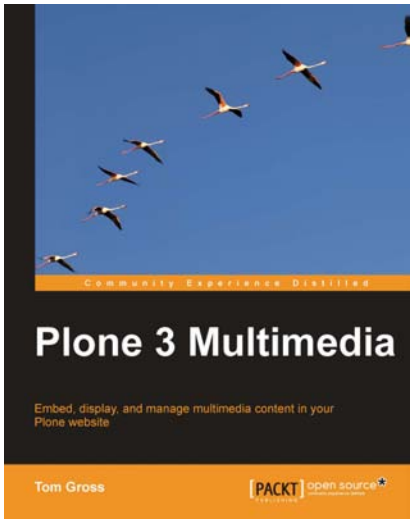
Plone 3 Multimedia

Tom Gross



Chapter No.3
"Managing Audio Content"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Managing Audio Content"

A synopsis of the book's content

Information on where to buy this book

About the Author

Tom Gross is a longtime Zope and Plone user and developer. Since Plone 4.0, he has been a core contributor, and he took responsibility for the rewrite of the reference browser widget. Besides his development and consultant work in Australia, Germany, and Switzerland, he writes technical and philosophical (audio) books.

Tom lives in Zurich and is currently working as a Zope/Plone consultant for the University of Applied Sciences Northwestern Switzerland. Casually, he is doing some other Python, GAE, and repoze.bfg projects.

Tom wrote the German audiobook *Können Maschinen denken? Searles moderne Interpretation des Körper-Geist-Problems*.

I'd like to thank Anne for her support and patience while writing this book. Furthermore, I'd like to thank Tom D. for showing me some really cool Python techniques.

Plone 3 Multimedia

Multimedia is the dominant aspect of the Internet today. There is almost no site with no pictures, videos, Flash animations, or audio content. The integration of multimedia content is the daily mission of web editors and site integrators.

Plone is a mature, stable, and flexible content management system. With the batteries included it provides a complete and user friendly system for managing web content. Completely object-oriented, it is well suited for extensions written in Python.

In this book you will learn to bring these two topics together. It will show you how you can prepare multimedia data for the Web and turn it into valuable content using Plone. With step-by-step examples you will learn how to use Plone and add-ons to provide an appealing multimedia web experience.

What This Book Covers

Chapter 1, *Plone and Multimedia*, tells you what multimedia is all about and what you can expect from Plone. It also shows some reasons why we can, and should, use our favorite Open Source CMS Plone for some additional multimedia candy.

Chapter 2, *Managing Image Content*, shows how we can add images, organize them in folders with thumbnail view, and how to access them. It also discusses two gallery products.

Chapter 3, *Managing Audio Content*, shows how to add audio content to Plone and enhance its features with Plone4Artists products. It also shows how to include audio data in HTML with plugins and Flash.

Chapter 4, *Managing Video Content*, discusses how to add video content to Plone. It also discusses the difference between downloading and streaming, and various products used for enhancing videos in Plone. Chapter 5, *Managing Flash Content*, shows how to include Flash and Silverlight in Plone. It discusses two products that help in improving the inclusion of Flash content in Plone.

Chapter 6, *Content Control*, investigates classic categorization methods. It glances at some products that ease or extend the categorization methods of the default Plone CMS. It also looks at the important techniques of tagging and rating, and few more techniques of content control.

Chapter 7, *Content Syndication*, talks about syndication. It shows how to use RSS syndication with an unmodified Plone installation with collections and searches. It also shows how to enhance syndication with add-on products.

Chapter 8, *Advanced Upload Techniques*, shows how to get content into Plone. It shows various approaches to upload multiple fi les. It also shows how to upload fi les using alternative protocols such as FTP and WebDAV.

Chapter 9, *Advanced Storage*, shows some storage mechanisms in Plone. It also investigates publisher hooks.

Chapter 10, *Serving and Caching*, looks at applications other than Plone such as reverse proxy cache Varnish and Red5, and how to use use CacheFu and also to set cache headers.

Appendix A, *Multimedia Formats and Licenses*, looks at details of formats and codecs used for the storage and transmission of audio and video content. It also looks at the Creative Commons licenses, which can be used to license open (multimedia) content.

Appendix B, *Syndication Formats*, looks at specifications of RSS 2.0, Atom, and the MediaRSS syndication format extension.

Appendix C, *Links and Further Information*, discusses how to use different sources such as the Web, e-mails, and so on to find Plone add-ons and links to selected multimedia topics.

3
Managing Audio Content

Another type of multimedia content besides images is audio. There are at least four use cases when we think of integrating audio in a web application:

1. We want to provide an audio database with static files for download.
2. We have audio that we want to have streamed to the Internet (for example, as a podcast).
3. We want a audio file/live show streamed to the Internet as an Internet radio service.
4. We want some sound to be played when the site is loaded or shown.

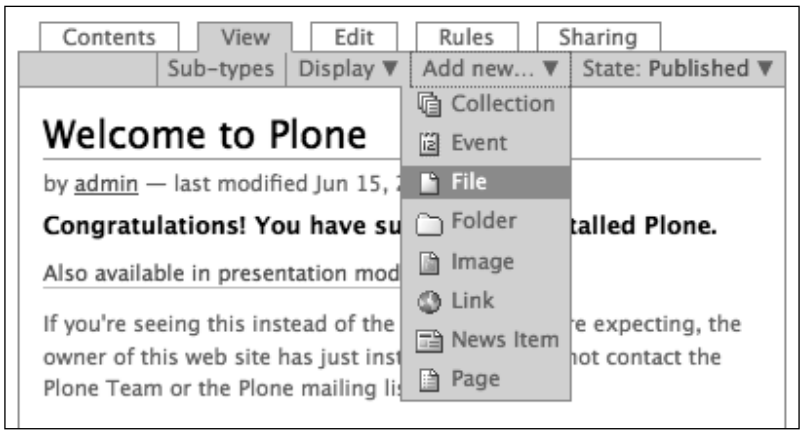
In this chapter we will discuss three of the four cases. The streaming support is limited to use case 2. We can stream to one client like a podcast does, but not to many clients at once like an Internet Radio does. We need special software such as Icecast or SHOUTcast for this purpose. Further, we will investigate how we solve use cases 1, 2, and 3 with the Plone CMS and extensions.

These are the topics covered in this chapter:

- Manipulation of audio content stored as File content in Plone
- The different formats used for the binary storage of audio data
- Storing and accessing MP3 audio metadata with the ID3 tag format
- Managing metadata, formats, and playlists with `p4a.ploneaudio` in Plone
- Including a custom embedded audio player in Plone
- Using the Flowplayer product to include an audio player standalone in rich text and as a portlet
- Previewing the `audio` element of HTML5
- Extracting metadata from a FLAC file using mutagen

Uploading audio files with an unmodified Plone installation

The out of the box support of Plone for audio content is limited. What is possible to do is to upload an audio file utilizing the File content type of Plone to the ZODB. A File is nothing more and nothing less than a simple binary file. Plone does not make any difference between a MP3 file and a ZIP, an EXE, or an RPM binary file.



