panel.

Setting up knocking for guests

Turning on knocking sets the boolean `guestsHaveToKnock` property to true so that guests have to knock and then be granted permission before entering a room.

To require knocking:

1. Choose the **Manage** tab.
2. Check **Guests have to Knock**.

Setting the room state

The Room State drop down list sets the `roomState` property, thereby specifying the room state from among values supplied by RoomSettings constants.

To set the initial room state:

1. Choose the **Manage** tab.
2. In the Room State drop down list, select a state:
 - **active**: RoomManager state constant for an open, active room.
 - **ended**: RoomManager state constant for a room which has been closed.
 - **hostNotArrived**: RoomManager state constant for a room with no host.
 - **onHold**: RoomManager state constant for a room which has been placed on hold.

Setting the room bandwidth

The Room Bandwidth drop down list sets the `selectedBandwidth` property, thereby specifying the bandwidth from among values supplied by RoomSettings constants.



Audio and video components are sensitive to how a room is tuned; however, that sensitivity can be overridden with the API. A slower setting degrades the stream quality to account for the narrower bandwidth. The recommended option is *auto*.

To set the bandwidth:

1. Choose the **Manage** tab.
2. In the Room Bandwidth drop down list, set the bandwidth:
 - **auto**: (Recommended) Room connection speed constant for automatically calculating speed.

- **modem**: Room connection speed constant for MODEM.
- **LAN**: Room connection speed constant for LAN.
- **dsl**: Room connection speed constant for DSL.

Locking a room

The Room Locked checkbox sets the `roomLocked` property, and locks a room so that new users cannot enter.

To lock a room:

1. Choose the **Manage** tab.
2. Check **Room Locked**.

Prohibiting guests

The Guests Not Allowed checkbox sets the `guestsNotAllowed` property and prevents unauthenticated guests from entering the room.

To prohibit guests:

1. Choose the **Manage** tab.
2. Check **Guests Not Allowed**.

Setting a timeout

The time out field sets the `roomTimeOut` property sets a timeout (in seconds) for the room after which the room session ends. The default is null, which sets the property to `NO_TIME_OUT`. Only users with role `UserRoles.OWNER` can set this property.

To prohibit guests:

1. Choose the **Manage** tab.
2. Enter a value in seconds for the timeout.
3. Choose **Set**.

Limits on users

The User Limit field sets the `roomUserLimit` property and thereby specifies the number of users permitted to enter the room. Only users with the role `UserRoles.OWNER` can set this property. The default is null, which sets the property to `NO_USER_LIMIT`.

To limit the number of total users:

1. Choose the **Manage** tab.
2. Enter the number of allowed users.
3. Choose **Set**.

Managing room users

The Current Users panel lists a room's current users by role. The panel allows you to view a room's current users, change a their role or remove them from the room.



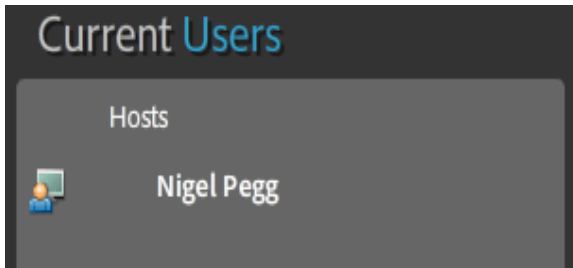
Developer's are logged into a room as a user, and the console itself is a room user.

Changing a user's role

User roles determine the level of a user's permissions. From the Room Console, choices are limited to those provided in the drop down list. If you'd like to specify other or custom roles, do so programmatically via the API. For example, you can specify a custom role with the `UserManager` class. Roles are all stored as integers.

To specify a user role:

1. Choose the **Manage** tab.
2. In the Current Users panel, highlight a user.
3. To change their role, choose the role button:
 - **OWNER** (100): OWNER can create, configure, and delete nodes, as well as publish and subscribe.
 - **PUBLISHER** (50): PUBLISHER can publish and subscribe to most nodes but cannot create, delete or configure nodes.
 - **VIEWER** (10): VIEWER can subscribe to most nodes but cannot publish or configure.



Room Console: Current Users panel

Removing a user from a room

Removed users are automatically disconnected from the service but are not deleted.

To remove a user:

1. Choose the **Manage** tab.
2. In the Current Users panel, highlight a user.
3. Choose **Delete**.
4. When the confirmation dialog appears, choose **Yes**.

Managing room streams

The Streams panel allows you to view streams by user and group. Once you've highlighted a user in the Current Users panel, that user's streams are listed by group.

A group is a set of streams with configurable permissions and roles. For instance, there could be a hosts-only group within which only hosts could talk to each other over VoIP. Anyone with lesser permissions wouldn't be permitted to participate.

The following features are available:

1. Expand and collapse the group tree by choosing the arrow on the left of the group name.
2. Subscribe to a stream (see [Subscribing to streams](#)).
3. Delete a stream (see [Deleting streams](#)).

Subscribing to streams

Active streams are listed in the Streams panel. You can subscribe to these streams to view/hear them in real time.

To subscribe to a stream:

1. Choose the **Manage** tab.
2. Highlight a stream.
3. Double click the stream name or choose **Subscribe**.



A free standing pod should appear. For example, a Camera stream will pop up a camera pod so that you can view the stream in real time.

Deleting streams

Deleting a stream stops the publisher from continuing to publish that stream.

To delete a stream:

1. Choose the **Manage** tab.
2. Highlight a stream.
3. Choose **Delete**.
4. When the confirmation dialog appears, choose **Yes**.

Managing files

The Files panel displays a room's uploaded files. Files are listed by group. By default, files appear under the default group name.

Downloading files

Any file uploaded from a client application can also be downloaded via the Room Console.

To download a file:

1. Choose the **Manage** tab.
2. Expand the file group.
3. Highlight a file.
4. Choose **Download**.
5. Browse to a location where you would like to save the file.
6. Choose **Save**.

Deleting files

Any file uploaded from a client application can also be deleted via the Room Console. Files are deleted in real time and are immediately removed from the current room.

To delete a file:

1. Choose the **Manage** tab.
2. Expand the file group, if any.
3. Highlight a file.
4. Choose **Delete**.
5. When the confirmation dialog appears, choose **Yes**.

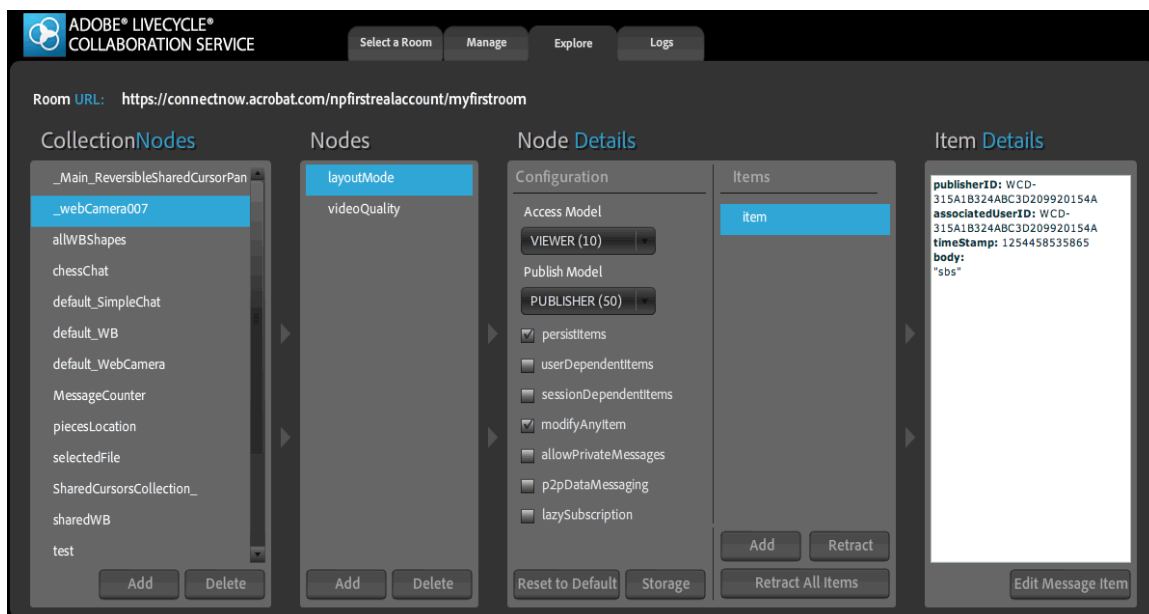
5.4 Managing room collections and nodes

When you are logged in to a room, the Explore tab is automatically populated with the room's CollectionNodes, Nodes, and MessageItems. From here you can add and delete items, change node configuration items (including the access and publish model), and edit items.



A CollectionNode is destination on the service through which messages are sent and received. For example, a pod component in your application can create destination on the server (the CollectionNode) and keep messages synchronized. You can add CollectionNodes and Nodes from the console or programmatically in advance.

- Adding a collection node
- Deleting a collection node
- Adding and removing nodes
- Configuring nodes
- Adding items
- Retracting items
- Editing items



Room Console: Explore tab

Adding a collection node

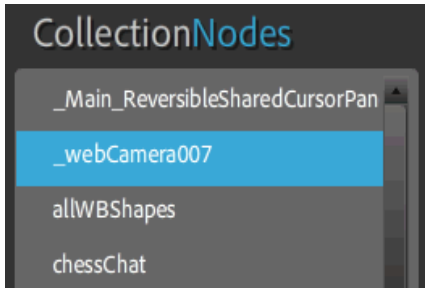
Items in the CollectionNode panel map to the API's pod components, although you can create your own components. For example, depending on what your room contains, the following may appear:

- default_FileShare
- default_HorizontalRoster
- default_Note

- default_Roster
- default_SharedWhiteBoard
- default_SimpleChat

To add a CollectionNode:

1. Choose the **Explore** tab.
2. Choose **Add**.
3. Enter a CollectionNode name in the Add dialog.
4. Choose **OK**.
5. When the CollectionNode appears, view and/or configure the other panels to the right.



Room Console: CollectionNode panel

Deleting a collection node

You can delete any CollectionNode in the CollectionNode tab. Doing so removes it from the service.



You should only delete CollectionNode that you are not using anymore. Verify they are not being used prior to deleting them. Deleting a node automatically deletes all its underlying nodes and message items.

To delete a CollectionNode:

1. Choose the **Explore** tab.
2. Highlight a CollectionNode.
3. Choose **Delete**.
4. When the confirmation dialog appears, choose **Yes**.

Adding and removing nodes

The Nodes panel displays all of the nodes of the pod component (a CollectionNode) that is highlighted in the CollectionNode panel. Since each node can be individually configured, it's advantageous to add nodes as needed when you would like to modify the default behavior.

To add a node:

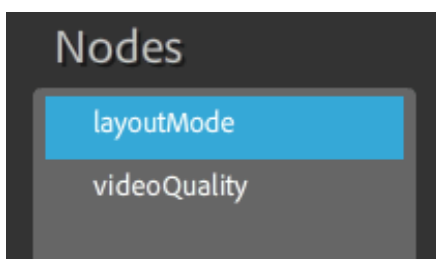
1. Highlight a pod component in the CollectionNode panel.
2. Choose **Add** under the Node panel.
3. Enter a node name.
4. Choose **OK**.

To delete a node:

1. Highlight a pod component in the CollectionNode panel.
2. Highlight a node.
3. Choose **Delete** under the Node panel.
4. Enter a node name.
5. When the confirmation dialog appears, choose **Yes**.



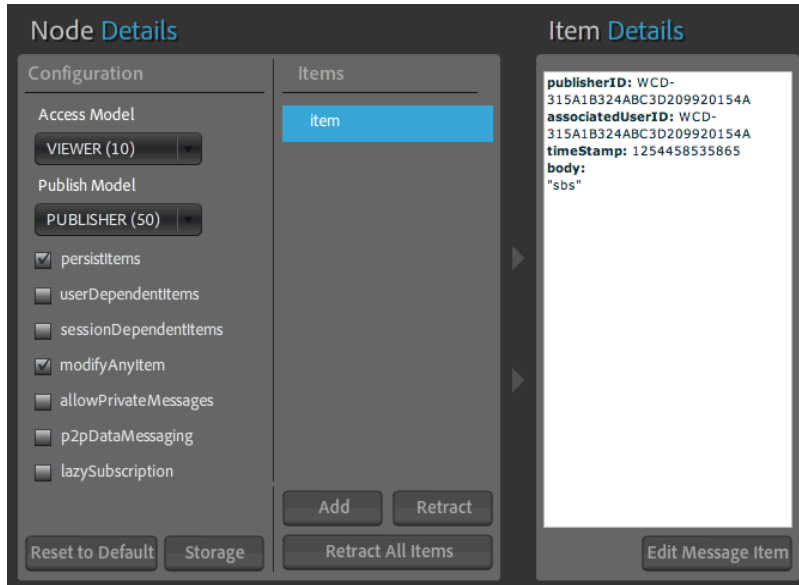
You should only delete nodes that you are not using anymore. Verify they are not being used prior to deleting them.



Configuring nodes

The Node Details panel allows you to set the NodeConfiguration class properties for any node in the current CollectionNode. Here you can set the user roles for publishing and viewing the node, and you can also set many of the nodes properties.


As nodes are selected (highlighted), any nodes with messageItems have that item appear in the Items panel. You can add new items and retract existing items for each node. Item details appear in the Item panel to the right.



Room Console: Node Configuration panel

To configure a node:

1. Select a CollectionNode and a node so that the Node Details panel becomes active.
2. Configure what role a user must have to subscribe to a node by setting the `accessModel` property. Values derive from constants in the UserRoles class; however, you can set a custom value:
3. **OWNER:** 100 OWNER can create, configure, and delete nodes, as well as publish and subscribe.
4. **PUBLISHER:** 50 PUBLISHER can publish and subscribe to most nodes but cannot create, delete or configure nodes.
5. **VIEWER:** 10 VIEWER can subscribe to most nodes but cannot publish or configure.

 You can also add custom roles to a node. The values must be between 1-100 and can not be 10, 50, and 100. This is true for both the access and publish Model.

6. Configure what role a user must have to publish to a node by setting the `publishModel` property. Values derive from constants in the UserRoles class (see above); however, you can set a custom value by programmatically setting it on a node since the access and publish model is exposed by all components.
7. Set the rest of the properties as needed.

Node configuration properties

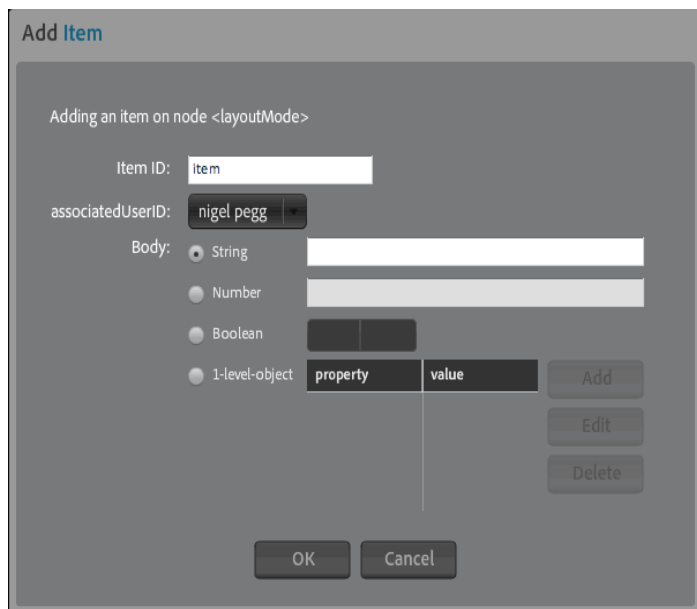
Property*	Description
<code>accessModel</code>	int The minimum role value required to subscribe to the node and receive
<code>publishModel</code>	int The minimum role value required to publish MessageItems to the node.
<code>allowPrivateMessages</code>	Boolean Whether or not private messages are allowed.
<code>itemStorageScheme</code>	int Storage scheme for the MessageItems sent over this node.
<code>modifyAnyItem</code>	Boolean Whether or not publishers may modify other users' stored items on the node (true) or only MessageItems they have published (false).
<code>persistItems</code>	Boolean Whether or not MessageItems should be stored and forwarded to users arriving later (true) or not stored at all (false).
<code>sessionDependentItems</code>	Boolean Whether or not stored MessageItems should be retracted from the server when meeting session ends (true) or left until manually retracted (false).
<code>userDependentItems</code>	Boolean Whether or not stored MessageItems should be retracted from the server when their sender leaves the room (true) or left until manually retracted (false).