When adding File content to Plone, we need to upload a file (of course!). We don't necessarily need to specify a title, as the filename is used if the title is omitted. The filename is always taken for the short name (ID) of the object. This limits the number of files with any specific name to one in a container.

While uploading a file, Plone tries to recognize the MIME type and the size of the file. This is the smallest subset of information shared by all binary files the content type **File** was intended for. Normally, detecting the MIME type for standard audio is not a problem if the file extension is correctly set.

Clicking on the link in the default view either downloads the file or opens it with the favorite player of your operating system. This behavior depends on the settings made on the target browser and corresponds with the method 1 of our audio use cases. It goes without saying that we can add the default metadata to files and organize them in folders.

Like Images, File objects do not have a workflow associated in a default Plone setup. They inherit the read and write permissions from the container they are placed into. Still, we can add an existing workflow to this content type or create a new one via the `portal_workflow` tool if we want.

That's pretty much it. Fortunately, we can utilize some extensions to enhance the Plone audio story greatly.

What we will see in this chapter is as follows: First, we will go over some theoretical ground. We will see what formats are available for storing audio content and which is best for which purpose. Later we will investigate the Plone4Artists extension for Plone's File content type—`p4a.ploneaudio`. We will talk about metadata especially used for audio content and how to manipulate it. As a concrete example, we will use **mutagen** to extract metadata from a FLAC file to add FLAC support to `p4a.ploneaudio`. Finally, we will have a word on streaming audio data through the Web and see how to embed a Flash player into our Plone site. We will see how we can do this programmatically and also with the help of a third-party product called `collective.flowplayer`. At the very end of the chapter, we have a small technical preview on HTML5 where a dedicated **audio element** is available. This element allows us to embed audio directly into our HTML page without the detour with Flash.

Accessing audio content in Plone

Once we upload a file we want to work with to Plone, we will link it with other content and display it in one way or another. There are several ways of accessing audio data in Plone. It can be accessed in the visual editor by editors, in templates by integrators and in Python code by developers.

Kupu access

Unlike for images, there is no special option in the visual editor to embed file/audio content into a page. The only way to access an audio file with Kupu is to use an internal link. The file displays as a normal link and is executed when clicked. Executed means (as for the standalone file) saved or opened with the music player of your operating system as is done in the standard view of the File content type.

Of course, it is possible to reference external audio files as well.

Page template access

As there is no special access method in Kupu, there is none in page templates. If we need to access a file there, we can use the `absolute_url` method of the audio content object. This computes a link we can refer to. So the only way to access a file from another context is to refer to its URL.

```
<a tal:attributes="href audiocontext/absolute_url"
   tal:content="audiocontext/Title">audio</a>
```

Python script access

If we need to access the content of an (audio) file in a Python script, we can get the binary data with the Archetype accessor `getFile`.

```
>>> binary = context.getFile()
```

This method returns the data wrapped into a Zope `OFS.File` object. To access the raw data as a string, we need to do the following:

```
>>> rawdata = str(binary.data)
```

Accessing the raw data of an audio file might be useful if we want to do format transformations on the fly or other direct manipulation of the data.

Field access

If we write our own content type and want to save audio data with an object, we need a `file` field. This field stores the binary data and takes care of communicating with the browser with adequate view and edit widgets. The `file` field is defined in the `Field` module of the Archetype product. Additional to the properties, it exclusively defines that it inherits from the `ObjectField` base class. The following properties are important.

Key	Default value
`type`	`'file'`
`default`	`''`
`primary`	`False`
`widget`	`FileWidget`
`content_class`	`File`
`default_content_type`	`'application/octet-stream'`

The `type` property provides a unique name for the field. We usually don't need to change this. The `default` property defines the default value for this field. It is normally empty. If we want to change it, we need to specify an instance of the `content_class` property.

One field of the schema can be marked as `primary`. This field can be retrieved by the `getPrimaryField` accessor. When accessing the content object with FTP, the content of the primary field is transmitted to the client.

Like every other field, the `file` field needs a widget. The standard `FileWidget` is defined in the `Widget` module of the Archetypes product.

The `content_class` property declares the instance, where the actual binary data is stored. As standard, the `File` class from Zope's `OFS.Image` module is used. This class supports chunk-wise transmission of the data with the publisher.

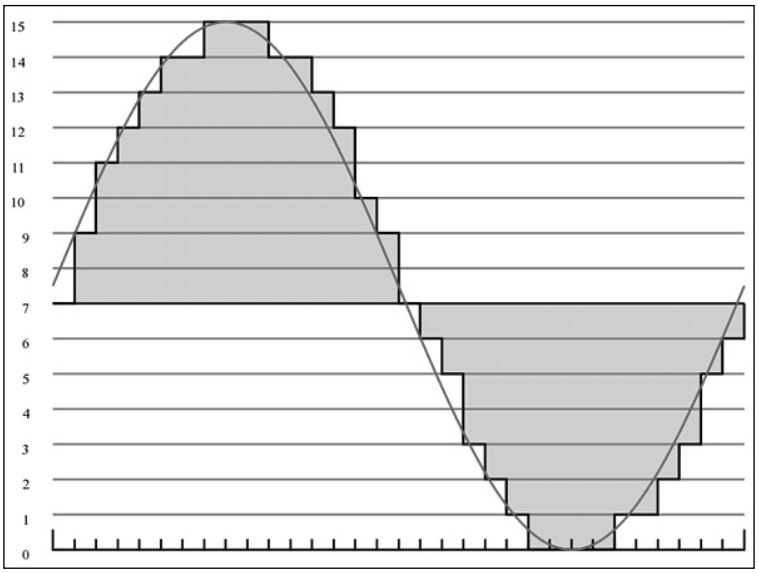
Field can be accessed like any other field by its `accessor` method. This method is either defined as a property of the field or constructed from its name. If the name were "audio", the accessor would be `getAudio`. The accessor is generated from the "get" prefix with the capitalized name of the field.

Audio formats

Before we go on with Plone and see how we can enhance the story of audio processing and manipulate audio data, we will glance at audio formats. We will see how raw audio data is compressed to enable effective audio storage and streaming. We need to have some basic audio know-how about some of the terminology to understand how we can effectively process audio for our own purposes.

As with images, there are several formats in which audio content can be stored. We want to learn a bit of theoretical background. This eases the decision of choosing the right format for our use case.

An analog acoustic signal can be displayed as a wave:



If digitalized, the wave gets approximated by small rectangles below the curve. The more rectangles are used the better is the sound (fidelity) of the digital variant. The width of the rectangles is called the **sampling rate**.

Usual sampling rates include:

- 44.1 kHz (44,100 samples per second): CD quality
- 32 kHz: Speech
- 14.5 kHz: FM radio bandwidth
- 10 kHz: AM radio
- 8 kHz: Telephone speech

Each sample is stored with a fixed number of bits. This value is called the audio **bit depth** or **bit resolution**.

Finally, there is a third value that we already know from the analog side. It is the **channel**. We have one channel for **mono** and two channels for **stereo**. For the digital variant, this means a doubling of data if stereo is used.

So let's do a calculation. Let's assume we have an audio podcast with a length of eight minutes, which we want to stream in stereo CD quality. The sampling rate corresponds with the highest frequency of sound that is stored. For accurate reproduction of the original sound, the sample rate has to be at least twice that of the highest frequency sound. Most humans cannot hear frequencies higher than 20 kHz. The corresponding sampling rate to 20 kHz is a sampling rate of 44100 samples. We want to use a bit resolution of 16. This is the standard bit depth for audio CDs. Lastly, we have two channels for stereo:

44100 x 16 x 2 x 60 x 8= 677376000 bits = 84672000 bytes ≈ 80.7 MB

This is quite a lot of data for eight minutes of CD-quality sound. We do not want to store so much data and more importantly, we do not want to send so much data over the Internet. So what we do is compress the data. Zipping the data would not give us a big effect because of the binary structure of digital audio data. There are different types of compressions for different types of data. ZIPs are good for text, JPEG is good for images, and MP3 is good for music—but why? Each of these algorithms takes the nature of the data into account. ZIP looks for redundant characters, JPEG unifies similar color areas, and MP3 strips the frequencies humans do not hear from the raw data.

Audio compression algorithms are called **codecs**. There are two kinds of these codecs—**lossless codecs** and **lossy codecs**. We don't want to go into further details here. All we need to know is that lossless codecs don't lose data (quality) when they compress. Lossy codecs compress better but loose data. Thus if we convert a raw stream to a lossy format (such as MP3 or Ogg Vorbis), converting it back to raw, and back again to MP3, the output will differ from the first one. Usually, one won't hear the difference between a raw file and a lossy-encoded one after a single encoding pass, but there is some recognizable quality loss after multiple passes. If we do the same with a lossless codec, the output stays the same no matter how often we encode and decode.

Some commonly used audio codecs are:

- Lossy: MP3, Ogg Vorbis, Musepack, WMA, and AAC
- Lossless: FLAC, WavPack, Monkey's Audio, and ALAC/Apple Lossless
- Raw: WAV and AIFF

Choosing the right audio format

You may ask the question: There are so many formats, which one shall I use? If you have a choice, which may not always be the case, you can rely on some short guidelines:

Format	Decision guidelines
MP3	You want to reach as many people as possible.
	You want your audio content to be playable with almost all mobile players.
	Your audio may be used together with Flash; MP3 is easily embedded there.
	You want a format that is easily streamable.
	You want small file sizes for storing and streaming.
Ogg Vorbis	You want most of the advantages of MP3.
	You want a patent-free format (this can be helpful if you plan to use HTML5).
FLAC	You have high-quality audio content.
	You have big disk space.
	You and your users don't care about Internet bandwidth.
Other formats	You have a special reason to do so (for example, if your users stick to iTunes, you may probably use AAC).

Converting audio formats

Sometimes we need to convert one audio format to another. Most web audio players understand only a few formats. Often, they are limited to MP3 only. If we want to play our audio—available in the Ogg Vorbis format—with such players, we have to convert it first. We will see how to do that in this section. If you work a lot with multimedia, you probably know the **VLC player** from VideoLAN. VLC is a media player and server. It is available on most platforms, including Windows, Mac OS X, and Linux. If VLC doesn't support the format you need, check the home page of the audio format.

Many audio players support the encoding of audio too. On Windows, the popular audio player Winamp can be used to convert audio formats. On Linux, you probably want to try Amarok. Amarok is a player for KDE and its plugins are scriptable with Python. There are ready available plugins for converting audio data.

Sometimes it is not possible to convert directly. This makes it necessary to convert to raw audio (WAV) first, and then convert it to the desired target format.

Converting audio with VLC

If we use the VLC player for converting audio files, we are utilizing the streaming mode of the player. We open the **Streaming/Export Wizard...** from the **File** menu. There we choose the second option **Transcode/Save to file** in the dialog box. Next, we select a file available in the playlist of the player or from the hard disk. After doing so, we select the target format. As stated before most players support MP3, so **MP3** might be good choice. If the raw format is needed, we have to choose **Uncompressed, integer**. On the next screen, we have to pick the encapsulation format. If we have selected MP3 before, **RAW** is a good choice here because we are able to read and manipulate the created file with most audio editing software (for example. Audacity). If we have selected **Uncompressed, integer** in the earlier step, we don't have a choice now as **WAV** is the only supported encapsulation format in this case. As the last step we choose a filename for the file, which is created with the new format.

After confirming the summary, we are ready and have a new item in our VLC playlist: **Streaming Transcoding Wizard (1/1)**. We need to "play" this item to make the actual transcoding happen. The process might take some time depending on the source and target format. We don't hear anything during the transcoding process. In the case of success, there is a new file on our hard disk that we can test with VLC and then use with our favorite web player in Plone.

Audio metadata

As for most digital photo formats, it is also possible for most audio formats to store some additional data on the binary file. This data contains information on the artist, the album, the genre, the encoding itself, and some more information.

ID3 tag: The metadata format for MP3

The ID3 tag is the metadata format for MP3. **ID3** stands for **Identify an MP3**. Before it was introduced, the only chance of storing metadata was in the filename, which tended to get very long. The ID3v1tag is capable of storing this information:

Offset	Length	Description
0	3	"TAG" Identifier
3	30	Song title
33	30	Artist
63	30	Album
93	4	A four-digit year
97	30	Comment
127	1	Genre

There have been some revisions in the format. Nowadays, ID3v2 tags are commonly used. The ID3v2 tag is a complete rewrite of the original ID3 tag implementation. The format is capable of storing icons of the cover art, supports character encoding, and the stored information is not limited to a few characters. The maximum for storing metadata on MP3 with the ID3v2 tag is 256 megabytes. This is enough space for storing karaoke lyrics in several languages.

Metadata of other audio formats

Most other audio formats support storing metadata information on the file as well. The Ogg Vorbis metadata is called Vorbis comments. They support metadata tags similar to those implemented in the ID3 tag standard for MP3. Music tags are typically implemented as strings of the `[TAG]=[VALUE]` form (for example, "ARTIST=The Rolling Stones").

Like the current version of the ID3 tag, users and encoding software are free to use whichever tags are appropriate for the content.

FLAC defines several types of metadata blocks. One of these blocks is favored. It is the only mandatory STREAMINFO block. This block stores audio-centric information such as the sample rate, the number of channels, and so on. Also included in the STREAMINFO block is the MD5 signature of the unencoded audio data. This is useful for checking an entire stream for transmission errors.

Metadata blocks can be any length and new ones can be defined. A decoder is allowed to skip any metadata types it does not understand.

Editing audio metadata

Let's see how the metadata comes into the audio. Most CD encoding programs query an open metadata database such as freedb to generate the metadata for our audio content automatically. If we have MP3 files that are not encoded on their own, we need a tag editor. Almost every modern player supports accessing the ID3 tag information nowadays. If you have a Mac, you can use iTunes to manipulate the ID3 tag information of every track. Use *command* for accessing the metadata window.