Adding items

Items are messageItems - adding and editing a messageItem is equivalent to publishing a messageItem. You can use the MessageItem class to manipulate these items programmatically.

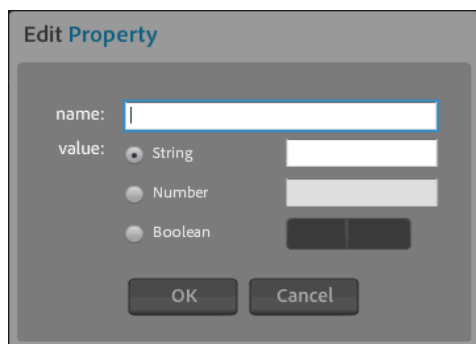
To add a new node item:

1. Select a CollectionNode and a node so that the Node Details panel becomes active.
2. Under the Items panel, choose **Add**. The Add Item dialog appears.



Room Console: Edit Item panel

3. Configure the item details:
 - **Item ID:** An ID which identifies the message item.
 - **associatedUserID:** The user ID of the person associated with this message item.
 - **Body:** The key value pair of name and value.
 - **String:** A simple string.
 - **Number:**
 - **Boolean:**
 - **1-level-object:** A property name and value. See below.
4. Choose **OK**. If configuring a 1-level-object, do the following:
 - a. Select 1-level-object.
 - b. Choose **Add**.
 - c. Configure the property.
 - d. Choose **OK**.
 - e. Choose **OK** again to exit the edit dialog.



Room Console: Edit Property panel

Retracting items

Retracting a item is rarely necessary, but it is possible. For example, if someone is abusing the service with negative language in a chat window, you can delete the offensive text from the service.

To retract an item:

1. Select a CollectionNode and a node so that the Node Details panel becomes active.
2. Under the Items panel, choose **Retract**.
3. When the confirmation dialog appears, choose **Yes**.

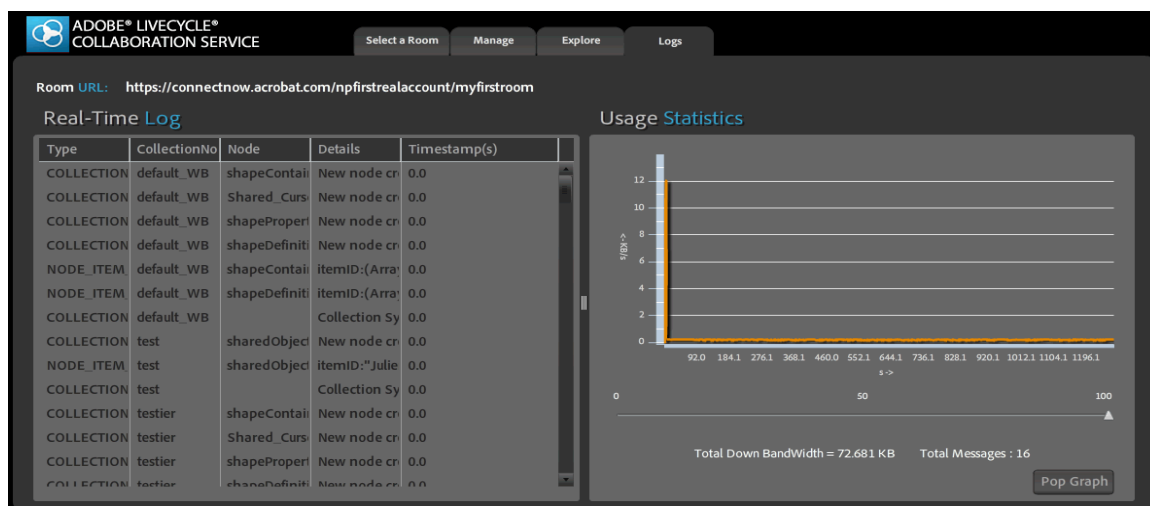
Editing items

Items are edited in the same way they are added. To edit an item, highlight it, choose Edit, and configure it as described in [Adding node items](#).

5.5 Viewing logs and service usage

The Logs tab provides real-time details about a room's usage.

- [Viewing log data](#)
- [Viewing service usage](#)
- [Viewing usage statistics in a modal window](#)



Room Console: Logs tab

Viewing log data

When viewing log data, you can sort the columns by clicking on column header.

Recorded data includes the following:

1. **Type:** The event type generating the log entry.
2. **CollectionNode:** The collection node name.
3. **Node:** The affected property of the relevant node within the collection node, if any.
4. **Details:** The node property details, if any.
5. **Timestamp:** The time of the event, in seconds starting from the time the Room Console entered the room.

Viewing service usage

The Room Console is not meant to be an overall collector of statistics or an authoritative recorder of overall usage. Instead, it only measures bandwidth that's coming to the console itself and only when console is connected to a particular room. Thus, the recorded bandwidth is really limited to the bandwidth for one person in the room (that is, one client's consumed resources).

Usage statistics are logged in kilobytes per second. You can adjust the displayed data in two ways:

1. Change the bandwidth scale to view kilobyte usage over a custom time frame by moving the slider to the left or right. This allows you to zoom in and out.
2. Change time you are viewing by moving the horizontal scroll bar to the left or right. This allows you to move forward and backward to select a specific viewing time.

As an overview, the following totals are provided:

- **Total down bandwidth:** The total bandwidth used by the Room Console itself and any application the person is running in the same room. This figure is meant to provide an estimate of the bandwidth for one person in a particular room (that is, one client's consumed resources).
- **Total messages:** The total message sent to the server. For example, when a user types a chat message, one message is sent to identify the typist, and another is sent for the actual message.

Viewing usage statistics in a modal window

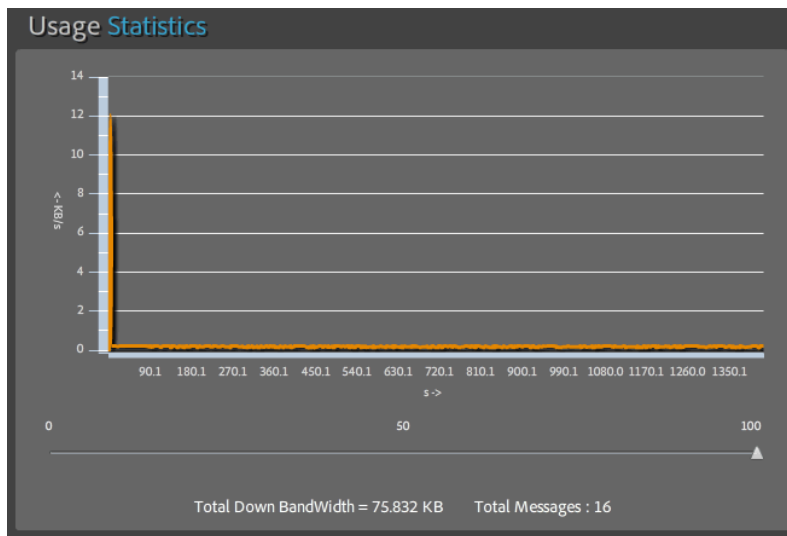
The real time Usage Statistics graph can be "popped" out of the console so that it can be viewed independently. Not only can it be viewed anywhere on the desktop, but it will also continue to run even after the Room Console is closed.

To view the graph window as a modal window:

1. To the lower right of the graph, choose Pop Graph.



You can use scroll bar and zoom to see the activity at a specified time.



Room Console: Usage Statistics

06 Deploying Applications

By now you probably know that a room is simply a virtual location on the service to which clients connect. The room's address is an URL. People go there to meet and exchange messages; data goes in and out. Of course, you'll want to manage the room, its members, and the flow of data.

As usual, development details start out fairly simple but grow increasingly complex. You'll likely start out by using the default template and a simple room. During testing and development, you'll work offline with the Local Connection Server rather than online. When you do connect to the hosted service, you'll authenticate with your Adobe ID.

However, since you're clever and excited about the possibilities, you'll soon start architecting more complex rooms, devising user role schemes, creating room templates, and crafting rich user interfaces. Once you deploy an application, many of these development-specific details change. For example, only developers need Adobe ID, and your destination room URL will likely be different than your development URL. You'll also need to set up rooms and publish-access models for your components as well as decide how users roles will be handled and what authentication mechanism to use.

Irrespective of the use case, workflows usually involve the following steps:

- **Deployment Scenarios:** Does the nature of your project and application justify the use of templates and server side integration, or should you simply manually create and configure your room(s)? Your decisions here affect the rest of the deployment steps.
- **Provisioning rooms:** Creating a new virtual room on the LCCS service. All rooms are based on a template, but you can add features manually or via custom templates; e.g. essentially furnishing the room with elements that control room behavior (time out, guest access, etc.) as well as the flow of information in and out of the room (text, streams, files, etc.)
- **Templating rooms:** Once you have one or more "well furnished," provisioned room, you can save it as an application template for future use. Application templates provide library-like functionality as the basis for similar rooms and are also used for programmatically spawned rooms.
- **Authentication setup:** Deciding whether or not to require authentication for room entry. You can allow anyone as a guest, or you can leverage your existing systems and authentication workflows to direct users to the right rooms and assign permissions based on their credentials.

6.1 Deployment Scenarios

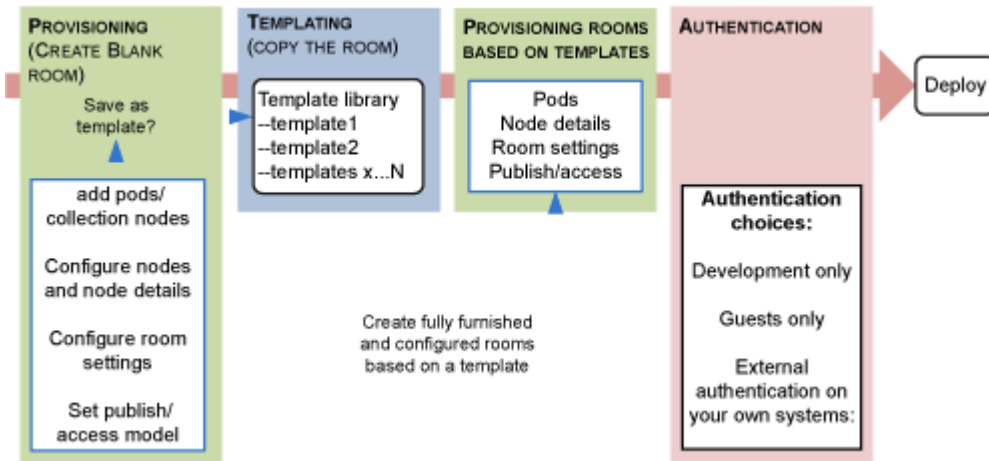
There are two types of deployment workflows:

- **Scenario 1: Non-dynamic room management**
- **Scenario 2: Dynamic room management**

In most cases, you will do what developers always do: Create something, template it, create more "somethings" from the template, and then deploy your creation with the requisite elements that satisfy your particular needs. LCCS is all about ease of application development and speed to deployment, so in addition to support for the FLEX and Flash frameworks and a multitude of [Developer Tools](#), the recommended workflow includes setting up the ability to mass produce unlimited numbers of feature rich rooms:

1. Provisioning rooms.
2. Creating application templates from those rooms.

3. Creating more rooms programmatically from those templates.
4. Deploying client applications with fully functional authentication that leverages your existing mechanisms.



Application development workflow

Scenario 1: Non-dynamic room management

Occasionally, but not often, you may want a one-of-a-kind room or rooms that anyone can use. While most deployment scenarios include programmatic provisioning and authentication, you can simply create and deploy a room via the [Room Console](#). Do so when your application:

- Requires one room or a fixed number of rooms.
- Doesn't need authentication or dynamic assignment of user roles and permissions.
- Is the first one you're creating. This is an excellent way to get started. See also:
 - [Building your first application](#)
 - [Sample applications](#)

Scenario 2: Dynamic room management

Dynamic room management unleashes the power of programmatic room creation and server side integration. For many application types have multiple rooms with different groups of people, publish-access models, and other features which are too unweildly manual management. For example, you might create a virtual classroom or game with many rooms for each. To avoid manually creating all the right rooms, you provision rooms automatically via the programmatic server-side scripts as described in [Programmatic provisioning](#).

6.2 Provisioning rooms

Provisioning is the act of creating a new room and optionally furnishing it with the needed components and configurations when the room is based on one of your custom templates.

```
<LCCSLCCS root URL>/<your developer account name>/<room name>
```

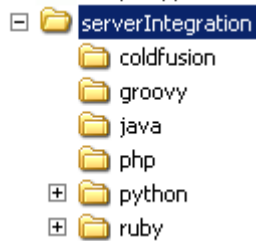
Room URL format

You can create rooms as follows:

- **Manually:** For one room or a fixed number of rooms where dynamic control is not required.
 - **Developer Portal:** The Developer Portal offers the first chance to create a room. Simply choose Add in the Room Instances panel. To furnish your room and configure its settings, build your application in Flash Builder (logged in as developer), and use the Room Console.
 - **Room Console:** The Room Console allows you to both create and configure rooms. For details, see [Room Console](#).
- **Programmatic provisioning:** After deployment, you'll likely create rooms programmatically using server-side scripts. The server can create template-based rooms which are selected on the basis of the user's credentials on your own systems. For examples, see the SDK's server scripts in the "extras" directory.

Programmatic provisioning

Most deployed applications provision rooms programmatically since it is simpler to have the server manage room creation, user access, and the data model. For applications containing hundreds of rooms and thousands of users, manual management isn't even an option.



The SDK provides a number of scripts in common languages that you can leverage directly on your server. These scripts call the LCCS APIs on the service to perform common tasks on-the-fly.

The following code snippets use Ruby, but these descriptions below are basically language agnostic ; that is, similar methods are available regardless of your server environment of choice. Whether you use the provided scripts in Cold Fusion, Groovy, Java, PHP, Python, or Ruby, you'll find the techniques similar.

At a high level, programmatic provisioning involves the following:

1. Load any libraries or resources you might need. For example, Ruby, requires base64, digest/md5, date, net/https, rexml/document, and openssl.
2. Create a new account manager object using your account URL. This URL is displayed in the Developer Portal when you log in.

```
am = RTC::AccountManager.new("http://collaboration.adobelivecycle.com/<YOUR DEVELOPER  
ACCOUNT NAME>")
```

3. Use the login method to log in with your developer account's username and password. Note that this is the only time (other than during development) that you would use your personal credential. Since your credential is used in a server to server context, it is private.

```
am.login(accountowner, accountpassword)
```

4. Create a room:

```
am.createRoom("newRoom", "fromTemplate")
```

i While you can create a room without a template, it will be empty and contain no nodes. By leveraging your own pre-constructed application templates, your rooms will arrive prepopulated with the facilities to operate the components you need, whether a whiteboard, chat, or one or more other pods. To create a room from a template simply pass the template name to `createRoom`.

5. Perform other tasks as needed; use the provided methods to list, create, and delete rooms and templates on-the-fly. There are other APIs than those listed here:
 - `rooms = am.listRooms()`
 - `templates = am.listTemplates()`
 - `am.deleteRoom("myRoom")`
 - `am.deleteTemplate("myTemplate")`
6. Use external authentication to authenticate users to your systems when they enter the room.

i For client-side details, see the SDK's External Authentication sample application.