On Windows and Linux there is a product called EasyTAG (`http://easytag.sourceforge.net/`), which allows you to manage the metadata of your audio files for whole directories. You can use this software on the Mac too, if you have MacPorts.

EasyTAG also supports writing the Ogg Vorbis and FLAC metadata.

There are other options. Check the manuals of your favorite audio player. Very likely, it comes with some support of reading and writing metadata.

Now we are perfectly prepared to manage our content with Plone: We chose a compression format for our data. We structured the data with additional metadata. What we want is to take this effort into Plone. A simple File content type is not sufficient any longer. We will investigate an extension in the next section, which aims to solve this issue.

Audio enhancements with p4a.ploneaudio

One of the most advanced products to boost the audio features of Plone is `p4a.ploneaudio`. Like its image sister `p4a.ploneimage` it doesn't bring a content type on its own, but expands existing ones. As you might have guessed already, the File content type and the folderish ones (Folder, Large Folder, and Collection) are chosen for the enhancement.

To install it, add the following code to the buildout of our instance:

```
[buildout]
...

[instance]
...
eggs =
    ${buildout:eggs}
    Plone
    p4a.ploneaudio
zcml =
    p4a.ploneaudio
```

After running the buildout and restarting the instance, we need to install the product as an add-on product. We find it with the product name **Plone4Artists Audio (p4a.ploneaudio)**.

Enhancing files

Installing the `p4a.ploneaudio` product enables the enhancement of the File content type of `ATContentTypes`. Unlike with the image enhancement, not all files are automatically enhanced with additional audio features. `p4a.ploneaudio` comes with a MIME type filter. If we upload a file with one of the meta types such as **audio/MPEG** or **application/Ogg**, the file will automatically be audio enhanced. All other files (such as ZIP files, EXE files, and so on) stay untouched.

Technically speaking, this works via a named adapter.

The first thing we see is the modified default view for audio-enhanced files:

On the right side we see some technical information about the audio file itself. We find the size, the format (MP3 or Ogg), the bitrate, the frequency, and the length of the track there. As with any other content type, we have the title and the description at the top of the content area. For the audio-enhanced content, the description has changed from a simple text field to a rich text field. We find four buttons after the usual CMS bar (the `belowcontenttitle` slot) containing the author and the modification date of the file. Each of these buttons retrieves the audio file in a different way:

Button	Action	Description
	Play	An embedded audio player written in Flash. Clicking on it starts playback immediately.
	Pop up	Opens a pop-up window with a Flash player playing the audio track. This is useful if someone wants to play the track while continuing to browse the site.
	Stream	Clicking on the button returns an M3U playlist with the URL of the file. The browser/ operating system needs to take care of the handling of the stream. If you have a Mac, the file will be streamed to iTunes.
	Download	This is the standard behavior of Plone's File content type. The file is returned to the client like any other file object. It can be saved or opened with the favorite media player of the operating system.

We find a long list of additional information on the audio track below the player and streaming buttons. This information is the metadata stored with the audio file and is gathered during the enhancing process. It can be changed and written back to the file.

Let's take a closer look on the enhancement process.

One important step here is marking the content object with two interfaces:

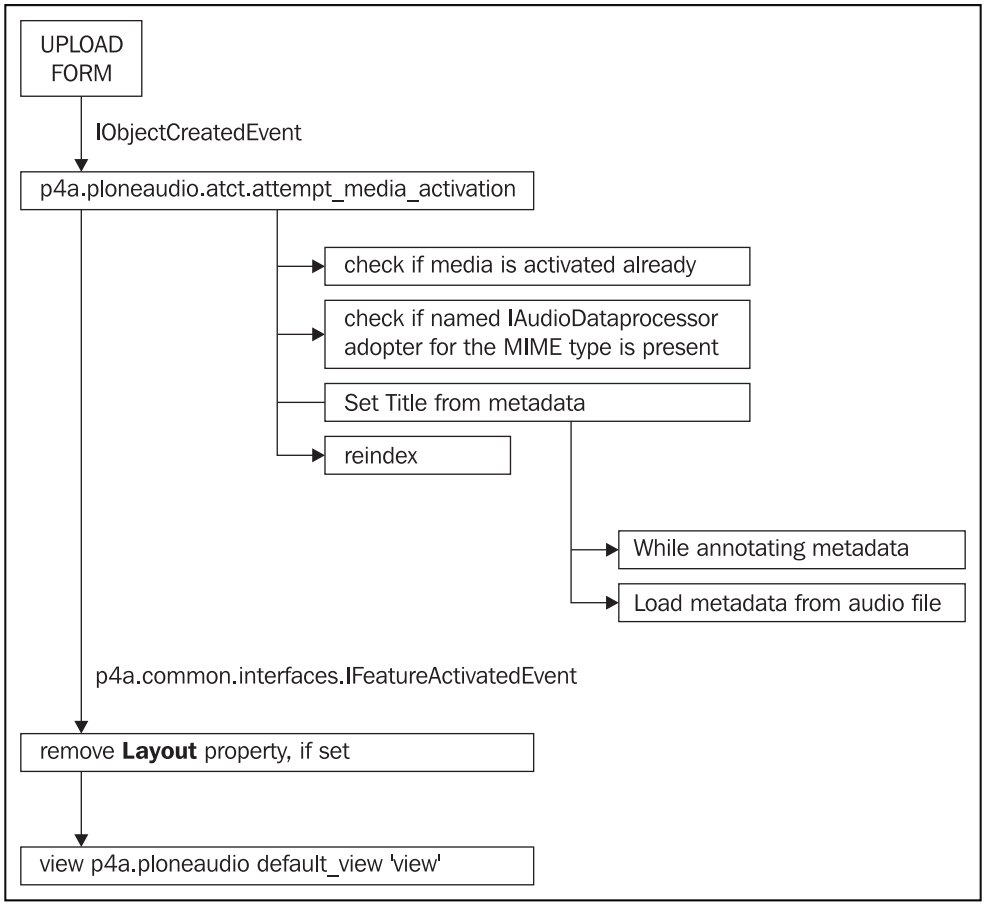
- One is `p4a.subtyper.interfaces.ISubtyped`. This interface marks the content as subtyped.
- The other interface is `p4a.audio.interfaces.IAudioEnhanced`. This interface describes the type of enhancement, which is audio content in this case.

All other views and components available with the enhanced version bind to the `p4a.audio.interfaces.IAudioEnhanced` interface.

Additionally, `p4a.ploneaudio` tries to extract as much of the metadata information from the file as possible.

The title is retrieved from the metadata and changed.

Let's see what happens when a file is added in detail:



It is possible to write custom audio enhancers if needed. The enhancers are queried as named adapters by the MIME type on the `IAudioDataAccessor` interface.