- a. Create a session object from the current session.

```
session = am.getSession("room1")
```

- b. Instantiate a secret variable using your shared secret provided by the Developer Portal.

```
secret = "<YOUR ACCOUNT SHARED SECRET>"
```

- c. For each user and from your own system, get an user ID, a username, and an assigned role.
- d. Generate an `authToken` by passing the required parameters to `getAuthenticationToken`. The token is a unique, signed string created from the your shared secret, the user's ID and username, and their assigned role.

```
puts session.getAuthenticationToken(secret, "<USERID>", "<USERDISPLAYNAME>", 100)  
puts session.getAuthenticationToken(secret, "<USERID>", "<USERDISPLAYNAME>", 50)  
<repeat n times; once for each user>
```

Authentication Token retrieval

7. Provide the client application with the authToken.

```
<mx:Script>
    <![CDATA[
        [Bindable]
        private var roomURL:String;
        [Bindable]
        private var authToken:String;

        private function init():void {
            roomURL = Application.application.parameters["roomURL"];
            authToken = Application.application.parameters["authToken"];
            cSession.login();
        }
    ]]>
</mx:Script>
```

8. Invalidate the session. A session object is valid for a set amount of time. Tokens also get locked down so they can be used again. Both a session and tokens are invalidated when meeting ends.

```
am.invalidateSession(session)
```

i `invalidateSession` immediately makes all users token invalid instead of waiting for when a meeting ends. You may not need to invalidate a session since they automatically expire five minutes after all users have left the room. Invalidate the session when you want to assure that users cannot log in again after their application has decided the session has ended.

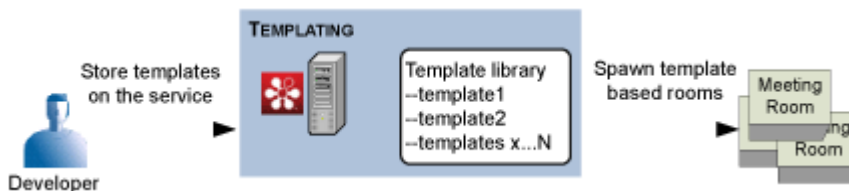
6.3 Templating rooms

All rooms are based on application templates. Your account provides a default, empty application. You can't delete it, but you will create your first room with it. However, after you've furnished and configured your room with pods, node details, room settings, and so on, you may want to save it as an application template. You can build a library of templated applications with the Room Console. The applications are stored on the service so you can create rooms on the fly by calling the appropriate API.

i A template is simply a shortcut to creating new rooms with all the components and settings you're likely to reuse.

For example, a company might need hundreds of room instances that need to be provisioned on the fly, with each room having the same components and configurations. In the case where many clients need to create rooms, then the developer can write the web application to use LCCS's server scripts to provision new rooms based on that template library.

- [Templating with the Room Console](#)
- [Templating with the Developer Portal](#)



Provisioning rooms based on a template

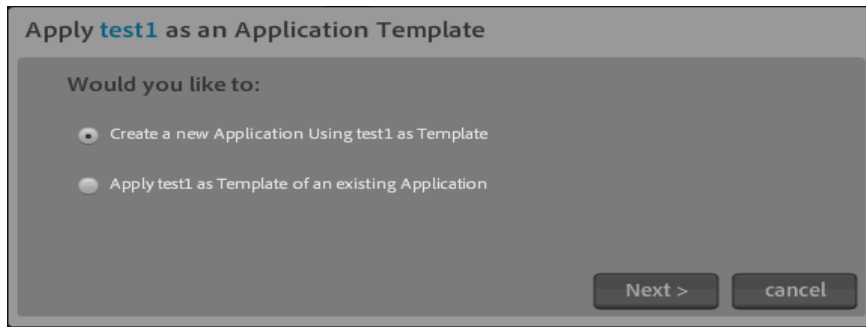
Templating with the Room Console

While you could create templates programmatically, the Room Console provides an intuitive user interface to get you going quickly. It provides the same functionality as the API, so your templates can be as richly configured as if you were using the API. For details about using the Room Console to create templates, see [Working with application templates](#).

Templating with the Developer Portal

The Developer Portal (<https://collaboration.adobe.livecycle.com>) provides facilities to create application templates. You simply select a room and click Template. From here, you can choose to use the selected room as the template for an existing application, or create a new application based on that template. Choosing an existing application simply overwrites that application's template, and any subsequent

rooms created in that application will use the new template. Existing rooms are not affected by a template change.



Room Console: New Template dialog

6.4 Authentication setup

LCCS's authentication mechanism is pretty simple, yet it is still capable of providing a full range of features through its support of your existing infrastructure. That is, clients authenticate through a single line of code, but what parameters you send that line will allow you to achieve just about anything you'd like to with respect to authentication. Let's take a look at that line:

```
<rtc:AdobeHSAuthenticator <----- some parameters -----> id="auth"/>
```

AdobeHSAuthenticator

AdobeHSAuthenticator is used to log users in to a LCCS hosted room with the requisite parameters for the given log in scenario. These include:

- **Development stage authentication:** Adobe ID username and password
- **No role-based permissions or authentication:** any username
- **Authenticating on your own systems:** authenticationKey

AdobeHSAuthenticator stores the login values with an ID of auth (an arbitrary ID). When you pass them to the ConnectSessionContainer authenticator property, the ConnectSession authenticates the user based on the supplied parameters.

```
<session:ConnectSessionContainer
  roomURL="http://connect.acrobat.com/exampleAccount/exampleRoom"
  authenticator="{auth}">
  <pods: <----- some components -----> "/>
</session:ConnectSessionContainer>
```

authenticator passed to ConnectSessionContainer

Development stage authentication

Developers may find it expedient to hard code their Adobe ID during development and testing so that their application will communicate directly with the service. In this case, pass your Adobe ID username and password to AdobeHSAuthenticator. It should be obvious that authentication details should never be hard coded anywhere, including deployed SWFs, remove your credentials before deployment.

When offline or for performance reasons you decide to develop locally, you can connect to the Local Connection Server on your own machine. Running an application with any arbitrary username. Just change AdobeHSAuthenticator to LocalAuthenticator.

 For an example, see the Local Connection Server demo in the SDK's sampleApplications directory.



No role-based permissions or authentication

Some application may only need anonymous or guest users. In this case, AdobeHSAAuthenticator accepts any arbitrary username, only the username is required, and used only as a display name.

Since the application requires no authentication, every room member has the same role. In order to facilitate collaboration, it is likely that all the guests would be auto-promoted to publishers. For more information on managing guest users, see `autoPromote` and `guestsHaveToKnock` in the RoomManager class.

For example, you could create a an application that doesn't require authentication on your servers. The application could be a SWF people can download or receive via an email. Anyone that knows the room URL would be able to log in as a guest and start participating in the chat and on the whiteboard. Such an application could be simple and useful on a small scale, but the lack of control could create both bandwidth and participant problems for the developer.



Workflow: for authenticating guest users

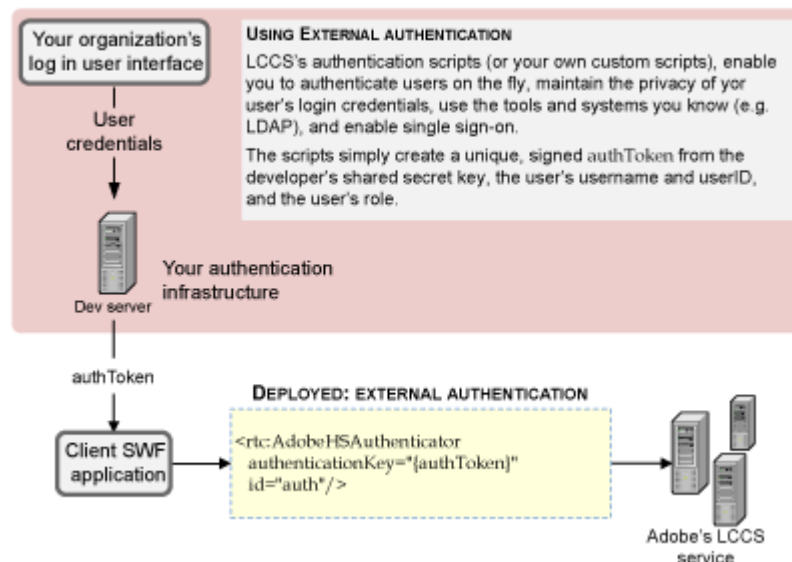
Authenticating on your own systems

Most developers will require users to authenticate on their own systems. If your and/or your organization already have authentication infrastructure in place, it makes sense to leverage those deployed assets. If your users can simply log in to your own systems as usual and thereby access LCCS applications, then their experience will be seamless and simple. LCCS's external authentication support provides exactly this kind of transparent client integration with your systems.

LCCS applications rely on external authentication to take advantage of the following:

- You can leverage existing databases and infrastructure; use what you know and have in place for zero-cost deployment.
- End users do not have to have an Adobe ID and remain anonymous to the service.
- Users retain the log in credentials they have always used.
- Clients can be pre authenticated; users enter rooms without having to log in (single-sign on).
- Your account will use less bandwidth and a reduced message count due to less contact with the service.

YOUR TOOLS, YOUR SYSTEMS, AND YOUR PRIVATE CREDENTIALS



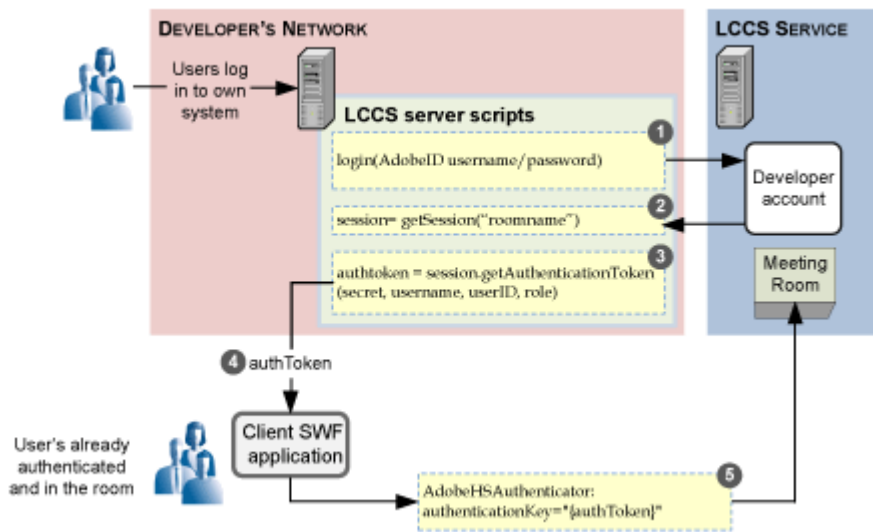
Workflow: for authenticating with external authentication

Workflow

Setting up external authentication involves the following:

1. Obtain an account shared secret from the developer portal.
2. Get the requisite server script from the SDK's extras directory.
3. Set up the scripts as described in the script documentation.
4. The server script performs the following tasks:
 - a. Logs the developer into the LCCS service with the developer's Adobe ID.

- b. Requests a session for a particular room name.
 - c. Generates an authToken. The token is a unique, signed string created from the developer's shared secret, the end user's user ID and username, and the end user's assigned role.
 - d. Provides the client application with the authToken.
 - e. The client connects to the LCCS room.
5. The end user is now provided access to the user interface. They are already in the appropriate room with the specified role and associated permissions.



Application scenario: Provisioning rooms based on a template and user credentials

07 Server to Server APIs

LiveCycle Collaboration Service lets developers add their own logic via code on their own servers. The Server to Server APIs allow third-party servers to add notification hooks and subscribe to events in various collections. When an important event occurs, the third-party server can send its own response to the LCCS server. The LCCS server then redistributes the third-party response to room clients. In effect, the Server to Server APIs implement remote procedure calling with a simple HTTP POST protocol.

- [About Server to Server functionality](#)
- [High level workflow](#)
- [Revisiting the chat example](#)
- [Server to Server API Reference](#)

7.1 About Server to Server functionality

If you create an unmoderated chat application with LCCS, your application logic resides in the web client. Each client receives messages from LCCS. The client then displays the messages, lets the user create replies, and sends the user's input back to LCCS for routing. Your application doesn't customize LCCS behavior; it simply takes advantage of standard LCCS routing functionality.

In more sophisticated applications, however, developers often need to centralize sensitive logic to protect it and to manage it efficiently. For example, suppose you want to apply a profanity filter to your chat rooms. For efficiency and security, you want to remove profanity from a message before it is routed to clients. Logic such as this must be imposed at the server level, but in this case the chat is hosted on an LCCS server. Because you don't own that server, you have no direct control over its behavior.

This dilemma illustrates the need to insert third-party logic into an LCCS room. LCCS addresses this need with Server to Server APIs. These APIs cover cases where a client isn't the appropriate place to put application logic. With these APIs, your external server can virtually add its own logic to a room.

- [Advantages of Server to Server APIs](#)
- [Real-time notifications with hooks and subscriptions](#)
- [LCCS and AMF](#)
- [Security considerations](#)
- [Sample applications](#)

Advantages of Server to Server APIs

Server to Server APIs give external servers much of the same LCCS integration that Flash-based clients enjoy. They harness HTTP to enable a variety of functions, such as:

1. Monitoring, auditing, and storing messages in a room
2. Intercepting messages and taking relevant action (for example, applying a profanity filter), and publishing the result back to the room
3. Getting notifications when a room starts up and shuts down

4. Adding a point from which to inject business logic as if you were the room owner (for example, splitting the room into sub-rooms, or controlling game logic)
5. Federating between rooms or between external systems and rooms (for example, it is possible to integrate XMPP chat into a room)
6. Querying room details (for example, who is in the room, how long has a room been running, and so on)
7. Publishing items, removing nodes, and performing other native room actions

Server to Server APIs extend the existing LCCS scripting APIs. They also support the same scripting languages as existing APIs. See [Install the SDK](#) for the list of supported languages.

Real-time notifications with hooks and subscriptions

Server to Server APIs use a hook notification model. Hook notifications give developers the chance to integrate their internal logic into LCCS rooms. By registering hooks and subscribing to collections, third-party servers receive notifications of events that occur in the LCCS service. Examples of events include the publication of a message item or the removal of a node.

The hook mechanism provides a simple method of registering a callback URL with a service. The Adobe server calls the hook URL via HTTP POST when an event occurs. LCCS uses Action Message Format (AMF) Remoting to deliver notifications.

See <http://www.webhooks.org> for conceptual information.

LCCS and AMF

LCCS uses Action Message Format (AMF) for sending notifications to external servers. AMF provides a simple framework for making remote procedure calls on a third-party server. This framework offers many advantages:

1. It uses a simple model for registering remote procedure calls that is familiar to most developers working with Flash/Flex applications.
2. It's supported by the scripting languages LCCS supports, through free libraries.
Encoded objects sent through AMF are deserialized by the libraries on third party servers.
3. It can support most of the richly typed object formats available in Flash-based clients.
4. Objects of registered custom ActionScript classes can be round-tripped.
5. It's a more efficient data transfer mechanism than JSON or XML. It allows LCCS to send objects directly from memory, rather than through the string-based serializers required by JSON or XML. This efficiency results in scalability advantages for LCCS.

In order to use hook notification through AMF, your server requires an AMF Remoting gateway. See <http://raghuonflex.wordpress.com/2008/04/17/data-push-in-flex-with-backend> for information on setting up the gateway. Also, your remoting gateway must implement a method for each hook type that LCCS defines. See 7.4.6 Hook APIs.

Security considerations

As with the LCCS server libraries, the Server to Server APIs require that the third party server log into LCCS using the developer's account credentials. After initial login, session tokens are sent with each API call to LCCS. LCCS treats each call as coming from a "super server" entity. The "super server" label means that the third-party server has full owner credentials for rooms in that developer's account.

For notification hooks, the primary security goal is to guarantee that the remote call is in fact coming from LCCS, thus avoiding "man in the middle" attacks. The registerHook() function provides an optional parameter that accepts a security token. Thus, security is optional, at the developer's discretion, but Adobe recommends registering your hook with a security token. These tokens are passed back on every hook notification, allowing developers to verify that the call came from LCCS. For developers with heightened security concerns, Adobe recommends re-subscribing regularly with a new security token (for example, once a day). This strategy shortens the window in which attackers can intercept and compromise tokens.