Enhancing containers

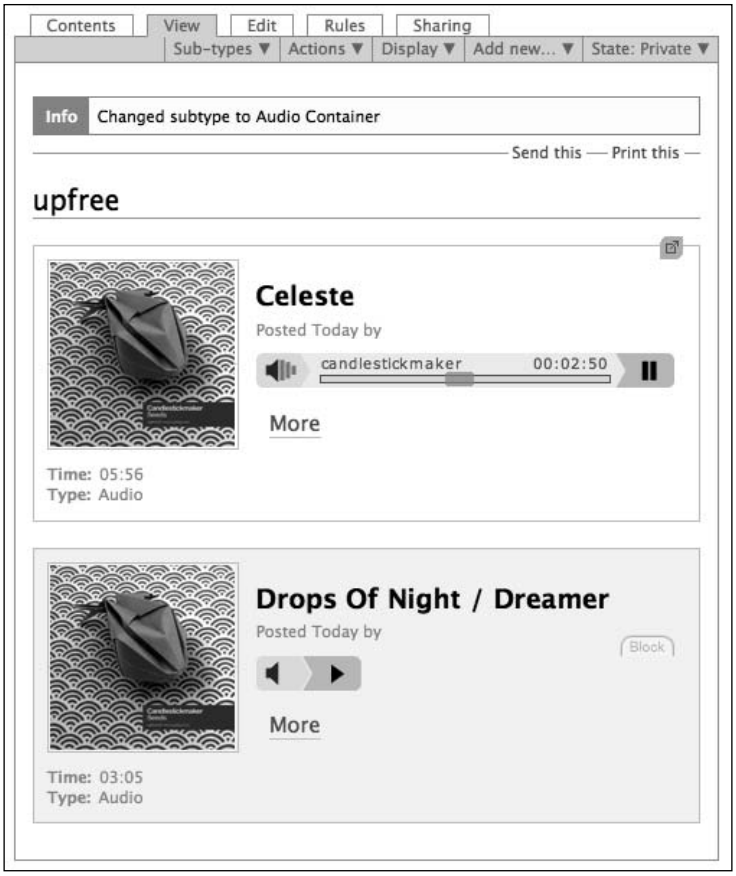
With `p4a.ploneaudio`, we can turn any Plone folder into an audio-enhanced folder by choosing **Audio container** from the **Subtype** menu.

Two marker interfaces are attached to the container:

`p4a.subtyper.interfaces.ISubtyped`

`p4a.audio.interfaces.IAudioContainerEnhanced`

The default view of the container is changed utilizing the `IAudioContainerEnhanced` interface. The new default view is `audio-container.html`, which comes with the `p4a.ploneaudio` product.



In the **View** option we see one box for each track. For each track, the title and the cover artwork is shown. There is also an embedded Flash player to play each track directly.

There are some other container views that come with the product as follows:

- `audio-album.html` (album view)
- `audiocd-popup.html` (pop-up view)
- `cd_playlist.xspf`

And the default edit view for `IAudioContainerEnhaced`-marked folders is overridden.

The `audio-album.html` view presents the audio content of the container in a slightly different way. The single tracks are arranged in a table. Two variants of this view are exposed in the **Display** menu of the container—Album view and Album view with no track numbers. Both these views are more compact than the default audio container view and have the same layout, except that the second one omits the column with the track numbers. As for the other audio container views, there is a player for each track. Moreover, there are two buttons on the top of the page. One is the commonly used RSS icon  and the other one is the familiar pop-up player button  .

By clicking on the RSS icon, we get to the stream (or podcast of the folder). This may not be too interesting for a standard static Folder, but can be for a Collection where the content changes more dynamically.

The pop-up player for a folder looks and operates slightly differently as the file standalone variant. It contains all tracks that are part of the audio container as a playlist.

Lastly, there is the `cd_playlist.xspf` view. This view is not exposed via a display or as an action. It provides the audio data as a XSPF stream.

The XML Shareable Playlist Format: XSPF

The **XML Shareable Playlist Format** (**XSPF**) is an XML file format for storing playlists of digital audio. The audio can be located locally or online. Last.fm uses XSPF for providing its playlists. An example playlist looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<playlist version="1" xmlns="http://xspf.org/ns/0/">
  <trackList>
    <track>
      <location>example.mp3</location>
```

```
        <creator>Tom</creator>
        <album>Thriller</album>
        <annotation>A comment</annotation>
      </track>
      <track>
        <location>another.mp3</location>
        <creator>Tom</creator>
      </track>
    </trackList>
  </playlist>
```

There is a list of applications supporting the XSPF format on the XSPF (`http://xspf.org/`) website.

The default XSPF view of `p4a.ploneaudio` does not include all possible information. It provides the following attributes:

- `location`: This is the required location (URL) of the song.
- `image`: The album artwork that may be included with each track.
- `annotation`: `p4a.ploneaudio` collects information about the title, artist, album, and year in this field.

It is easy to write a custom implementation of an XSPF view. Look at the available fields at the XSPF home page and the XSPF implementation of `p4a.audio`. We find the template named `cd_playlist.xspf.pt` in the `p4a.audio.browser` module.

p4a.ploneaudio and the Plone catalog

Besides the customized default views and the additional views, `p4a.ploneaudio` comes with a number of other changes. One very important change is the disclosure of some audio metadata fields to the portal catalog. This allows us to use them in catalog queries in general and smart folders in particular.

The following indexes are added:

- Artist
- Genre
- Track
- Format

The "Artist" attribute is exposed as metadata information in the catalog. Also, the genre and the artist name are added to the full text index "SearchableText".

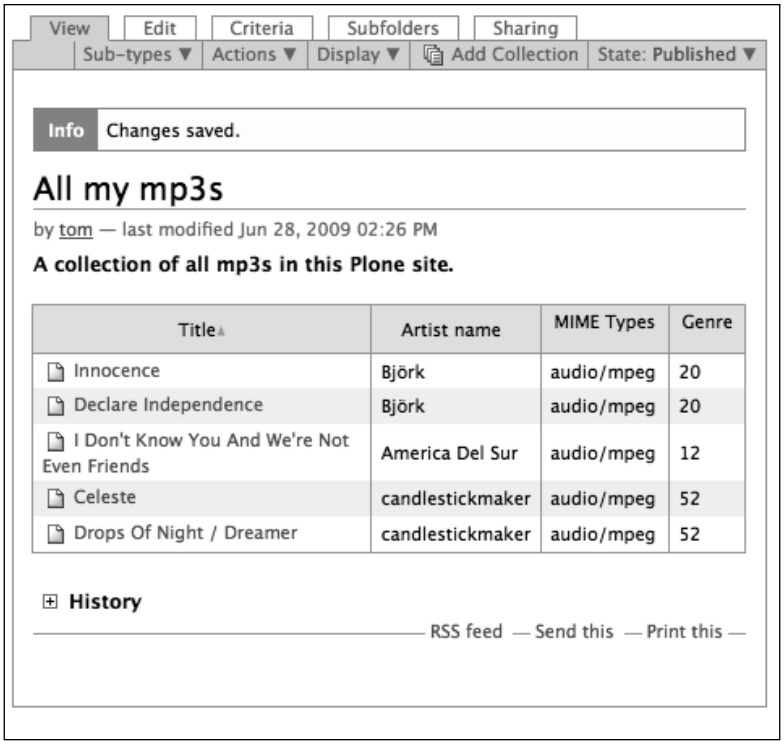
The values for the genre field are hardcoded. The ID3 tag genre list is used together with the Winamp extensions. They are stored as a vocabulary. The term titles are resolved for Searchable Text, but not for the index and the Collection field.

Accessing audio metadata in Collections

To access catalog information in collections, it needs to be exposed to the collections tool. This can either be done by a product or TTW in the Plone configuration panel. `p4a.ploneaudio` comes with a modification of the fields:

- Artist (**Artist name)**
- Genre (**Genre**)
- Format (**MIME Types**)

Let's say we want a collection of all our MP3 files. All we have to do is add a **MIME Types** criterion to our collection and set **audio/mpeg** as the value:



ATAudio migration

If you have an older Plone site (2.5), you probably have `ATAudio` installed to deal with audio content. `ATAudio` has features similar to `p4a.ploneaudio`. This is not surprising as `p4a.ploneaudio` was derived from `ATAudio`. The main difference is that `ATAudio` provides a content type, while `p4a.ploneaudio` reuses an existing one. If you want to switch from `ATAudio` to `p4a.ploneaudio` for one or the other reason, `p4a.ploneaudio` comes with a migration routine. One reason for switching might be that `ATAudio` is not actively developed any more and probably doesn't work with recent versions of Plone. For migrating, you need to call `migrate-ataudio-configlet.html` and follow the instructions there.

The migration view is available only if `ATAudio` is installed. There is a little bit of a catch-22 situation because `p4a.ploneaudio` doesn't run on Plone 2.5 and `ATAudio` doesn't run on Plone 3. This means there is no good starting point for the migration. At least, there is a version of `ATAudio` that does install in Plone 3 in the collective repository:

```
http://svn.plone.org/svn/collective/ATAudio/tags/0.7-
plone3migration/.
```

Extracting metadata with AudioDataAccessors

The `IAudioDataAccessor` interface is used for extracting metadata from binary audio content. If uploading a file, Plone tries to acquire a named adapter for the interface with the MIME type as the key. It has the following layout:

```
class IAudioDataAccessor(interface.Interface):
    """Audio implementation accessor (ie MP3, ogg, etc).
    """
    audio_type = schema.TextLine(title=_(u'Audio Type'),
                                 required=True,
                                 readonly=True)
    def load(filename):
        """Load from filename"""

    def store(filename):
        """Store to filename"""
```

The `audio_type` field contains a human-readable description of the audio type. In the case of MP3, the Unicode string "MPEG-1 Audio Layer 3" is used. The `load` and `store` methods are used for reading/writing the metadata from/to the audio file.

The definition for the MP3 adapter looks like this:

```
<adapter
    for="p4a.audio.interfaces.IPossibleAudio"
    factory="._audiodata.MP3AudioDataAccessor"
    provides="p4a.audio.interfaces.IAudioDataAccessor"
    name="audio/mpeg"
    />
```

The component is registered for the `p4a.audio.interfaces.IPossibleAudio` interface. All classes marked with this interface are capable of getting converted to an audio-enhanced object. The standard product marks the File content type with this interface. The adapter provides the `p4a.audio.interfaces.IAudioDataAccessor` interface, which is the marker for the lookup. The `name` attribute of `adapter` is the key for the lookup and needs to be set to the MIME type of the audio file that should be processed.

Now, the `factory` does the actual work of loading and storing the metadata from the audio file.

p4a.ploneaudio and FLAC

For the moment, `p4a.ploneaudio` only supports MP3 and Ogg Vorbis. This is a reasonable choice. Both formats are streamable and were made for good audio quality with small file sizes. We want to add FLAC support. We use mutagen for metadata extraction. Mutagen is an audio metadata extractor written in Python. It is capable of reading and writing many audio metadata formats including:

- FLAC
- M4A
- Monkey's Audio
- MP3
- Musepack
- Ogg Vorbis
- True Audio
- WavPack
- OptimFROG

(For a description of the individual formats, see *Appendix A*.)

Let's remember the flow chart of the audio adding process. We recall that we need a metadata extractor for our FLAC MIME type.

First, we need to register a named adapter for this purpose. This is very similar to the MP3 adapter we saw before:

```
<adapter
      for="p4a.audio.interfaces.IPossibleAudio"
      factory=".flac._audiodata.FlacAudioDataAccessor"
      provides="p4a.audio.interfaces.IAudioDataAccessor"
      name="application/x-flac"
      />
```

The adapter is used for objects implementing the `p4a.audio.interfaces.IPossibleAudio` interface. The factory does the work of extracting the metadata. The adapter provides the `p4a.audio.interfaces.IAudioDataAccessor` interface. This is what the adapter is made for and the `name` is `application/x-flac`, which is the MIME type of FLAC audio files.

Next, we define the metadata accessor:

```
from mutagen.flac import Open as openaudio
...
from p4a.audio.ogg._audiodata import _safe
```

First, we need some third-party imports. For the metadata extraction, we use the FLAC accessor of the mutagen library. `_safe` is a helper method. It returns the first element if the given parameter is a list or a tuple, or the element itself.

```
class FlacAudioDataAccessor(object):
    """An AudioDataAccessor for FLAC"""

    implements(IAudioDataAccessor)

    def __init__(self, context):
        self._filecontent = context
```

The first lines are the boilerplate part. The adapter class implements the interface the adapter provides. In the constructor, we get the context of the adapter and save it in the `_filecontent` variable of the instance.

```
    @property
    def audio_type(self):
        return 'FLAC'
    @property
    def _audio(self):
        return IAudio(self._filecontent)
    @property
    def _audio_data(self):
        annotations = IAnnotations(self._filecontent)
        return annotations.get(self._audio.ANNO_KEY, None)
```

The `audio_type` property is just for information purposes and required by the interface. It is displayed in the audio view. The `_audio` property is a shortcut for accessing the `IAudio` adapter for the context. The `audio_data` property is a shortcut for accessing the metadata annotated to the context.

```
    def load(self, filename):
        flacfile = openaudio(filename)

        self._audio_data['title'] = _safe(flacfile.get('title', ''))
        self._audio_data['artist'] = _safe(flacfile.get('artist', ''))
        self._audio_data['album'] = _safe(flacfile.get('album', ''))
        self._audio_data['year'] = _safe(flacfile.get('date', ''))
        self._audio_data['idtrack'] = _safe(flacfile.
                                            get('tracknumber', ''))
        self._audio_data['genre'] = _safe(flacfile.get('genre', ''))
        self._audio_data['comment'] = _safe(flacfile.
                                            get('description', ''))

        self._audio_data['bit_rate'] = long(flacfile.info.
                                            bits_per_sample)
        self._audio_data['length'] = long(flacfile.info.length)
        self._audio_data['frequency'] = long(flacfile.info.
                                             sample_rate)
```

The `load` method is required by the `IAudioDataAccessor` interface. It fetches the metadata using the mutagen method from the audio file and stores it as an annotation on the context.