Sample applications

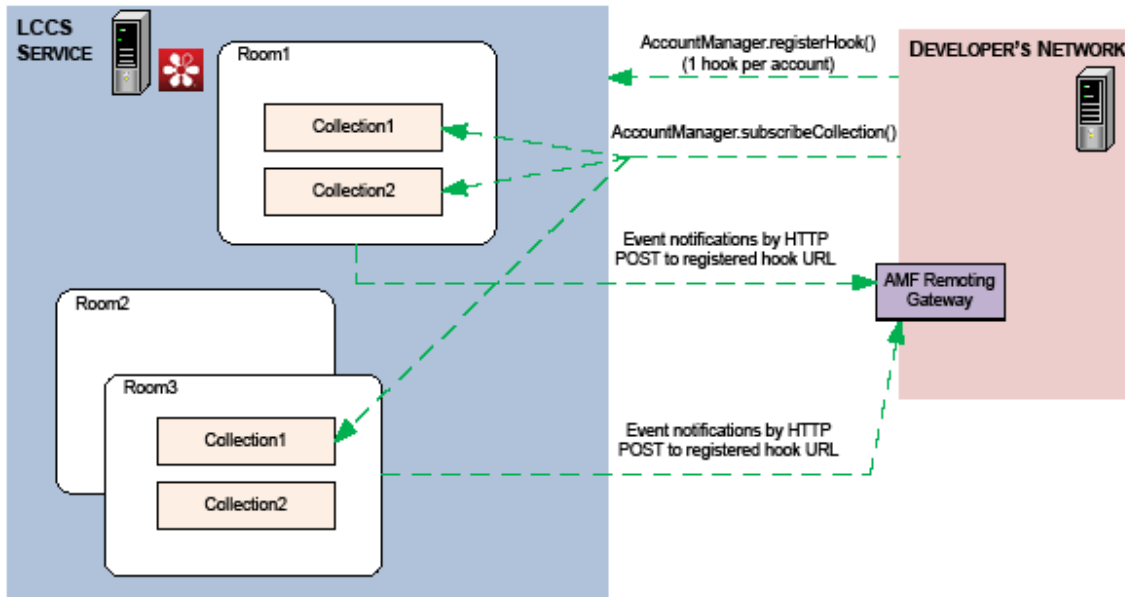
The LCCS SDK provides several sample applications that demonstrate the Server to Server APIs and related configuration. See the `<sdkhome>/com.adobe.lccs/sampleApps/Server2Server` directory.

7.2 High level workflow

The following steps outline how you configure and implement Server to Server functionality:

1. Set up an AMF remoting gateway at a URL in your domain. See [LCCS and AMF](#).
 2. Implement LCCS hook methods on your remoting gateway. See [Implementing LCCS hook methods](#).
 3. Register your remoting URL as the hook for your account. [Registering your hook](#).
 4. Subscribe to collections within one or more LCCS rooms. [Subscribing to collections](#).
 5. Receive event notifications for all subscriptions at the hook URL you provided. Event notifications include a data payload describing the action. Notifications arrive via HTTP POST using AMF. See [Real-time notifications with hooks and subscriptions](#).
 6. Process event payloads locally.
 7. Call `AccountManager.publishItem()` if you wish to respond to events with your processed results. See [AccountManager.publishItem](#).
LCCS distributes your response payload to its routing list.
- [Implementing LCCS hook methods](#)
 - [Registering your hook](#)
 - [Subscribing to collections](#)

The following diagram shows the architecture underlying this workflow:



Implementing LCCS hook methods

LCCS sends notifications when a server or Flash clients call certain methods. The following table shows the hook notification methods and the associated method calls that trigger them:

Hook methods and triggering methods

Hook notification method	Triggering method
receiveItem()	publishItem()
receiveNode()	createNode()
receiveNodeConfiguration()	setNodeConfiguration()
receiveNodeDeletion()	removeNode()
receiveItemRetraction()	retractItem()
receiveUserRole()	setUserRole()

When a callback arrives, your application interprets the action and payload according to your application context. For example, apply the appropriate logic for your application. If your use case requires it, you can then return a payload by calling `publishItem()`. For API reference, see [Hook APIs](#).

Registering your hook

Use the following APIs for hook registration:

```
AccountManager.registerHook(callbackURL, securityToken); //Account hook
AccountManager.unregisterHook();
```

The `registerHook` API provides a callback URL to which LCCS posts notifications. Each account calls `registerHook()` only once. You can use the optional security token parameter to pass a token to LCCS. LCCS caches the token and returns it with each callback. Your token is opaque to the server and can use any format you desire (for example, you could pass a GUID). The `unregisterHook` function is equivalent to calling `registerHook` with a null callback URL. Unregistering your account hook cancels all notifications to your server.

Subscribing to collections

Each room contains one or more collections. You determine which notifications you receive by subscribing to collections within the room. To select the collection whose notifications interest you, use the `subscribeCollection` API. When an event occurs on the selected collection (for example, "chat" or "UserManager"), the server calls your callback URL with a payload describing the event.

```
AccountManager.subscribeCollection(room, collectionName, nodeNames=null);
AccountManager.unsubscribeCollection(room, collectionName);
```

The callback URL and subscriptions are persistent within LCCS. In other words, you subscribe to a collection within a particular room only once. LCCS remembers the subscription for as long as that room exists on the service.

Subscriptions are made only at the collection level. To subscribe to an entire room or application, subscribe to each collection individually. However, this total subscription practice is discouraged: Most developers don't need every transaction event, and too many notifications can degrade service performance.

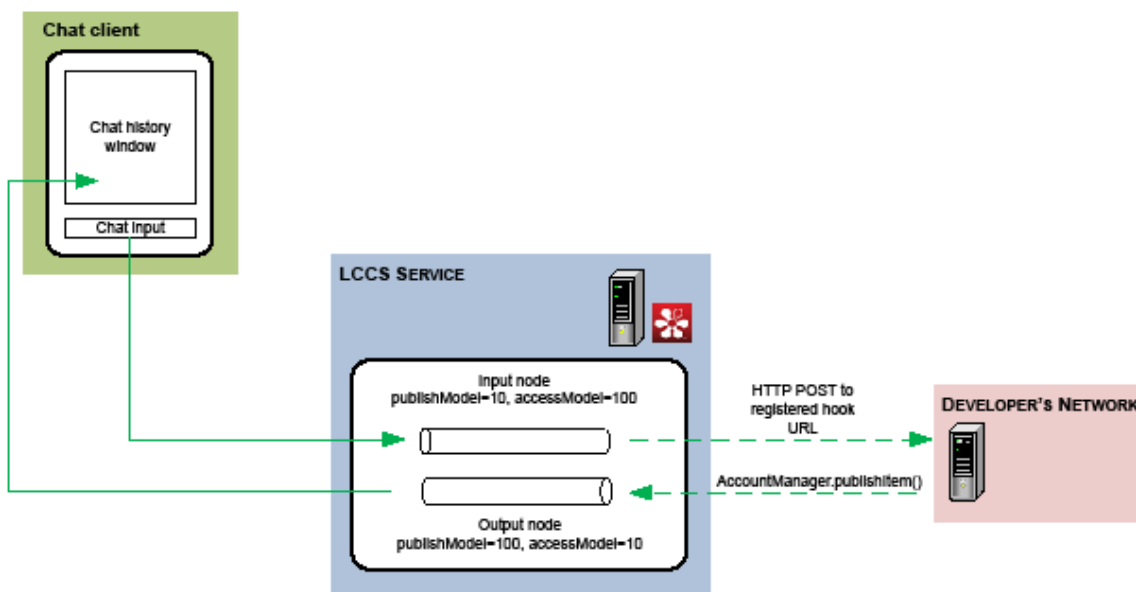
7.3 Revisiting the chat example

When you implement moderated chat, you subscribe to a chat room to receive notifications. A user posts a chat, and the chat client forwards the message to an input node on the collection. The input node has a high security setting for `accessModel` and a low setting for `publishModel`. These settings mean that a user can publish a comment but cannot read any message that is on its way to the LCCS server.

LCCS sends a notification to your server each time a user commits a message. The notification payload is the user's message. Because `accessModel` is high, only your server sees the message.

You apply your profanity filter to the message. You then call `AccountManager.publishItem()` to send the revised message to an output node on LCCS. LCCS remembers the original author of the message. It recognizes you as a "super server" and publishes the revised message under the original author's name. The output node, which represents the chat history panel, carries a high `publishModel` setting and a low `accessModel` setting. These settings mean that the user can see the message in chat history but cannot change the history. Only your server has this ability in this scenario. For more information on `publishModel` and `accessModel` settings, see [UserRoles](#).

The following diagram shows the moderated chat workflow:



Moderated chat workflow

The LCCS SDK provides a moderated chat example. See the `<sdkhome>/com.adobe.lccs/sampleApps/Server2Server/ModeratedChat_PHP` directory.

7.4 Server to Server API Reference

Using the LCCS server libraries

With LCCS server libraries, developers log into their accounts using an `accountManager` object:

```
var accountManager = new com.adobe.lccs.AccountManager(p_accountURL);
accountManager.login(p_username, p_password);
```

For strongly typed languages such as Java, the server libraries include classes that represent datatypes used in the ActionScript SDK. For example, the Java libraries include classes based on `MessageItem` and `NodeConfiguration` ActionScript classes. Classes based on ActionScript objects support deserialization through `readValueObject()` methods. These methods convert virtual objects caught in hooks to strong ActionScript types. For less strongly typed languages, flexible associative arrays represent these classes.

Hook implementation

To use LCCS hooks, you implement methods in your AMF gateway that correspond to the names of the LCCS hooks. You must provide a

method for each hook name. Each hook call contains parameters for the room, the security token, and the data payload. The LCCS server libraries provide a skeleton class for handling the complete set of hooks. See `<sdkhome>/com.adobe.lccs/serverintegration/<language>/rtchooks.<ext>`

Server to Server API categories

Server to Server APIs fall into four groups:

- [Calls](#) (performing operations on the service)
- [Queries](#) (fetching information from the service)
- [Hook APIs](#) (notifications of actions occurring on the service)

These APIs mirror the base set of calls and responses used by the Flash SDKs.

7.4.1 Calls

All calls are asynchronous. Other than throwing an error if the call itself fails, each call returns its results only through the corresponding hook.

Each of these methods sends information that identifies the object on which the action occurs. Most of the calls provide room, collection, and node names. The `setUserRole()` method provides the room name, user name, and an integer that identifies the user's role. The hook methods forward this context information to your server, along with any applicable payload in the form of a virtual object. The public libraries for your scripting language provide AMF translators that deserialize the payload into ActionScript objects.

- [AccountManager.registerHook](#)
- [AccountManager.unregisterHook](#)
- [AccountManager.subscribeCollection](#)
- [AccountManager.unsubscribeCollection](#)
- [AccountManager.publishItem](#)
- [AccountManager.retractItem](#)
- [AccountManager.createNode](#)
- [AccountManager.setNodeConfiguration](#)
- [AccountManager.setUserRole](#)
- [AccountManager.removeNode](#)

AccountManager.registerHook

Signature

```
AccountManager.registerHook(callbackURL, securityToken);
```

Description

Activates hook notifications for the account, and specifies a callback URL to use for receiving any notifications. The optional `securityToken` parameter can be used to specify a token of the caller's choosing. If the token is specified, LCCS passes it back with each hook notification to identify that the hook originated from LCCS. Tokens can use any format (for example, a GUID) and can be any length. LCCS stores the token as a private, opaque cookie and returns it as the `securityToken` parameter in notification callbacks.

Parameters

Parameter name	Description
<code>callbackURL</code>	The URL on the developer's remoting gateway to which LCCS should send hook notifications
<code>securityToken</code>	(optional) A token, in any format, that is passed back to the remoting server with each notification

AccountManager.unregisterHook

Signature

```
AccountManager.unregisterHook();
```

Description

Removes the account hook, and disables all hook notification for the account.

AccountManager.subscribeCollection

Signature

```
void AccountManager.subscribeCollection(String room, String collectionName, String[] nodeNames)
```

Description

Activates notifications for the given collection.

Parameters

Parameter name	Description
room	String; the name of the room to subscribe to
collectionName	String; the name of the collection to subscribe to in the given room
nodeNames	(optional) String array; names of nodes with the given collection to subscribe to

AccountManager.unsubscribeCollection

Signature

```
void AccountManager.unsubscribeCollection(String room, String collectionName);
```

Description

Disables all notifications for the given collection.

Parameters

Parameter name	Description
room	String; name of the room to unsubscribe
collectionName	String; name of the collection in the given room to unsubscribe

AccountManager.publishItem

Signature

```
void AccountManager.publishItem(  
    String room,  
    String collectionName,  
    String nodeName,  
    MessageItem item,  
    Boolean overwrite);
```

Description

Publishes an item at the collection or node specified. When republishing an item from a user, this method allows the server to publish the item under the original publisher's name.

Parameters

Parameter name	Description
room	String; the name of the room to which the item is to be published
collectionName	String; the name of the collection within the given room to which the item is to be published
nodeName	String; the name of the node within the given collection to which the item is to be published
item	MessageItem object; the content to be published
overwrite	Boolean; if true, the server can overwrite the original message. See <code>CollectionNode.publishItem()</code> in the <i>LiveCycle Collaboration Services API Reference</i> .

AccountManager.retractItem

Signature

```
void AccountManager.retractItem(  
    String room,  
    String collectionName,  
    String nodeName,  
    String itemID);
```

Description

Removes the given item from the service.

Parameters

Parameter name	Description
room	String; the name of the room from which the item is to be retracted
collectionName	String; the name of the collection within the given room from which the item is to be retracted
nodeName	String; the name of the node within the given collection from which the item is to be retracted
itemID	String;

AccountManager.createNode

Signature

```
void AccountManager.createNode(  
    String room,  
    String collectionName,  
    String nodeName,  
    NodeConfiguration nodeConfig = null);
```

Description

Creates a new node at the specified location, optionally with a specified configuration.

Parameters

Parameter name	Description
room	String; the name of the room in which the node is to be created
collectionName	String; the name of the collection within the given room in which the node is to be created
nodeName	(optional) String; the name of the node within the given collection in which the node is to be created. If omitted, creates a new collection.
nodeConfig	(optional) NodeConfiguration object; the configuration settings to be applied to the new node

AccountManager.setNodeConfiguration

Signature

```
void AccountManager.setNodeConfiguration(  
    String room,  
    String collectionName,  
    String nodeName,  
    NodeConfiguration nodeConfig);
```

Description

Applies the settings in nodeConfig to the specified node.

Parameters

Parameter name	Description
room	String; the name of the room in which the node is to be configured
collectionName	String; the name of the collection within the given room in which the node is to be configured
nodeName	String; the name of the node within the given collection in which the node is to be configured
nodeConfig	NodeConfiguration object; the configuration settings to be applied to the specified node

AccountManager.setUserRole

Signature

```
void AccountManager.setUserRole(
    String room,
    String userID,
    int role,
    String collectionName=null,
    String nodeName=null);
```

Description

Sets the role of the given user, either on the room as a whole (if collectionName and nodeName are omitted), or on a particular collection or node. For information on user roles, see [UserRoles](#).

Parameters

Parameter name	Description
room	String; the name of the room in which the specified user resides
userID	String; ID of the user whose role is to be set
role	int; developer-defined integer value that indicates the role to be applied to the specified user
collectionName	(optional) String; the name of the collection within the given room in which the specified user resides
nodeName	(optional) String; the name of the node within the given collection in which the specified user resides

AccountManager.removeNode

Signature

```
void AccountManager.removeNode(
    String room,
    String collectionName,
    String nodeName = null);
```

Description

Removes the specified node. If the nodeName parameter is omitted, the entire collection is removed.

Parameters

Parameter name	Description
room	String; the name of the room in which the node is to be removed
collectionName	String; the name of the collection within the given room in which the node is to be removed
nodeName	(optional) String; the name of the node within the given collection in which the node is to be removed

7.4.2 Queries

Queries are synchronous. They block further processing until the service returns a result.

- [AccountManager.getHookInfo](#)

- [AccountManager.getNodeConfiguration](#)
- [AccountManager.fetchItems](#)

AccountManager.getHookInfo

Signature

```
String AccountManager.getHookInfo();
```

Description

Returns a string containing the hook URL and security token for the account.

Return value

A string containing the hook URL and a security token (if provided). If no hook is found, returns a string with two empty parts.

AccountManager.getNodeConfiguration

Signature

```
NodeConfiguration AccountManager.getNodeConfiguration(  
    String room,  
    String collectionName,  
    String nodeName);
```

Description

Returns the configuration of the specified node.

Parameters

Parameter name	Description
room	String; the name of the room in which the node resides
collectionName	String; the name of the collection within the given room in which the node resides
nodeName	String; the name of the node within the given collection for which configuration information is returned

Return value

A `NodeConfiguration` object containing the given node's configuration settings.

AccountManager.fetchItems

Signature

```
MessageItem[] AccountManager.fetchItems(  
    String room,  
    String collectionName,  
    String nodeName,  
    String[] itemIDs = null);
```

Description

Returns an array of `MessageItem` objects that correspond to the specified item IDs on the given node. Any item ID not found on the service is simply ignored. If the `itemIDs` array is omitted, this method returns the full list of items in the node.

Parameters

Parameter name	Description
room	String; the name of the room from which the items are to be fetched
collectionName	String; the name of the collection within the given room from which the items are to be fetched

nodeName	String; the name of the node within the given collection from which the items are to be fetched
itemIDs	(optional) Array of strings specifying the IDs of the items to be fetched

Return value

An array of `MessageItem` objects that either correspond to the specified item IDs on the given node (when `itemIDs` is provided), or represent the full list of items in the node (when `itemIDs` is omitted).

7.4.3 Hook APIs

Hook functions must be implemented on the remoting gateway. These functions represent the interface that an LCCS collection listener must implement.

All callbacks accept a security token (used to validate the sender) and the name of the room that generated the notification.

- `receiveItem`
- `receiveNode`
- `receiveNodeConfiguration`
- `receiveNodeDeletion`
- `receiveItemRetraction`
- `receiveUserRole`

receiveItem

Signature

```
void receiveItem(
    String securityToken,
    String roomName,
    String collectionName,
    Object messageItemVO);
```

Description

Called when `publishItem()` is triggered on a subscribed collection node. The `messageItemVO` parameter contains the pertinent node ID. All messages - private or public - are sent to server listeners.

The `messageItemVO` parameter contains a simple representation of a `messageItem` object, similar to a plain Java object (POJO).

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed in the <code>registerHook()</code> call
roomName	String; the name of the room on which the item is published
collectionName	String; the name of the collection within the given room on which the item is published
messageItemVO	Object; a simple representation of the <code>messageItem</code> object that was published

receiveNode

Signature

```
void receiveNode(
    String securityToken,
    String roomName,
    String collectionName,
    String nodeName,
    Object nodeConfigurationVO);
```

Description

Called when `createNode()` is triggered on a subscribed collection.

The `nodeConfigurationVO` parameter contains a simple representation of a `nodeConfiguration` object, similar to a plain Java object (POJO).

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed in the <code>registerHook()</code> call
roomName	String; the name of the room on which the node was created
collectionName	String; the name of the collection within the given room on which the node was created
nodeName	String; the name of the node that was added to the given collection
nodeConfigurationVO	Object; a simple representation of the <code>nodeConfiguration</code> object (if any) that was passed to <code>createNode()</code>

receiveNodeConfiguration

Signature

```
void receiveNodeConfiguration(  
    String securityToken,  
    String roomName,  
    String collectionName,  
    String nodeName,  
    Object nodeConfigurationVO);
```

Description

Called when `setNodeConfiguration()` is triggered on a subscribed collection node. `nodeConfigurationVO` contains a simple representation of a `nodeConfiguration` object, similar to a plain Java object (POJO).

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed in the <code>registerHook()</code> call
roomName	String; the name of the room on which the node was configured
collectionName	String; the name of the collection within the given room on which the node was configured
nodeName	String; the name of the node that was configured on the given collection
nodeConfigurationVO	Object; a simple representation of the <code>nodeConfiguration</code> object (if any) that was passed to <code>setNodeConfiguration()</code>

receiveNodeDeletion

Signature

```
void receiveNodeDeletion(  
    String securityToken,  
    String roomName,  
    String collectionName,  
    String nodeName);
```

Description

Called when `removeNode()` is triggered on a subscribed collection node.

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed to <code>registerHook()</code>
roomName	String; the name of the room on which the node was removed

collectionName	String; the name of the collection within the given room on which the node was removed
nodeName	String; the name of the node that was removed from the given collection

receiveItemRetraction

Signature

```
void receiveItemRetraction(
    String securityToken,
    String roomName,
    String collectionName,
    String nodeName,
    Object messageItemVO);
```

Description

Called when `retractItem()` is triggered on a subscribed `collectionNode`. The `messageItemVO` parameter contains a simple representation of a `messageItem` object, similar to a plain Java object (POJO).

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed in the <code>registerHook()</code> call
roomName	String; the name of the room on which the item was retracted
collectionName	(optional) String; the name of the collection within the given room on which the item was retracted
nodeName	(optional) String; the name of the node within the given collection on which the item was retracted
messageItemVO	Object; a simple representation of the <code>messageItem</code> object that was retracted

receiveUserRole

Signature

```
void receiveUserRole(
    String securityToken,
    String roomName,
    String collectionName = null,
    String nodeName = null,
    String userID,
    int p_role);
```

Description

Called when `setUserRole()` is triggered on the user specified by `userID` in a subscribed collection.

Parameters

Parameter name	Description
securityToken	String; the token (if any) that was passed in the <code>registerHook()</code> call
roomName	String; the name of the room on which the user role was set
collectionName	(optional) String; the name of the collection within the given room on which the user role was set
nodeName	(optional) String; the name of the node within the given collection on which the user role was set
userID	String; the ID of the user whose role was set
p_role	int; developer-defined ID of the role that was assigned to the given user

08 Audio, WebCam, and Screen Sharing

Audio and Video Streams

The ability to publish, receive, and manage A/V streams across an unlimited number of users and user groups is one of LCCS's strongest assets. Certainly audio, web camera, and screen sharing capabilities lend a richness to collaborative applications that isn't provided by chat and file sharing.

 For API-level information, refer to `com.adobe.rtc.collaboration` in the SDK's LCCS API Reference.

LCCS provides the means for you to build fully customized audio/video workflows in your applications. The SDK will allow you to :

- Manage publication and subscription to streams via
 - **Publisher** components which handle capturing A/V streams from your devices (microphones, web cameras, screens) and broadcasting them to the whole room or to a subset of the users therein
 - **Subscriber** components which can receive the A/V streams and play them, whether from everyone in the room or from a specified set of users
- Broadcast via Peer-to-Peer or hub and spoke connections, depending on your users' capabilities. For details, see [RTMFP vs. RTMP](#)
- Broadcast via application multicasting in Flash Player 10.1.
- Manage who may publish or subscribe to streams via a publish-access model that leverages user roles you define

Default behavior

The components discussed in this chapter allow sharing of **Audio**, **Web Cameras**, and **Screens** . Each sharing capability is made up of **publisher** and **subscriber** components, each of which use the StreamManager to notify participants in the room that there are A/V streams to access.

All publishers publish StreamDescriptors to the StreamManager which notifies subscribers that a new stream has been initiated. The default behavior is as follows:

- A StreamDescriptor identifies the stream (its location).
- By default, stream publishers publish to everyone in the room. They automatically handle notifying clients that a stream exists as well as management of the physical stream.
Because it is often desirable to limit access to a subset of the room members, publisher components have an API for setting and getting a subset of user IDs. Simply populate their recipientIDs array with the user IDs to whom you wish to publish.
- Users with the role UserRoles.PUBLISHER or greater may publish, and all users with role of greater than UserRoles.VIEWER are able to subscribe to these streams. The default roles and behavior may be customized.
- Subscribers automatically subscribe to any incoming streams, and play them in the manner most appropriate (for example, by playing the audio through the speakers, or rendering web camera video in a rectangular viewing area).
- The developer can specify a set of publisherIDs (userIDs of the publishers they wish to subscribe to) on subscriber components to narrow down the number of streams. If no streams are specified, it plays all the streams available in the streamManager.

8.1 Audio


The AudioPublisher and AudioSubscriber components are the principal ways to add audio communication (also known as Voice over IP, or VoIP) to a collaborative application. As they're meant to fit within highly customized workflows, these components are completely "headless" - that is, they don't have any user interface whatsoever, and the desired UI is to be built by the developer of the application.

However, hooking the APIs of these components up to a UI couldn't be much easier - publishing audio is a simple matter of instantiating an audioPublisher and calling `audioPublisher.publish()`; In the Flash Player, this will cause the publishing user to see the following :



Stream prompt dialog

Upon accepting this dialog, the user in question will have audio captured from their computer microphone and broadcast to any subscribers listening.

 It is highly recommended that your users use microphone-and-speaker headsets when publishing and subscribing to VoIP streams - this will greatly reduce echo and improve audio quality.

The audioPublisher also provides a variety of APIs for controlling the broadcast settings of the audio - microphone gain, silence level and timeOut.

In order to receive audio, simply instantiate an `audioSubscriber`, which will receive any audio streams and play them through the computer speakers. This component is also "headless", with any UI workflow to be designed and coded by the developer. The subscriber also provides APIs for setting the local volume of the various streams it is receiving.

A word about codecs

A "codec" is a library used to encode audio from the microphone for sending over a stream, and for decoding that audio stream for playing through the computer's speakers. The Flash Player provides 2 codecs :

1. Player 6 to Player 9 : The NellyMoser codec is used.
2. Player 10.0 and above : The Speex coded is used.

It is advisable that if you can deploy your application to use Player 10 or higher, the Speex codec is automatically the default choice for LCCS, which presents a noticeable improvement in audio quality for VoIP conversations (Speex is a codec especially tuned to the frequency domain of the human voice, and is more tolerant of packet loss than NellyMoser).

8.2 Web Camera

As with other A/V components, LCCS provides a **publisher** component for capturing and broadcasting a user's Web Camera video screen, and a **subscriber** component for receiving the publisher's stream and rendering it. In this case, the publisher component is "headless", similar to the `AudioPublisher`, in that it has no user interface. The `WebcamSubscriber` has a rudimentary user interface - it displays each incoming video stream as a separate rectangle in a tiled layout within the subscriber component's dimensions.

WebcamPublisher

Similar to the `AudioPublisher`, the `WebcamPublisher`'s API makes it easy to add video publishing - simply instantiate the component and call `webcamPublisher.publish()`; In the Flash Player, this will cause the publishing user to see the following :



The Flash Player prompts for Audio Video capture

Upon accepting this dialog, the user in question will have video captured from an attached web camera and broadcast to any subscribers listening.

The `WebcamPublisher` also provides a variety of APIs for controlling the broadcast quality of the video, including the fps (frames per second captured) and quality (compression factor).

WebcamSubscriber

In order to receive web camera video, simply instantiate a `WebcamSubscriber`, add it to a display list, and give it a size (set its width and height). The subscriber will automatically display all incoming webcam streams available to it.

In the interests of saving bandwidth, the `WebcamSubscriber` will NOT subscribe to the stream being published from the current users computer, if any. Instead, assign the `WebcamPublisher` to the subscriber so that video can be captured locally, like so :

```
myWebcamSubscriber.webcamPublisher = myWebcamPublisher;
```

By default, the subscriber will display any available incoming streams and lays them out as a set of tiles within its dimensions. If you'd like to customize this layout, the recommended approach is to use multiple subscribers, each restricted to show only one stream. To restrict which streams the subscriber displays, use the `publisherIDs` API, like so :

```
myWebcamSubscriber.publisherIDs = [userID];
```

This will ensure that only one stream (corresponding to one publisher) will be displayed. From there, one subscriber can be instantiated per stream - to find all available streams, the `StreamManager` can be used :

```
var streamIDs:Object =  
myConnectSession.streamManager.getStreamsOfType(StreamManager.CAMERA_STREAM);
```

From here, you could iterate over the `streamIDs` object and create one `WebcamSubscriber` per stream, and fetch stream details via

```
var streamDesc:StreamDescriptor = myConnection.streamManager.getStreamDescriptor(streamID);
```

The StreamDescriptor contains details such as streamPublisherID which can be used for assigning WebcamSubscriber.publisherIDs. How you want to lay out each WebcamSubscriber is then totally up to you.

The WebCamera Component

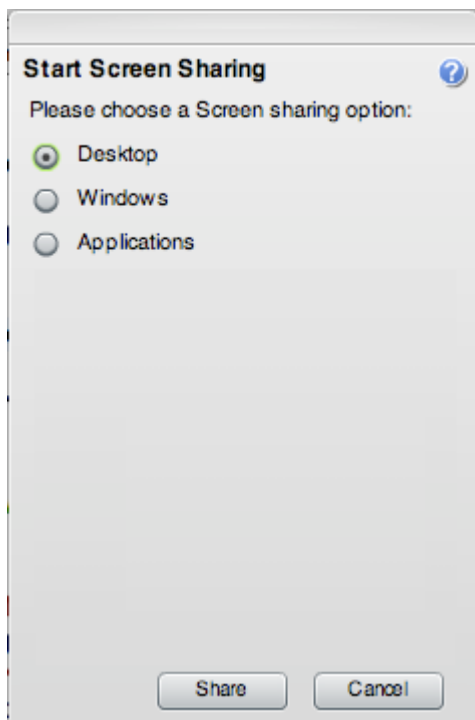
The WebCamera component is a "pod" level component, meant to provide an easy way to include a web cam publish and subscribe workflow into your application with minimal (as few as one!) lines of code. In following the philosophy of such "pods", the WebCamera provides limited customizability - it's really meant as an easy entry point to the publisher and subscriber components. It's important to note that this component is built completely from the WebcamPublisher and WebcamSubscriber - you can open up its source and see the way in which it uses them. If you'd like to customize your web camera workflow, you could either repurpose or extend the WebCamera code, or build your own workflow from scratch using the publisher and subscriber components, which are lower-level building blocks useful for this purpose.

8.3 Screen Sharing

As with other A/V components, LCCS provides a publisher component for capturing and broadcasting a user's screen, and a subscriber component for receiving the publisher's stream and rendering it. In this case, the publisher component is "headless", similar to the AudioPublisher, in that it has no user interface. The ScreenShareSubscriber has a rudimentary user interface - it displays one incoming video stream within the subscriber component's dimensions.

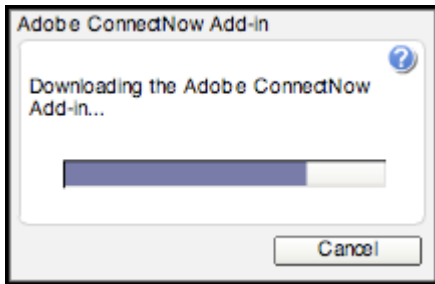
ScreenSharePublisher

Similar to the AudioPublisher, the ScreenSharePublisher's API makes it easy to add screen publishing - simply instantiate the component and call screenSharePublisher.publish(); In the Flash Player, this will cause the publishing user to see the following :



The user is given the choice to capture all of the screen, or specific applications or windows within it. If the user chooses specific apps or windows, any screen real-estate outside the selected apps or window appears as cross-hatched shading to subscribers.

It's worth noting that capture of the screen is not a capability of the Flash Player itself; rather, a small addin component is downloaded, installed, and launched under the control of the ScreenSharePublisher. This addin is available for Windows XP (or higher) and Mac OSX in the browser plugin and AIR.



Note that for the first release, this addin is labeled as "Adobe ConnectNow" for the download. In the future, this naming may change.

Upon accepting any download and the permissions dialog, the user in question will have video captured from the desktop screen (or specific app or window, as requested) and broadcast to any subscribers listening.

The ScreenSharePublisher also provides a variety of APIs for controlling the quality of the screen share, such as fps (frames per second), keyFrameInterval, and quality. Set these properties of the publisher before calling publish() to specify settings. The addin will be closed down if ScreenSharePublisher.stop() is called or if the user ends their ConnectSession by logging out or closing the browser or app that contains the publisher.