```
def store(self, filename):
        flacfile = openaudio(filename)

        flacfile['title'] = self._audio.title or u''
        flacfile['artist'] = self._audio.artist or u''
        flacfile['album'] = self._audio.album or u''
        flacfile['date'] = self._audio.year or u''
        flacfile['tracknumber'] = self._audio.idtrack or u''

        flacfile.save()
```

The `store` method is required by the `IAudioDataAccessor` interface as well and its purpose is to write the metadata from the context annotation back to the audio file.

Including audio into HTML

Generally, we have two options to include a sound file on a HTML page:

- Streaming
- Non streaming

Another option is to simply include a link to the file like this:

```
<a href="example.mp3" type="audio/x-mpeg" title="MP3 audiofile , ...
kB">example.mp3 </a>
```

This is the standard way Plone includes files into the visual editor.

The advantage of this approach is that virtually everyone is able to access the file in some way. What happens with the file after it has been downloaded depends on the client browser and how it is configured. The shortcoming of this method is that we depend on an external player to listen to the audio. Probably one needs to download the file and start the player manually.

Including audio with plugin elements

Another option to spread an audio file is to use the `embed` element or the `object` element. The former looks like this:

```
<embed src="example.mp3">
```

The `embed` element was introduced by Netscape in browser version 2.0. However, it has not yet made it into the HTML standard, and probably will never do so.
An alternative element to the Netscape variant is the `object` element that was introduced by Microsoft. Including an `example.mp3` file located in the `data` folder looks like this:

```
<object type="audio/x-mpeg" data="data/example.mp3" width="200"
        height="20">
  <param name="src" value="data/example.mp3">
  <param name="autoplay" value="false">
  <param name="autoStart" value="0">
  alt : <a href="data/example.mp3">example.mp3</a>
</object>
```

Including audio with the `embed` or the `object` element assumes that there is a plugin installed on the client side that can play the multimedia format. In most cases, we can't tell what the client is equipped with and want a more robust solution.

The third way to include audio into your site is Flash. We still need a plugin on the client side, but Flash is more widespread than audio player plugins. There are a couple of free audio players written in Flash. An older but easy-to-use Flash player is EMFF.

A custom view with an embedded audio player

What we do now is to write a custom view for the audio-enhanced File content type of Plone. We reuse the `mm.enhance` product we created in the previous chapter and add the additional code there.

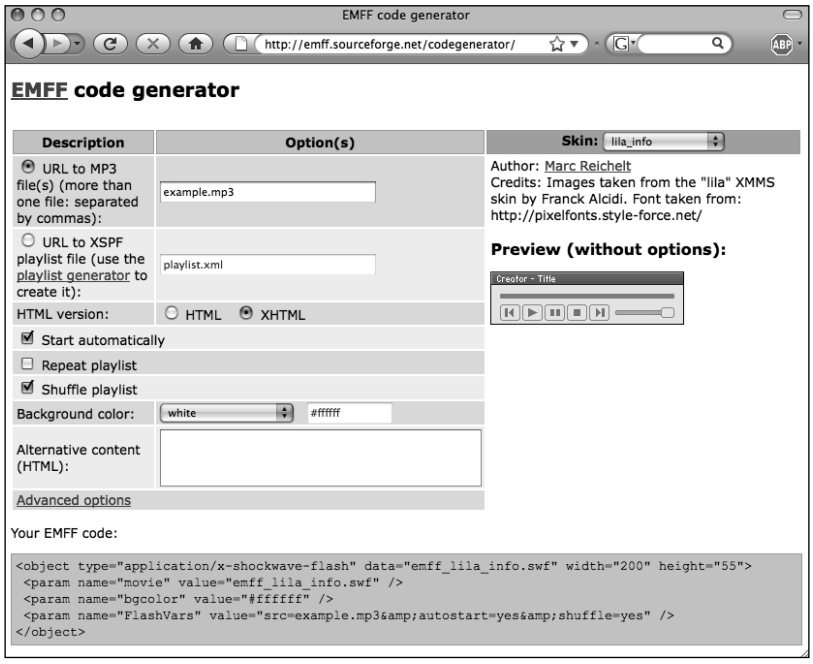
We utilize the `p4a.audio.interfaces.IAudioEnhanced` interface to register our view on.

Let's do so:

```
<browser:page
      for="p4a.audio.interfaces.IAudioEnhanced"
      name="my-audioplayer-view"
      class=".browser.AudioPlayerView"
      permission="zope2.View"
      template="player.pt"
      />
<browser:resourceDirectory
      directory="thirdparty/emff"
      name="emff"
      />
```

The page is named `my-audioplayer-view` and has the `AudioPlayerView` view class in the browser module. Further, we register a `thirdparty/emff` directory where we can put the Flash resources of the Flash player. Next, we need to create this view class and add the `player.pt` template.

We fill the template with the HTML code we get from the EMFF code generator:



Using the EMFF code generator

Choose the first option **URL to MP3**, though it doesn't really matter what you write into the text field. The value is overridden with the name we retrieve from our context object. For the HTML version, you can either choose HTML or XHTML as Plone 3 doesn't output valid XHTML itself. Nevertheless, XHTML might still be the better option, as it is future proof. Selecting XHTML closes the `param` elements inside of the `object` element.

We can copy this literally into our Zope page template.

```
<object type="application/x-shockwave-flash" data="emff_lila_info.swf"
        width="200" height="55">
 <param name="movie" value="emff_lila_info.swf" />
 <param name="bgcolor" value="#00ff00" />
 <param name="FlashVars" value="src=example.mp3&amp;autostart=yes" />

</object>
```

The Plone template is a standard one using the `main_template`. We copy the generated code from the bottom of the codegenerator window and put into the main slot of the template. There are just two changes we make. One is the location of the Flash file, which is registered as a resource, and the other is the audio file itself, which is our context.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      i18n:domain="p4a.audio"
      metal:use-macro="context/main_template/macros/master">
  <body>
    <div metal:fill-slot="main">
      <object type="application/x-shockwave-flash"
              data=" ++resource++emff/emff_lila_info.swf" width="200"
              height="55">
      <param name="movie" value="emff_lila_info.swf" />
      <param name="bgcolor" value="#ffffff" />
      <param name="FlashVars"
             value="src=example.mp3&amp;autostart=yes"
             tal:attributes="value string:src=${context/getId}&amp;
                                              autostart=yes"  />
      </object>
    </div>
  </body>
</html>
```

Next, we download the necessary Flash code from its website. The player is provided as a ZIP package with all sources and stuff included. We need to copy the chosen skin file to the `thirdparty/emff` directory of our `mm.enhance` product. In our example we used the `emff_lila_info` skin.