ScreenShareSubscriber

In order to receive screen capture video, simply instantiate a ScreenShareSubscriber, add it to a display list, and give it a size (set its width and height). The subscriber will automatically display the first screen share stream available to it. If there are multiple publishers broadcasting their screens, developers can specify which one to display in a given subscriber by setting

```
myScreenShareSubscriber.publisherID = publisherID;
```

In order to find a list of streams available for subscription, use

```
var streamIDs:Object =  
myConnectSession.streamManager.getStreamsOfType(StreamManager.SCREENSHARE_STREAM);
```

This will return an object with the streamIDs of all screenshare streams. To fetch the descriptors for each stream, use

```
var streamDesc:StreamDescriptor = myConnectSession.streamManager.getStreamDescriptor(streamID);
```

This will give you the details of a given stream, such as the streamPublisherID, for use specifying the stream for the ScreenShareSubscriber.

8.4 RTMFP vs RTMP

Real Time Media Flow Protocol (RTMFP) is a new protocol from Adobe that enables direct end user to end user peering communication between multiple Adobe Flash Players and applications built on LCCS or the Adobe AIR framework. RTMFP enables real time communication applications such as social networks and multi-user games to deliver high quality solutions. RTMFP enables end users to connect and communicate directly with each other using their computer's microphone and webcam.

RTMFP is based on the User Datagram Protocol (UDP), whereas RTMP is based on the Transmission Control Protocol (TCP). UDP-based protocols have many advantages over TCP-based protocols when delivering live streaming media, including decreased latency and overhead as well as greater tolerance for missing packets. A server side connection is always required to establish the initial connection between the end users and can be used to provide server side data execution or gateways into other systems.

RTMP (TCP) uses a hub and spoke model which is less than ideal for live streaming. RTMFP doesn't force audio and video packets to be retransmitted in the case of failure. It continues playing despite gaps in the data - this means that the stream plays more smoothly in the case of small-scale packet loss. It also supports client to client streaming. Client to client streaming is not only free; it's also faster, because it isn't routed through the LCCS service. However, since a high number of users, a firewall, or other network conditions may prohibit the use of RTMFP, LCCS has been designed to automatically switch based on conditions: if RTMFP is enabled it uses it; if not, it uses RTMP.



RTMFP is the preferred protocol and always a better choice if using the required Flash player 10 SWC is acceptable. Because LCCS is smart enough to automatically switch between the two protocols as needed, you can always leverage RTMFP's capabilities whenever it's possible to do so and use RTMP as a fallback.

To use RTMFP, set your project to use the Flash Player 10 or 10.1 SWC files.

RTMFP pros

- **Client to client streaming:** C2C reduces latency, since there are fewer hops from source to destination. The advantages for the

LCCS service as well as you, is that the service can operate with less infrastructure to allow you to stream, thereby increasing reliability.

- **Speex access through Flash Player 10:** Speex is a better codec than the NellyMoser codec in older Flash Players because it encodes voice with higher quality and at a lower bitrate. It's also more compatible with UDP so that it handles lossiness relatively gracefully.
- **Reduced bandwidth costs:** RTMFP reduces the bandwidth costs for direct real time communication solutions such as audio and video chat and multiplayer games. Because RTMFP flows data between end user clients rather than the server, server bandwidth is reduced, thereby making solutions are less expensive to scale.
- **Increased speed of delivery:** RTMFP also increases the speed of delivery through the use of UDP. UDP is a more efficient (but less reliable) way to send video and audio data over the Internet because it reduces the penalties associated with missing, dropped, or out of order packets.
- **Superior connection error recovery:** RTMFP has two features that may help to mitigate the effects of connection errors:
 - **Rapid Connection Restore:** Connections are reestablished quickly after brief outages such as when a wireless network connection experiences a dropout. After re-connection, the connection has full capabilities instantly.
 - **IP Mobility:** Active network peer sessions are maintained even if a client changes to a new IP address. For example, when a laptop on a wireless network is plugged into a wired connection and receives a new network address.

RTMFP cons

Requires Flash player 10. This is usually a good thing.

09 Peer-to-Peer Data Messaging and AV Multicasting

Flash Player 10.1 provides functionality that helps developers build improved peer-to-peer (P2P) applications. With Flash Player 10.1 and the LCCS 10.1 SWC file, you can enable P2P audio/video multicasting as well as P2P data messaging.

- [Deploying the LCCS 10.1 SWC file](#)
- [Using P2P multicast with audio and video streams](#)
- [Using P2P data messaging](#)

Deploying the LCCS 10.1 SWC file

Deploying the LCCS 10.1 SWC file involves downloading the correct file and adjusting your project settings to use it, rather than an earlier version.

To use the LCCS 10.1 SWC file

1. Download and install Flash Player 10.1 from Adobe Labs at <http://labs.adobe.com/downloads/flashplayer10.html>. For best testing results, choose the debug version.
2. From the same website, download the 10.1 Flash Player Global SWC file. Use this file in your projects instead of the default file. To remove the default playerglobal.swc file in a Flash Builder project, select Properties > Flex Build Path. On the Library tab under Build Path Libraries, expand the Flex SDK library entry and remove the default playerglobal.swc.
3. In any Flash Builder project that uses the LCCS 10.1 SWC file, select Project Properties > Flex Compiler. In the HTML Wrapper panel, set Required Flash Player Version to 10.1.

Using P2P multicast with audio and video streams

With the LCCS 10.0 SWC file, the number of recipients that can receive a publisher's P2P audio/video stream is limited. This limitation arises because, with RTMFP in Flash Player 10.0, each recipient requires an additional slice of the publisher's bandwidth. Eventually, the stream exhausts all available bandwidth. The publisher usually needs to revert to hub-and-spoke distribution after only three recipients are connected.



You can set the actual recipient limit using the `StreamManager.maxP2PStreamPublish` property.

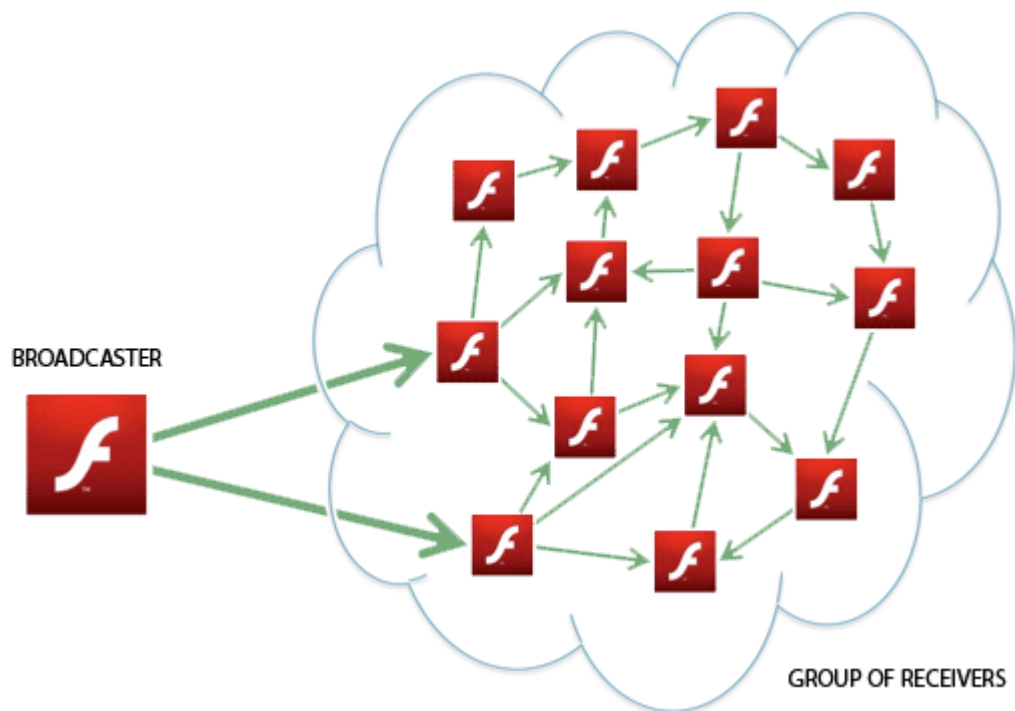
With the LCCS 10.1 SWC file, A/V streaming can take advantage of P2P multicasting. P2P multicasting is possible thanks to RTMFP enhancements in Flash Player 10.1. With P2P multicasting, recipients share the distribution load by sending A/V stream data directly to other recipients. This shared distribution model removes limits on the number of users who can receive the stream.

To enable A/V multicasting, a user with OWNER level permission sets the `streamManager.streamMulticast` property to true:

```
myConnectSession.streamManager.streamMulticast = true;
```

After this property is set, you can use the `AudioPublisher` and `WebcamPublisher` components provided in the LCCS.swc file. Multicast is managed automatically by the service.

The following diagram illustrates the P2P multicast model:



Using P2P data messaging

P2P data messaging offers significant benefits in certain messaging situations. It works well in scenarios that require ultra-low-latency messaging. It's also cost-effective: messages travel free of charge outside the service. However, P2P messaging also presents a few caveats:

- All messages are transient, which means they aren't persistent on the service. Thus, users entering a room in mid-session do not receive messages sent prior to their arrival.
- P2P data messaging provides no security on nodes. Essentially, any user can publish and subscribe to such nodes.
- P2P messaging offers no contention resolution. Two clients can send each other messages to update a particular item, and neither can be assured of who actually last updated it.

To enable peer-to-peer data messaging in a specific node, set `NodeConfiguration.p2pDataMessaging` to `true`:

```
nodeConfig.p2pDataMessaging = true;
```

This setting causes all messages in that node to be sent via peer-to-peer multicast.

The first time a user initiates or receives P2P data messaging, Flash Player presents a dialog box asking whether to allow or block P2P traffic. To participate in a room with P2P data messaging, the user must select **Allow**. For best results, the user should also select the checkbox to remember this choice.

10 Recording and Playback

The ability to record and play back collaborative sessions is an extremely powerful facility provided by the Collaboration Service. Everything from chat, camera, audio, screensharing, whiteboard, as well as your own custom collaborative components, can be recorded and played back via the service. The service takes care of a lot of the messy details around recorded streams, leaving you with more time to work on customizing the precise experience you want for your end users.

10.1 Overview

Capabilities

- Recording and playback of :

- AudioPublisher and AudioSubscriber
- WebcamPublisher, WebcamSubscriber, and the WebCam pod.
- ScreenSharePublisher and ScreenShareSubscriber
- SharedCursorPane
- All pod components, such as SharedWhiteBoard, SimpleChat, Note
- Any custom components that you include, based on SharedModels such as CollectionNode
- Fully synchronized seek within your playback.
- Programmatic control of your playback application. Make changes to the layout, remove features, remix the content altogether.
- Since we store recordings on your premises, you can extract individual audio/video streams and use as needed.
- Data messaging is stored as data messaging. From your copy of the recording, you can pull out those data messages for analysis - chat message logs, transcripts, etc.

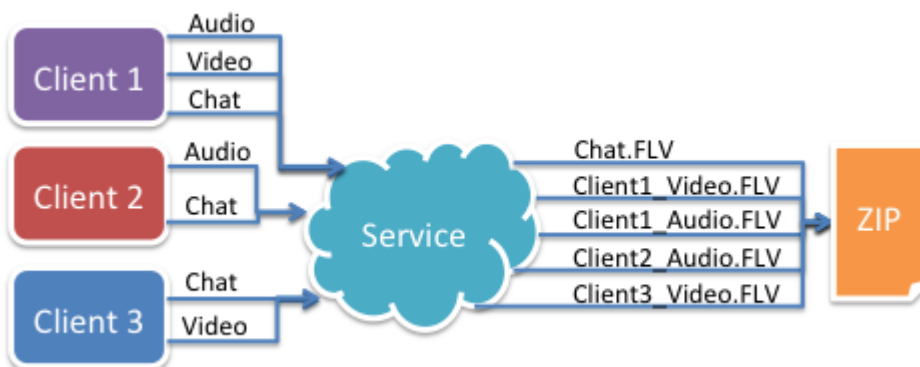
Approach

Instead of producing a monolithic "screen-capture" video of a recording, the collaboration service allows you to build **playback applications** of your recordings, which leverage the service to create your own customized experience for playback. Rather than "dumb" video, playback is a working app, which you customize to your own specifications and functionality.

The recording and playback feature works by capturing all audio-video and data traffic which passes through the service, and writing it into FLV streams.

i During recording, all peer-to-peer streaming is shifted automatically to hub and spoke, so that streams may be captured.

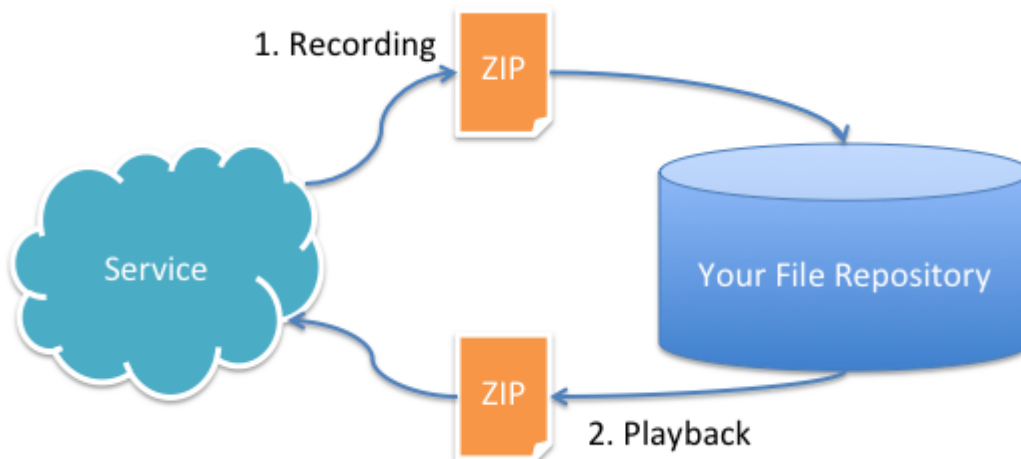
Each type of traffic is recorded separately - that is, each audio stream, video stream, and collectionNode data stream is recorded to its own FLV file.



Note from the diagram above, that the recording consolidates data streams, such as chat, but that it doesn't do so for audio/video streams - this means that there is one resultant chat stream in the recording, and one audio and video stream for each user who is publishing. The service keeps audio/video streams separate to conserve on CPU as well as ensure maximum flexibility for playback - you could choose to delete individual user AV streams during playback if desired. The service keeps all streams synchronized during playback.

i Data stream FLVs are named based on the sharedID of the CollectionNode, and audio/video FLVs are named based on UUIDs (equivalent to StreamDescriptor.id).

Once a recording is completed (because either the recording is stopped, or the room has sat empty for more than 5 minutes and shuts down), the collaboration service zips up all FLV files, and PUTs the resultant ZIP file to the developer's WebDAV-compatible file repository.



In order to play back the recording, the service fetches the appropriate recording from the developer's repository, and allows clients to connect to a special "recording playback" room, which streams the recording to the client app, with full synchronization and the ability to

seek.

For more on setting up a repository for storing recordings, see the following section, [Repository Setup](#).

10.2 Repository Setup

The Collaboration Service's recording and playback is designed to store the "archive" of a recording on a customer-provided server. When recording, the service will store the recorded streams locally; when the recording ends it will package them in a zip file and send them to the customer server. When playing back a recording, the service will retrieve the archive from that same repository, uncompress it, and play back the expanded streams.

Setting up a repository server (WebDAV) :

In order to enable recording and playback the developer needs to provide and register a "repository server". The collaboration service will use a simple HTTP/WebDAV PUT to store a recording and a GET to fetch the recording. Any HTTP server that supports PUT and GET of resources can be used as the "repository server" but a server that supports WebDAV out of the box is probably the fastest and easiest way to start.

Two servers that can be used this way are Apache (with mod_dav) and Tomcat (with the embedded WebDAV servlet). Here is how to configure them:

Using Apache with mod_dav

- Install a recent version of Apache (i.e. httpd 2.x)
- Enable mod_dav in /etc/httpd/conf/httpd.conf

```
LoadModule dav_module modules/mod_dav.so
LoadModule dav_fs_module modules/mod_dav_fs.so

#
# WebDAV module configuration section.
#
<IfModule mod_dav_fs.c>
    # Location of the WebDAV lock database.
    DAVLockDB /var/lib/dav/lockdb
</IfModule>
```

- Configure mod_dav and repository folder (i.e. in /etc/httpd/conf.d/dav.conf) :

```
<Directory /var/www/html/dav>
    DAV on
    AllowOverride All
    Options Indexes
    Order deny,allow
    Allow from all
</Directory>
```

- If needed enable stricter security (i.e. add basic authentication, disable indexes, etc.)
- Create the "repository folder" (i.e. /var/www/html/dav)
- Restart Apache, access <http://localhost:8080/dav/> and verify that the access succeeds.

For more information refer to the Apache documentation: http://httpd.apache.org/docs/2.0/mod/mod_dav.html

Using Tomcat with the WebDAV servlet

- Install a recent version of Tomcat (i.e. Tomcat 6.x)
- Create the 'webdav' application:

```
mkdir -p ../tomcat/webapps/webdav/WEB-INF
```

- Configure the 'webdav' servlet and make sure writes are enabled (create and edit ../tomcat/webapps/webdav/WEB-INF/web.xml) :

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
version="2.5">
<servlet>
  <servlet-name>webdav</servlet-name>
  <servlet-class>org.apache.catalina.servlets.WebdavServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>readonly</param-name>
    <param-value>false</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>webdav</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>

```

- If needed enable stricter security (i.e. with an authentication filter, etc.)
- Restart tomcat, access <http://localhost:8080/webdav/> and verify that the access succeeds.

For more information refer to the Apache documentation:

<http://tomcat.apache.org/tomcat-6.0-doc/api/org/apache/catalina/servlets/WebdavServlet.html>

Register the repository endpoint

In order to have the collaboration service send and retrieve recordings to/from your file server, it is necessary to register your repository with the service. This is done by using the server integration libraries provided in the SDK.



To get started with the server integration libraries, see the Deploying Applications section of these docs, specifically [Provisioning Rooms](#), for more detail. All pseudo-code shown below will be Java-oriented. For full JavaDoc API listings, see <SDK Home>/serverIntegration/java/doc/index.html in the folder you downloaded the SDK to. Equivalent APIs are available in the other language libraries provided for server integration.

The pseudo-code below will register a repository for a given developer account :

```

// typically, you only need to instantiate one accountManager per web application - caching one
makes sense.
AccountManager aM = new AccountManager("https:
//collaboration.adobe.livecycle.com/<YOUR-ACCOUNT-NAME>");

aM.login("<YOUR-EMAIL>", "<YOUR-PASSWORD>");
aM.registerRepository("<YOUR-WEBDAV-URL>");

```



Only one repository endpoint is available per account. The code above really only needs to be run once - the account will store this URL from that point forward. To retrieve details on this endpoint URL, use AccountManager's `getRepositoryInfo()` method. To remove any repository, use `unregisterRepository()`.

Notes :

1. Basic authentication will send the credentials in clear so it's suggested that an secure URL is used (i.e. <https://lccs:lccspassword@lccsrepo.com/dav/>) or digest authentication is used.

Archive files

When recording, the recorded streams are saved on the Collaboration Service's local disks and later packaged in a zip file and sent to the repository server.

The archive file name identifies the account, room and archiveID associated to the recording. For example:

```
na2-sdk-233e070b-f1e0-4df5-af1e-8527579cc8a2_x002F__defaultArchive__.zip
```

In this example 'na2-sdk-233e070b-f1e0-4df5-af1e-8527579cc8a2' is the account id (for example for the account 'testaccount'), 'camerarecording' is the room name and '__defaultArchive' is the archiveID (if no id is specified when starting a recording the default name 'defaultArchive' is used).

The _x002F_ that separates the fields is the '/' character encoded as ISO9075.

In the next section, [Starting and Stopping Recording](#), you'll learn how to actually trigger recording, both from the client and the server.

10.3 Starting and Stopping Recording

Once a repository endpoint has been set up, you're ready to actually start recording an application. The Collaboration Service allows you to do this from either the client (with sufficient permissions) or the server.

The 3 essential elements to specifying a recording are :

1. A **roomURL**. This specifies the room that will be recorded.
2. An **archiveID**. This specifies the "instance name" of the recording - since a room may have multiple recordings associated with it, specifying a unique ID for each recording identifies which recording is desired. By default, `_defaultArchive_` is used.
3. **guestsAllowed**. This optional setting specifies whether or not the resultant recording should allow anyone to view it (true, the default), or whether it requires its playback applications to use authenticationKeys from the service (false) - See [Authenticating Playback Applications](#) for more detail.



Recording using an existing room/archiveID combination will replace the existing recording.

From Client Code



Only a user with OWNER level permissions can start or stop a recording from the client-side. For applications where there is no OWNER user, the server code approach is likely more appropriate.

In order to start recording, the following steps are typical :

1. Specify an `IConnectSession` (`ConnectSessionContainer` or `ConnectSession`) as usual, with an appropriate roomURL.
2. Assign an archiveID to the `IConnectSession`, indicating the name of the recording. If none is specified, `_defaultArchive_` is used.
3. Optionally, assign `IConnectSession.archiveManager.guestsAllowed` to false. We'll cover this in more detail in the next section, [Authenticating Playback Applications](#). If you're just getting started, there's no need to set this property - it defaults to true.
4. When the `IConnectSession` is fully synchronized and you'd like recording to start, set `IConnectSession.archiveManager.isRecording = true;`

That's it! All activity in that `IConnectSession` is now automatically recorded.

To stop recording, simply set `IConnectSession.archiveManager.isRecording = false;`



For sample client code dealing with recording and playback, see `<SDKHome>/sampleApps/Recording/`



`archiveManager.isRecording` will reflect the current state of recording for all clients, no matter where it's initiated/stopped from.

From Server Code

Starting recording from the server integration APIs is also very simple. Each language provided contains an `AccountManager` class, which now contains 2 new APIs :

1. `AccountManager.startRecording("<YOUR-ROOM-NAME>", "<YOUR-ARCHIVE-ID>", guestsAllowed);`
2. `AccountManager.stopRecording("<YOUR-ROOM-NAME>");`




`AccountManager` needs to be instantiated and logged in. For more detail, see the [Provisioning Rooms](#) section of the docs.

Notice the "guestsAllowed" parameter in `startRecording`. As described above, by default, this is set to true, and means that the playback application for this recording will be open - ie, without authentication. If set to false, then the playback application for this recording will require authenticationKeys from the service in order to play back. We'll cover more on authentication for your recordings in the [Authenticating Playback Applications](#) section.

Recording Duration

A recording will last as long as the duration between when it is started (whether from the client or the server), until it's been stopped. There are 2 ways a recording can be stopped :

1. By explicitly stopping it (from the client or server, as shown in the sections above).
2. By allowing the room session to end. Room sessions end 5 minutes after the last user disconnects.


 Using the server APIs, calling startRecording actually starts the room, if no one's currently in it. You've got 5 minutes until someone had better enter the room, or the room session will end and the recording stopped!

Once your recording is stopped, it's automatically zipped and sent to your repository endpoint, as described in the [Overview](#) section.

In the next section, we'll cover [Building a Playback Application](#).


10.4 Building a Playback Application

In order to play back a recording made with the Collaboration Service, you need to build an application which consumes that recording, and plays it back. Fortunately, this is very easy to do, using your existing application, and provides many benefits over creating a "monolithic" video capture of the session.

 If you really just want a screen capture of the session, consider having a client to do screensharing, and recording the session as described. The recorded screenshare video could be used.

Client Application

To build a playback application, let's consider an application which just shows a WebCamera and SimpleChat.


 This is just pseudo-code, meant to get the point across!

```
<ConnectSessionContainer roomURL="<YOUR-ROOM-URL>" archiveID="<YOUR-ARCHIVE-ID>">
  <authenticator>
    <AdobeHSAAuthenticator username="guest" />
  </authenticator>
  <WebCamera/>
  <SimpleChat/>
</ConnectSessionContainer>
```

Here's an application which can play back a recording from this app :

```
<ConnectSessionContainer roomURL="<YOUR-ROOM-URL>" archiveID="<YOUR-ARCHIVE-ID>">
  <authenticator>
    <PlaybackAuthenticator username="guest" />
  </authenticator>
  <WebCamera/>
  <SimpleChat/>
</ConnectSessionContainer>
```

Note the difference : Just take the same application, point it at the same roomURL and archiveID, and change the AdobeHSAAuthenticator to a PlaybackAuthenticator.

 Even if you delete the room in question, as long as you don't delete the recording (either from your [repository endpoint](#) or from the [Developer Portal](#)), the same roomURL+archiveID combination will still work for playback.

Pausing and Seeking

The IConnectSession's archiveManager contains various APIs for manipulating the state of the play back, including being able to pause, un-pause, and seek within the recording. See the ArchiveManager class in the API Reference for more detail, but here's a sample of setting up a simple slider for seek :

```
<s:HSlider width="80%" maximum="{cSession.archiveManager.totalTime}"
value="{cSession.archiveManager.currentTime}" id="slider"
change="{cSession.archiveManager.seek(slider.value)}/>
```



For more sample code relating to playback applications, see [<SDKHome>/sampleApps/Recording](#).

Other Playback Considerations

In general, all your client collaboration components should "just work" for playback if they use SharedModel classes. However, it may be necessary to do a couple of customizations (the client components included in the SDK already implement these techniques) :

1. Disable certain UI elements (for example, the chat entry inputField), which don't make sense during playback.
2. Handle seeking by clearing the shared state of your components as seeking starts.

For 1), it's a simple matter of checking `ISession.archiveManager.isPlayingBack`. This will let you know whether your client component is in a playback state, and allow you to disable interaction if needed.

For 2), seeking works by starting from nothing, then setting the state of your sharedModel to what it should be at the given time. It's considered a best practice to have your component listen for either `reconnect` or `synchronizationChange` events coming from the sharedModel.

During playback, if seeking is occurring, a shared model

1. Briefly goes into an unsynchronized state (`isSynchronized==false`), and also
2. Dispatches a reconnect event.

In the case of chat, for example, this causes the chat UI to clear what it's displaying, so that as the seek finishes, it sets the state of the chat UI to its final state.



Most of this should be transparent to you as a developer, but on occasion you might have to implement a little code to clear state displays yourself. This is identical to handling a reconnect event in the real-time case, so you likely need this code anyhow!

Deep Customization of a Playback Application

While the usage of `archiveManager.isPlayingBack` above shows that your application can handle both real-time and playback use cases, it's worth considering that Playback Applications can really show any amount of the recording that you desire. For example, in the application listed at the beginning of this section, you could create a totally new layout for playback than was used in real-time. You could choose to omit the chat pod, if you didn't need to display it. You could even add interactivity to the playback application.

Because playback takes place in an application, rather than a video player, you're able to completely customize any aspect of how you'd like the playback user experience to look and feel.

In the next section, we'll cover [Authenticating Playback Applications](#).

10.5 Authenticating Playback Applications

By default, as we saw in [Section 10.3](#), all recordings have `guestsAllowed` set to true. This means that the Collaboration Service won't do any special checking as to whether or not a given playback application should be able to play back a recording. For some, this will be perfectly fine, but others may want additional security.



You can still prevent clients from accessing recordings by restricting access to the playback application for those recordings, but this section offers another layer of security you can apply to your recordings.

Using Collaboration Service Authentication

If a recording was started with `guestsAllowed` set to false, then any playback application attempting to play that recording must include a valid `authenticationKey` in its authenticator. If you've built a real-time application with authentication for the Collaboration Services, this will be very familiar - generating an `authenticationKey` follows nearly exactly the same rules shown in [Section 6.4, Authentication Setup, in the "Authenticating on your own systems" heading](#). It's worth reviewing that section of the docs if you plan to add authentication to your playback applications.

In generating an authentication key for a recording, the following steps are required:

1. Instantiate an `accountManager` from your server and log in with your developer credentials (as described in [Section 6.4](#)).
2. When requesting a session for a particular room, instead of just the room name, use `"roomName/archiveID"`. For example, for a room named "myFirstRoom", and a recording made of that room, with `archiveID` "myFirstRecording", call

```
accountManager.getSession("myFirstRoom/myFirstRecording");
```

3. Use the session object to generate an authenticationToken, as per [Section 6.4](#). Note that userID, name, and role are irrelevant in playback applications.
4. Pass that authenticationToken to your playback application, where it will populate the "authenticationKey" parameter of its authenticator with it. See below for a pseudo-code example of the general shape.