Using Flowplayer

If we want to use an audio Flash player on our site, but don't want to do any coding and are not satisfied with the one from `p4a.ploneaudio`, there is an alternative. It is possible to use the Flowplayer for our audio data. The Flash and JavaScript application Flowplayer is mainly designed to be a video player, but can be used as an audio player as well. Unfortunately, the only audio format the player understands is MP3. So if we have any other format we have to convert it to the MP3 format first. The Flowplayer is included with the Plone wrapper product `collective.flowplayer`. At the time of writing, the available version of the wrapper was 3.0b5. There are four ways in which we can use `collective.flowplayer` to include the audio into your site:

1. As a standalone player for files with the filename extension `.mp3`

2. As a playlist for folders containing MP3 files
3. As a portlet
4. Inline in any page supporting Rich Text (HTML)

The product itself can easily be added to our Plone setup by adding the following configuration lines to the instance in our `buildout.cfg`:

```
[instance]
...
eggs =
    ...
    collective.flowplayer
zcml =
    ...
    collective.flowplayer
```

After rerunning the buildout, restarting the instance, and installing **Flowplayer** via Plone's add-on products, we are ready to go.

Standalone Flowplayer for audio files

The most straightforward way to include the Flowplayer is to use the enhancing feature of the File content type. If we upload a file with an `.mp3` extension, the file is recognized as a supported audio file. It is marked with the `collective.flowplayer.interfaces.IAudio` interface and the default view is changed to `Flowplayer`. This view comes with `collective.flowplayer` and embeds a minimal Flowplayer instance into the page.

Combining p4a.ploneaudio and Flowplayer

If you have both the `p4a.ploneaudio` and `collective.flowplayer` products installed and you upload a file, it will get the default view of Flowplayer. The `p4a.audio.interfaces.IAudioEnhanced` interface still marks the file. Custom views binding on that interface will still work. For example you will still be able to access the `customskin` view you wrote binding on `IAudioEnhanced`. Calling the `view` view accesses the `p4a.ploneaudio` default view.

It is not necessary to store the audio files in the Plone site itself. With `collective.flowplayer` installed, Plone recognizes Link content objects pointing to audio data. If a URL ends with `.mp3`, Plone will change the default view to `flowplayer` accordingly. From the UI perspective, there is no difference. The only difference is the source of the audio file. While it is stored in the ZODB in the first case, it is fetched through a URL in the other case.

Playlist Flowplayer for audio containers

If we have folders or collections containing files with the `.mp3` extension or links pointing to MP3 files directly, we can provide a `flowplayer` view. All we have to do is to call the `flowplayer` view on the container. Alternatively, the `flowplayer` view can be chosen as default view for a container. The player looks the same as for single media, including two buttons between the play/pause button and the timeslide. These two buttons provide skip to the next/previous track functions.

Audio Flowplayer as a portlet

We can expose the Flowplayer as a portlet. All we need is a file or a container, as described before, as a data provider. We add the portlet as a **Video player** portlet in the **manage-portlets** view at the desired location in our site.

Inline audio player with Flowplayer

One killer feature of `collective.flowplayer` is the integration of the player applet in normal pages. With the simple addition of a certain CSS class, every link to an MP3 file can be turned into a Flowplayer. The code for doing this is here:

```
<a class="autoFlowPlayer audio" href="audio-file.mp3">
    This text is replaced.
</a>
```

If we only need a play button, the code will look like this:

```
<a class="autoFlowPlayer audio minimal" href="audio-file.mp3">
    This text is replaced.
</a>
```

For the integration of the Flowplayer in rich text, there is a shortcut in Kupu. We need to create a link pointing to an MP3 file in a separate paragraph. It doesn't matter if this link is internal or external. Then we choose one of **Audio**, **Audio (left)** or **Audio (right)** as the paragraph style.

To get an inline playlist, the following code is required:

```
<div class="playListFlowPlayer audio">
  <a class="playListItem" href="audio1.mp3">Audio one</a>
  <a class="playListItem" href="audio2.mp3">Audio two</a>
</div>
```

The documentation of the product suggests that the `div` element can be extended with the `random` option to get a randomized playlist.

Configuring Flowplayer

Look at the Flowplayer section in *Chapter 4, Managing Video Content* to get details about additional configuration, to get a detailed list of Flowplayer properties, and to know how to remove Flowplayer properly from a site.

Technology preview: HTML5

Let's peek into HTML5. HTML5 is the successor of HTML4. It supports the inclusion of audio and video without using Flash. There is a dedicated `audio` element for audio content. This element supports the following attributes:

Attribute	Value	Description
`autobuffer`	`autobuffer`	If present, the audio will be loaded at page load and will be ready to run. `autobuffer` has no effect if autoplay is present.
`autoplay`	`autoplay`	If present, the audio will start playing as soon as it is ready.
`controls`	`controls`	If present, the user is shown some controls, such as a play button.
`src`	URL	Defines the URL of the audio to play start

A player view with HTML5

Writing a player view with HTML5 is extremely easy with the `audio` element. Let's assume we have the following page definition in ZCML:

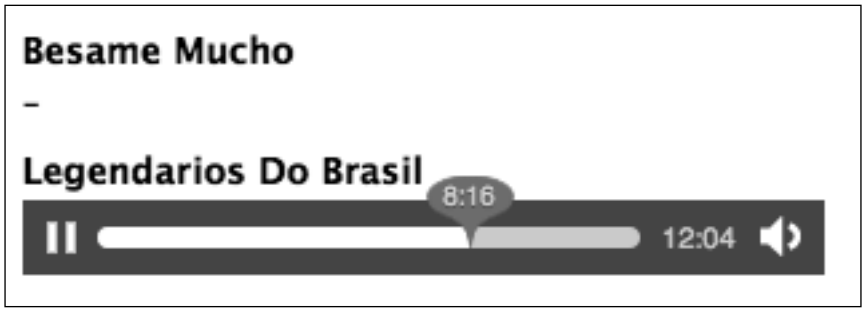
```
<browser:page
      for="p4a.audio.interfaces.IAudioEnhanced"
      name="newtechplayer"
      permission="zope2.View"
      template="html5player.pt"
      />
```

The `html5player.pt` page template contains only one element:

```
<div metal:fill-slot="main"
     tal:define="audio_info context/@@audio_view">
<h4 tal:content="context/title" /> -
<h4 tal:content="audio_info/artist" />
        <audio controls="controls"
          tal:attributes="src context/absolute_url">An audio player
        </audio>
    </div>
```

Browser support for HTML5

The described method works only with selected (preview) versions of current web browsers. When tested, the best results were with Safari 4 and Firefox 3.5. Opera 10 beta and Internet Explorer 8 don't seem to understand the `audio` element at all. And there is another limitation: Different browsers play different audio formats. While Firefox plays only the open codecs WAV, Ogg Vorbis and Ogg Theora; Safari seems to be limited to MP3. Bear this in mind when experimenting with HTML5.



Summary

In this chapter we saw how to include audio content into Plone. We saw how the File content type can be used to store audio data in Plone and how the data can be fetched from it. Additionally, we investigated the storage and the compression of audio data in general. We learned about different audio formats, and their advantages and disadvantages if using them in the web context.

Next we saw how we can enhance the audio features of Plone with the Plone4Artists product `p4a.ploneaudio`. We investigated the enhancement of single audio files, containers, and collections of audio files.

We learned how to include audio data in HTML with plugins and Flash. We wrote a simple player for our audio data and tried Flowplayer as an alternative player. Finally, we did a small preview on HTML5 where playing audio files is a core component of the Hypertext Markup Language.