```
<ConnectSessionContainer roomURL="<YOUR-ROOM-URL>" archiveID="<YOUR-ARCHIVE-ID>">
  <authenticator>
    <PlaybackAuthenticator authenticationKey="<GENERATED-ON-YOUR-SERVER>" />
  </authenticator>
  <WebCamera/>
  <SimpleChat/>
</ConnectSessionContainer>
```



It's not necessary for the room in question to exist in order to authenticate or play back a recording made from that room. The recording must exist, however, on both your [repository endpoint](#) and in the [Developer Portal](#).

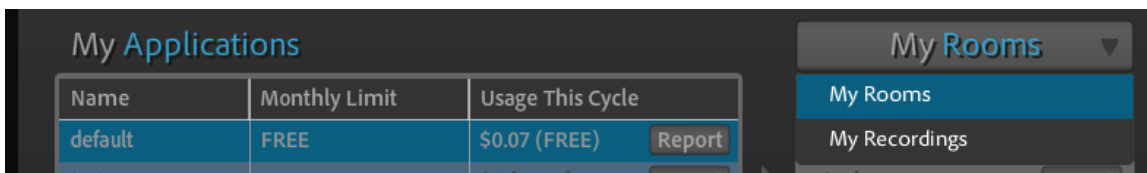
In the next section, we'll see how your recordings can be [viewed and administered from the Developer Portal](#).

10.6 Administration

Your recordings are available for listing and usage reporting via the [Collaboration Service's Developer Portal](#), very much in the same way rooms are handled.

Viewing your Recordings

In the Developer Portal, a new pulldown menu has been enabled in place of the "My Rooms" panel title.

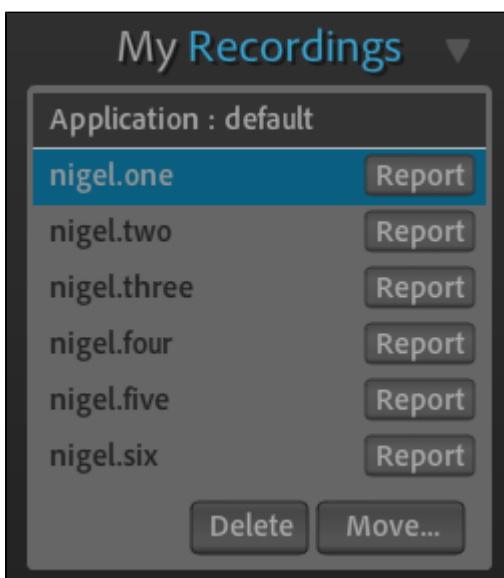


Similar to rooms, recordings are created within the context of an application, defaulting to the application in which the recorded room sat. For example, if you record a room in the "default" application, any recordings will be created within the "default" application.

To view the recordings within an application, select the "My Recordings" option from the pulldown menu, and select the application in question. Notice that all recordings are displayed with the format "<ROOM-OF-ORIGIN>.<ARCHIVE-ID>".



The recording will continue to exist, with the same name, even if its room of origin is deleted.



If you'd like to delete a recording, select one and click the "Delete" button. As the subsequent dialog asserts, this will make your recording unavailable to play back, but won't actually delete the ZIP file in your repository endpoint.

Recordings and Usage Quotas

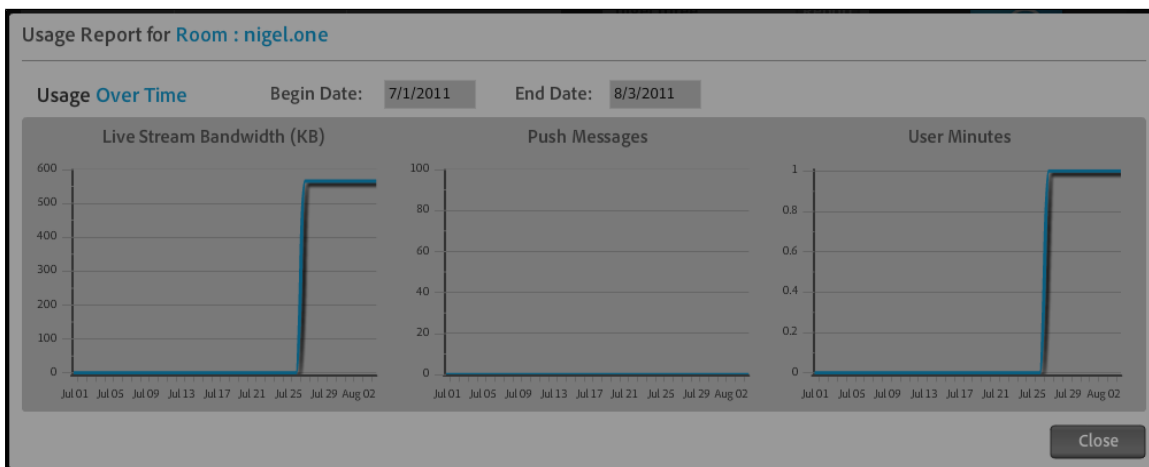
As is suggested by their placement within applications, usage of recordings is guided by the same quota management as is applied to rooms. For applications designated as "FREE", all rooms and recordings therein will have their usage metered, and quotas enforced against the monthly FREE quota. Likewise, for applications where a paid quota is set, usage of recording will be metered, enforced, and billed against those quotas.

i Recordings have their usage measured by GB of bandwidth consumed and user-minutes, but not by messages sent, as opposed to rooms.

If you'd like to move a recording to another application, so that it can be metered and enforced against that application's quotas, select the recording and click the "Move" button. The resultant dialog will allow you to choose the application you'd like to contain the recording.

i Note that any existing usage is still metered against the application it belonged to at the time - moving the recording does not retro-actively affect metering.

To see the amount of usage your recordings are incurring, you can click the "Report" button next to that recording, similar to rooms.



11 Tutorials

The SDK includes working sample applications you can install. The tutorials are based on those samples.

- [Building your first application](#)

11.1 Building your first application

1. Launch the Room Console
2. Create a room on the service
3. Start a new project
4. Connect to LCCS
4. Authenticate to LCCS
5. Make a simple "Hello World" application
6. Monitor your room in the Room Console
7. Develop locally

1. Launch the Room Console

1. Launch the SDK Navigator.
2. Choose the Developer Tools tab.
3. Log in to the Developer Portal. You will need your account URL and your developer credentials. Your account URL appears in the Developer Portal.



2. Create a room on the service

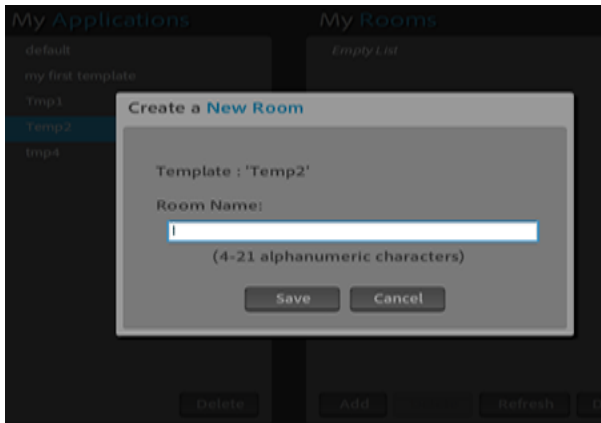
Notice that your portal is relatively. If you're just getting started, it contains only the default (empty) template and no rooms.

A room is LCCS's name for a virtual location on the service which is represented in your applications and to others as an URL. Clients connect to the and send and receive messages (data) to other present clients. Think of rooms as meeting places at some URL. While you can create rooms programmatically or via the Room Console, the Developer Portal provides is a good place to start.

Create a room now.

1. Highlight the Default template.
2. Choose **Add** from the Rooms panel.
3. Provide a simple and intuitive room name that's easy to remember.
4. Choose **Save**.

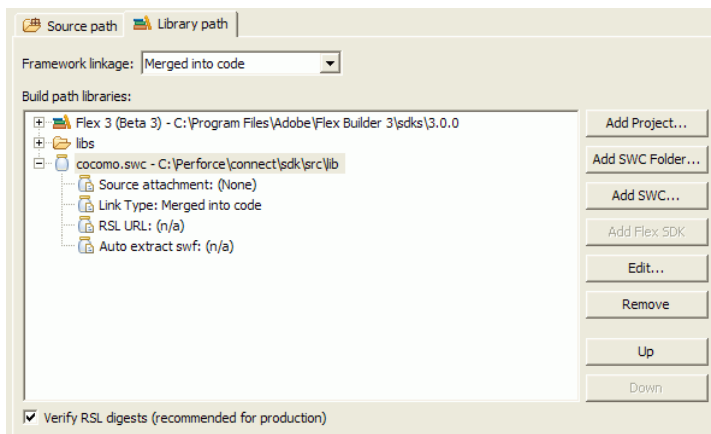
LCCS constructs a room URL you'll use later. It's in the format: <http://collaboration.adobelivecycle.com/<account name>/<room name>>



3. Start a new project

This tutorial uses Flex Builder, but you can also use Flash CS3/4 as your IDE.

1. Start a new Flex project.
2. Name it and select Web Application.
3. When the new project appears in Navigation pane, set the library path to the SWC:
 - a. Choose Project > Properties.
 - b. Choose Flex Build Path.
 - c. Choose the Library path tab.
 - d. Choose Add SWC.



SWC library path

- e. Navigate to <SDK install root>/lib/<player9 | player10>/.



Select the SWC version that supports the version of Flash you'd like to support. The SWC version must match the version of your Flash runtime as well as the version your browser uses. If you are unsure, see [Verifying your Flash versions](#).

- f. Add LCCS.swc to your library path.
 - g. Choose **OK** or go to the next step below to point to LCCS's source code during debugging.
4. Set the source path for debugging: To use LCCS's supplied source code to help with debugging, link it directly into the LCCS SWC (not the Source Path tab):
 - a. Choose the Library path tab, and open the LCCS.swc entry.

- b. Choose **Source Attachment**.
- c. Choose **Edit...**
- d. Navigate to <SDK install root>/src/.
- e. Choose **OK**.

3. Connect to LCCS

The first step is to set up a ConnectSession to the ACFS service and log in. To do so:

1. Open the projects new MXML file.
2. Enter the following code:

ConnectSessionContainer

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  xmlns:rtc="http://ns.adobe.com/rtc">

  <rtc:ConnectSessionContainer
    roomURL="http://collaboration.adobelivecycle.com/AccountName/YourRoomName"
    width="100%" height="100%">
    </rtc:ConnectSessionContainer>

</mx:Application>
```

ConnectSessionContainer is a simple UI container which can connect to LCCS. You can put other components inside it, just like any container, but they won't appear until the session is fully established. There is also a headless version called ConnectSession. It also connects to the service, but isn't a UI container.

4. Authenticate to LCCS

The service needs to know who you are, so use AdobeHSAAuthenticator to authenticate to the service.

1. Add AdobeHSAAuthenticator to the session container:

Authenticating to a session

```
<rtc:authenticator>
  <rtc:AdobeHSAAuthenticator userName="YOURID" password="YOURPASSWORD"/>
</rtc:authenticator>
```

2. Replace the dummy strings with your developer ID user name and password.



During development, you use your developer credentials to log in. However, you DO NOT want users logging in with your credentials because they can then own your account. NEVER publish applications with a hard coded username and password. At deployment time, you'll either allow guest users with no log in, or you'll leverage the authentication capabilities of your own systems.

3. Run the code in debug mode.
The application fires off a lot of output and eventually you'll be connected. At this point, the application has no user interface.

5. Make a simple "Hello World" application

While this application is extremely simple, using the default pods that LCCS provides allows you to add relatively powerful features with very little code.

1. For this example, add three pods after the authenticator but within ConnectSessionContainer:
 - WebCamera pod
 - SharedWhiteBoard
 - SimpleChat

Default pods

```

<mx:HBox width="100%" height="100%">
  <mx:VBox width="25%" height="100%">
    <rtc:WebCamera width="100%" height="50%" />
    <rtc:SimpleChat width="100%" height="50%" />
  </mx:VBox>
  <rtc:SharedWhiteBoard width="75%" height="100%" />
</mx:HBox>

```

2. Run the code.

i You're done! Note that despite the spartan components in the example, you still get a lot of UI because pods essentially provide turn key functionality that may be leveraged with little or no modification.

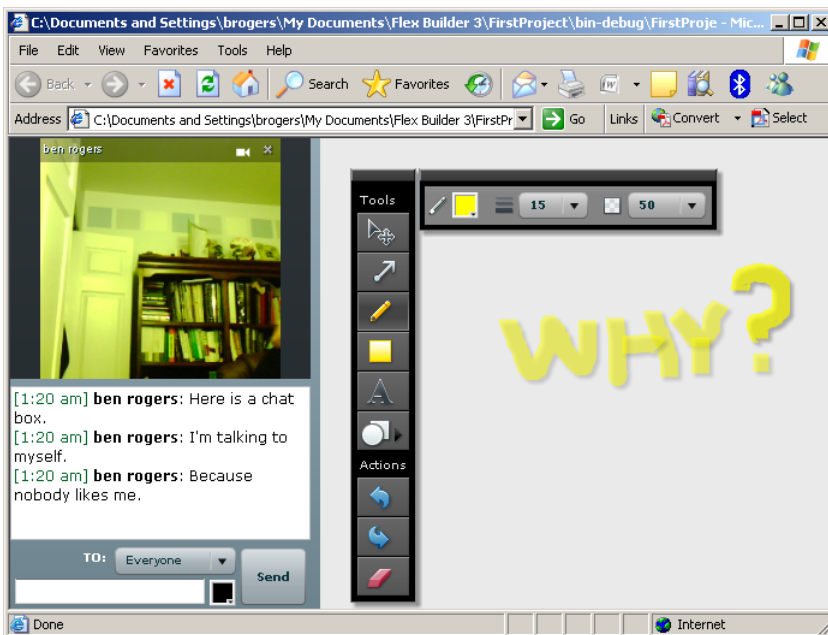
3. Launch another browser at the same URL to see that what you do in one browser affects the other.

Your 15 or 20 lines of code should look like the code below and the fully functional user interface should look like the screenshot below.
Tutorial: Hello World code

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  xmlns:rtc="http://ns.adobe.com/rtc">
  <rtc:ConnectSessionContainer
    roomURL="http://collaboration.adobelivecycle.com/aardvark/YOURROOMNAME"
    width="100%" height="100%">
    <rtc:authenticator>
      <rtc:AdobeHSAAuthenticator userName="YOURID" password="YOURPASSWORD" />
    </rtc:authenticator>
    <mx:HBox width="100%" height="100%">
      <mx:VBox width="25%" height="100%">
        <rtc:WebCamera width="100%" height="50%" />
        <rtc:SimpleChat width="100%" height="50%" />
      </mx:VBox>
      <rtc:SharedWhiteBoard width="75%" height="100%" />
    </mx:HBox>
  </rtc:ConnectSessionContainer>
</mx:Application>

```



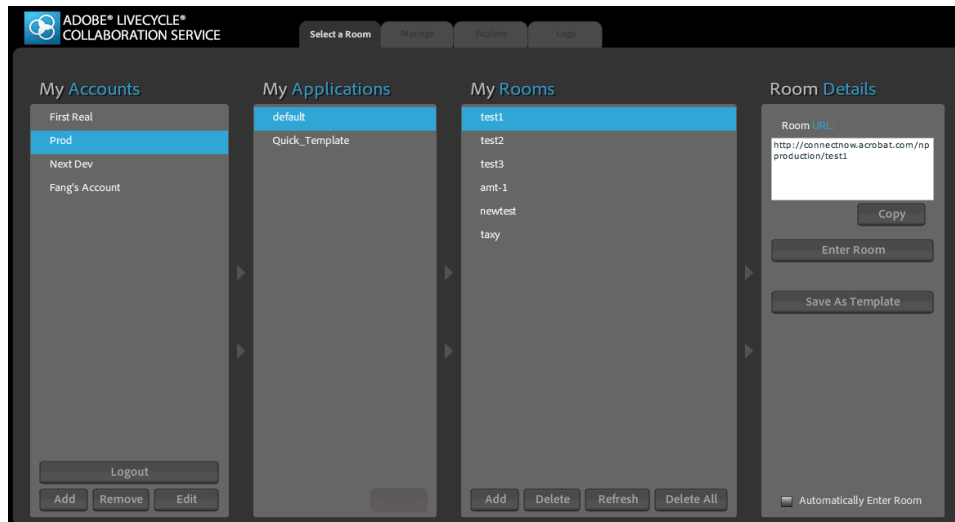
Tutorial: Hello World UI

6. Monitor your room in the Room Console

Now that you have a running application, you can monitor it via the Room Console. To do so:

1. Launch the SDK Navigator if it is not already open.

2. Choose the Developer Tools tab.
 3. Choose **Room Console**.
- The console opens to the Select a Room tab.



Room Console: Select a Room tab

4. If you haven't already added your account, do so now:
 - a. Choose **Add** in the Accounts panel.
 - b. Fill out the Add an LCCS Developer Account dialog.
 - c. Check **Log in to this Account**.
 - d. Choose **Save**.
5. Highlight your account and then the default template.
6. Highlight the room you're using and choose **Enter Room** in the Room Details panel.

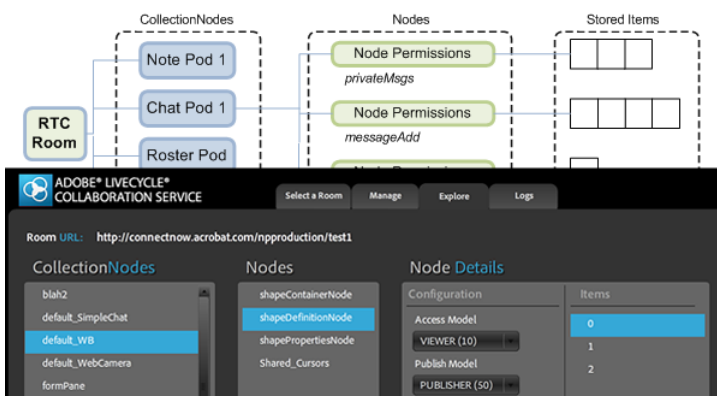
After you've entered the room, the Room Console's other tabs become enabled.

Explore tab

The Explore tab displays your room's collectionNodes; in this case, default_SimpleChat, default_WB, and default_WebCamera. For more detail, see [03 Messaging and Permissions](#).

The shared data is stored in a hierarchical data structure on the service. Branches of this hierarchy are called collectionNodes. You can navigate this hierarchy in the Room Console and see all the messages (called items) that are stored on the service as well as the configuration and permissions of different branches of that data structure.


It's important to note that only a user with OWNER permissions can add new collectionNodes to the service. An OWNER includes yourself and only those others that you explicitly provide this level of permission. This mechanism secures your rooms and assures that they can only be used as you've designed them to be used.



Room Console: Explore tab

7. Develop locally

Another useful item for your developer toolbox is the Local Connection Server. The Local Connection Server enables offline development and testing of non-streaming components without connecting to the service. When running, applications connect automatically, a room URL is not used, and any valid string may be used for the user name and password.

 For more information, see [Local Connection Server](#).

To use the Local Connection server:

1. In your application, change `authentication:AdobeAutheticator` to `authentication:LocalAutheticator`.
2. Start the LCCS Navigator.
3. Choose the Developer Tools tab.
4. Choose **Local Server**.
5. Run your application.

